INTERNATIONAL STANDARD

ISO/IEC 13817-1

First edition
1996-12-15

Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language —

Part 1: Base language

Technologies de l'information — Langages de programmation, leurs environnements et interfaces logiciel système — Méthode de développement de Vienne — Langage de spécification —

Partie 1: Langage de base

Reference number
ISO/IEC 13817-1:1996(E)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Contents

Page

Foreword ... xi

Introduction ... xii

1 Scope ... 1

2 Normative References ... 2

3 Definitions ... 3
3.1 Structure of Formal Definition ... 4
3.2 Conventions ... 4
3.2.1 Informative text ... 4

4 Conformity ... 5
4.1 Specifications ... 5

5 Basic Mathematical Notation ... 6
5.1 Logic Notation ... 6
5.2 Basic Set Theory ... 6
5.3 Cartesian Products ... 7
5.4 Binary Relations and Functions ... 8
5.5 Finite Sequences ... 9
5.6 Finite Mappings ... 10
5.7 Ordinal Numbers ... 11
5.8 Definition by Transfinite Induction ... 12
5.9 Cardinality and Cardinal Numbers ... 12
5.10 Structured Expressions ... 13
5.11 Semantic Function and Predicate Definitions ... 15
5.12 Use of Recursion ... 16

6 Core Abstract Syntax ... 17
6.1 Document ... 17
6.2 Definitions ... 17
6.2.1 Type Definitions ... 17
6.2.2 State Definition ... 19
6.2.3 Value Definitions ... 19
6.2.4 Function Definitions ... 19
6.2.5 Operation Definitions ... 20

© ISO/IEC 1996

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.3 Expressions . . . 21
6.3.1 Local Binding Expressions . . . 22
6.3.2 Conditional Expressions . . . 22
6.3.3 Unary Expressions . . . 22
6.3.4 Binary Expressions . . . 23
6.3.5 Quantified Expressions . . . 23
6.3.6 Iota Expression . . . 24
6.3.7 Set Expressions . . . 24
6.3.8 Sequence Expressions . . . 24
6.3.9 Map Expressions . . . 25
6.3.10 Tuple Constructor . . . 25
6.3.11 Record Expressions . . . 25
6.3.12 Apply Expressions . . . 25
6.3.13 Lambda Expression . . . 26
6.3.14 Is Expression . . . 26
6.3.15 Literals . . . 26
6.3.16 Identifiers . . . 27
6.4 State Designators . . . 28
6.5 Statements . . . 28
6.5.1 Local Binding Statements . . . 28
6.5.2 Block and Assignment Statements . . . 29
6.5.3 Conditional Statements . . . 29
6.5.4 Loop Statements . . . 29
6.5.5 Non-Deterministic Sequences . . . 30
6.5.6 Call and Return Statements . . . 30
6.5.7 Exception Handling Statements . . . 31
6.6 Patterns and Bindings . . . 31

7 Dynamic Semantic Domains . . . 34
7.1 The Domain Universe . . . 34
7.1.1 Basic Definitions . . . 34
7.1.1.1 Complete Partial Orders and Fixed Point Definitions . . . 34
7.1.1.2 Operators on Complete Partial Orders . . . 35
7.1.2 A Universe of Complete Partial Orders . . . 37
7.1.3 A Universe of Domains . . . 38
7.2 The Semantic Domains . . . 40
7.2.1 Basic Semantic Domains . . . 40
7.2.2 Extended Semantic Domains . . . 42
7.2.3 Semantic Domains for Evaluation Functions . . . 43
7.2.4 Auxiliary Semantic Domains . . . 45

8 The Dynamic Semantics . . . 46
8.1 Document . . . 46
8.2 Definitions . . . 48
8.2.1 Type Definitions . . . 48
8.2.1.1 The Verification Predicate for Types . . . 49
8.2.1.2 Evaluation Functions for Types . . . 50
8.2.2 State Definition . . . 57
8.2.3 Value Definitions . . . 57
8.2.3.1 The Verification Predicate for Value Definitions . . . 58
8.2.3.2 The Evaluation Function for Value Definitions . . . 61
8.2.4 Function Definitions . . . 63
8.2.4.1 Verification Predicates for Function Definitions . . . 63
8.2.4.2 Evaluation Functions for Polymorphic Functions . . . 67
8.2.5 Operation Definitions . . . 69
8.2.5.1 Verification Predicates for Operation Definitions . . . 69
8.2.5.2 Evaluation Functions for Operation Definitions . . . 70

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3 Expressions . . . 72
8.3.1 Local Binding Expressions . . . 74
8.3.2 Conditional Expressions . . . 76
8.3.3 Unary Expressions . . . 78
8.3.3.1 Numeric Operations . . . 78
8.3.3.2 Logical Operation . . . 79
8.3.3.3 Set Operations . . . 79
8.3.3.4 Sequence Operations . . . 80
8.3.3.5 Map Operations . . . 81
8.3.4 Binary Expressions . . . 82
8.3.4.1 Numeric Operations . . . 82
8.3.4.2 Numeric Comparison Operators . . . 85
8.3.4.3 Equality Operations . . . 85
8.3.4.4 Logical Operations . . . 86
8.3.4.5 Logical Set Operations . . . 87
8.3.4.6 Set Operations . . . 88
8.3.4.7 Sequence Operation . . . 88
8.3.4.8 Sequence and Map Modification Operation . . . 89
8.3.4.9 Map Operations . . . 89
8.3.4.10 Compose and Iterate . . . 90
8.3.5 Quantified Expressions . . . 93
8.3.6 Iota Expression . . . 98
8.3.7 Set Expressions . . . 99
8.3.8 Sequence Exprcssions . . . 104
8.3.9 Map Expressions . . . 106
8.3.10 Tuple Constructor . . . 108
8.3.11 Record Expressions . . . 108
8.3.12 Apply Expressions . . . 110
8.3.13 Lambda Expression . . . 111
8.3.14 Is Expression . . . 112
8.4 State Designators . . . 113
8.5 Statements . . . 117
8.5.1 Underlying Theory For Statements . . . 117
8.5.2 The Statement Evaluation Functions . . . 119
8.5.3 Local Binding Statements . . . 120
8.5.4 Block and Assignment Statements . . . 122
8.5.5 Conditional Statements . . . 124
8.5.6 Loop Statements . . . 126
8.5.7 Non-Deterministic Sequences . . . 129
8.5.8 Call and Return Statements . . . 130
8.5.9 Exception Handling Statements . . . 131
8.6 Patterns and Bindings . . . 135
8.6.1 Patterns . . . 136
8.6.1.1 Auxiliary Functions . . . 141
8.6.2 Bindings . . . 144
8.7 Auxiliary Functions and Predicates . . . 144
8.7.1 Expansion Functions . . . 144
8.7.2 Functions for Extending the Environment . . . 147
8.7.3 Functions and Predicates to deal with the State . . . 151
8.7.4 Functions and Predicates to deal with Curried Functions . . . 153
8.7.5 Compute Functions . . . 154
8.7.6 Generate Functions . . . 156
8.7.7 *Make* Functions . . . 157
8.7.8 *Get* Functions . . . 159
8.7.9 Collector Functions . . . 163
8.7.10 Selector Functions . . . 166
8.7.11 Tag-processing Functions . . . 169

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.12 General Functions and Predicates . . . 171

9 The Mathematical Concrete Syntax . . . 178
9.1 Document . . . 178
9.2 Definitions . . . 178
9.2.1 Type Definitions . . . 178
9.2.2 State Definition . . . 179
9.2.3 Value Definitions . . . 179
9.2.4 Function Definitions . . . 179
9.2.5 Operation Definitions . . . 179
9.3 Expressions . . . 180
9.3.1 Bracketed Expressions . . . 180
9.3.2 Local Binding Expressions . . . 180
9.3.3 Conditional Expressions . . . 180
9.3.4 Unary Expressions . . . 181
9.3.5 Binary Expressions . . . 181
9.3.6 Quantified Expressions . . . 182
9.3.7 Iota Expression . . . 183
9.3.8 Set Expressions . . . 183
9.3.9 Sequence Expressions . . . 183
9.3.10 Map Expressions . . . 183
9.3.11 Tuple Constructor Expression . . . 183
9.3.12 Record Expressions . . . 183
9.3.13 Apply Expressions . . . 183
9.3.14 Lambda Expression . . . 183
9.3.15 Is Expressions . . . 183
9.3.16 Names . . . 183
9.4 State Designators . . . 183
9.5 Statements . . . 184
9.5.1 Local Binding Statements . . . 184
9.5.2 Block and Assignment Statements . . . 184
9.5.3 Conditional Statements . . . 184
9.5.4 Loop Statements . . . 184
9.5.5 Nondeterministic Statement . . . 184
9.5.6 Call and Return Statements . . . 185
9.5.7 Exception Handling Statements . . . 185
9.5.8 Identity Statement . . . 185
9.6 Patterns and Bindings . . . 185
9.6.1 Patterns . . . 185
9.6.2 Bindings . . . 185
9.7 Lexical Specification . . . 185
9.7.1 General . . . 185
9.7.2 Characters . . . 186
9.7.3 Symbols . . . 188
9.8 Operator Precedence . . . 188
9.8.1 The Family of Combinators . . . 189
9.8.2 The Family of Applicators . . . 189
9.8.3 The Family of Evaluators . . . 189
9.8.4 The Family of Relations . . . 190
9.8.5 The Family of Connectives . . . 190
9.8.6 The Family of Constructors . . . 190
9.8.7 Grouping . . . 190
9.8.8 The Type Operators . . . 191

10 The Interchange Concrete Syntax . . . 192
10.1 Introduction . . . 192
10.2 Lexis . . . 192

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10.3 Symbols 192

11 The Outer Abstract Syntax 196
11.1 Document 196
11.2 Definitions 196
11.2.1 Type Definitions 196
11.2.2 State Definition 197
11.2.3 Value Definitions 198
11.2.4 Function Definitions 198
11.2.5 Operation Definitions 198
11.3 Expressions 199
11.3.1 Bracketed Expressions 200
11.3.2 Local Binding Expressions 200
11.3.3 Conditional Expressions 200
11.3.4 Unary Expressions 201
11.3.5 Binary Expressions 201
11.3.6 Quantified Expressions 202
11.3.7 Iota Expression 202
11.3.8 Set Expressions 202
11.3.9 Sequence Expressions 203
11.3.10 Map Expressions 203
11.3.11 Tuple Constructor Expression 203
11.3.12 Record Expressions 203
11.3.13 Apply Expressions 203
11.3.14 Lambda Expression 204
11.3.15 Is Expressions 204
11.3.16 Names 204
11.4 State Designators 204
11.5 Statements 204
11.5.1 Local Binding Statements 205
11.5.2 Block and Assignment Statements 205
11.5.3 Conditional Statements 205
11.5.4 Loop Statements 206
11.5.5 Nondeterministic Statement 206
11.5.6 Call and Return Statements 206
11.5.7 Exception Handling Statements 206
11.5.8 Identity Statement 206
11.6 Patterns and Bindings 207
11.6.1 Patterns 207
11.6.2 Bindings 207
11.7 Lexical Specification 208

12 The Syntax Mapping 210
12.1 Structure and Style of the Definition 210
12.1.1 Division into Modules 210
12.1.1.1 Module "OAS2CAS" 210
12.1.1.2 Module "OAS" 210
12.1.1.3 Module "CAS" 210
12.1.1.4 Module "GetUnusedId" 210
12.1.2 Pre-conditions in the VDM-SL Definition of the Syntax Mapping 211
12.1.3 Transformation of a Document to CAS.Definitions 211
12.1.3.1 The Introduction of Additional Identifiers 211
12.1.3.2 The Generation of Quoted Pre- and Post-conditions for Functions 212
12.1.3.3 The Generation of Quoted Pre- and Post-conditions for Operations 212
12.1.3.4 The Transformation of Expressions 213
12.1.3.5 Pre-conditions of Operations 213
12.1.3.6 Guards of Error Handlers 214

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.1.3.7 Quoted Post-conditions of Implicit Operations . . . 214
12.1.3.8 The Transformation of Type Definitions . . . 214
12.1.3.9 The Transformation of Value Definitions . . . 214
12.1.3.10 The Transformation of the State . . . 214
12.1.4 Notational Conventions . . . 215
12.2 Syntaxes and Auxiliary Functions . . . 215
12.2.1 Module "OAS" . . . 215
12.2.2 Module "CAS" . . . 215
12.2.3 Module "GetUnusedId" . . . 216
12.3 The Syntax Mapping Functions . . . 216
12.3.1 Document . . . 217
12.3.2 Definitions . . . 220
12.3.2.1 Type Definitions . . . 220
12.3.2.2 State Definition . . . 224
12.3.2.3 Value Definitions . . . 225
12.3.2.4 Function Definitions . . . 225
12.3.2.5 Operation Definitions . . . 232
12.3.3 Expressions . . . 237
12.3.3.1 Bracketed Expressions . . . 239
12.3.3.2 Local Binding Expressions . . . 239
12.3.3.3 Conditional Expressions . . . 240
12.3.3.4 Unary Expressions . . . 241
12.3.3.5 Binary Expressions . . . 241
12.3.3.6 Quantified Expressions . . . 241
12.3.3.7 Iota Expression . . . 241
12.3.3.8 Set Expressions . . . 242
12.3.3.9 Sequence Expressions . . . 242
12.3.3.10 Map Expressions . . . 242
12.3.3.11 Tuple Constructor Expression . . . 243
12.3.3.12 Record Expressions . . . 243
12.3.3.13 Apply Expressions . . . 244
12.3.3.14 Lambda Expression . . . 244
12.3.3.15 Is Expressions . . . 245
12.3.3.16 Names . . . 245
12.3.4 State Designators . . . 246
12.3.5 Statements . . . 246
12.3.5.1 Local Binding Statements . . . 247
12.3.5.2 Block and Assignment Statements . . . 248
12.3.5.3 Conditional Statements . . . 248
12.3.5.4 Loop Statements . . . 249
12.3.5.5 NonDeterministic Statement . . . 250
12.3.5.6 Call and Return Statements . . . 250
12.3.5.7 Exception Handling Statements . . . 250
12.3.5.8 Identity Statement . . . 251
12.3.6 Patterns and Bindings . . . 251
12.3.6.1 Patterns . . . 251
12.3.6.2 Bindings . . . 252
12.3.7 Lexical Specification . . . 253
12.3.7.1 General . . . 253
12.3.7.2 Characters . . . 253
12.3.7.3 Symbols . . . 253

13 The Static Semantic Domains . . . 255
13.1 Type Representations . . . 255
13.1.1 Special Subclasses of Type Representations . . . 256
13.2 Environments . . . 257
13.2.1 Accessing Environments . . . 257

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13.2.2 Updating Environments . . . 258
13.3 Well-formedness Classifications . . . 258
13.4 Type Relations . . . 259
13.4.1 Subtypes . . . 259
13.4.2 Overlapping Subtypes, Disjoint Types and Overlapping Types . . . 264
13.4.3 Auxiliary Type Relations and Functions . . . 267
13.5 Extended Abstract Syntax . . . 273

14 The Static Semantics . . . 273
14.1 Documents . . . 274
14.1.1 Auxiliary Well-formedness Requirements . . . 275
14.2 Definitions . . . 276
14.2.1 Type and State Definitions . . . 276
14.2.1.1 Types and Type maps . . . 277
14.2.1.2 Extraction of Type Representations . . . 279
14.2.2 Value, Function, and Operation Definitions . . . 280
14.2.2.1 Simultaneous Definitions . . . 280
14.2.2.2 Ordered Definition Groups and Definition Sequences . . . 281
14.2.2.3 Definitions . . . 282
14.2.2.4 Definition Groups and their Ordering . . . 285
14.2.2.5 Ordering of Definitions . . . 286
14.2.2.6 Asserted Visible Environments . . . 287
14.2.2.7 Pre-Conditions . . . 289
14.3 Expressions . . . 289
14.3.1 Expression Characteristic Predicates . . . 290
14.3.2 Relaxations and Restriction of Predicates . . . 291
14.3.2.1 The Subsumption Rule . . . 291
14.3.2.2 The InType Rule . . . 291
14.3.2.3 Union Close . . . 292
14.3.2.4 Sub-Environment Based Checks . . . 292
14.3.3 Well-Formedness of Expressions . . . 294
14.3.4 Bracketed Expression . . . 295
14.3.5 Local Binding Expressions . . . 295
14.3.5.1 Let Expression . . . 295
14.3.5.2 Let Be ST Expression . . . 295
14.3.5.3 Def Expression . . . 296
14.3.6 Conditional Expressions . . . 296
14.3.6.1 If Expression . . . 296
14.3.6.2 Cases Expression . . . 297
14.3.7 Unary Expressions . . . 298
14.3.7.1 Numeric Operations . . . 299
14.3.7.2 Logical Operation . . . 300
14.3.7.3 Set Operations . . . 301
14.3.7.4 Sequence Operations . . . 302
14.3.7.5 Map Operations . . . 303
14.3.7.6 Map Inverse Expression . . . 304
14.3.8 Binary Expressions . . . 304
14.3.8.1 Numeric Operations . . . 305
14.3.8.2 Numeric Comparison Operators . . . 307
14.3.8.3 Equality operations . . . 308
14.3.8.4 Logical Operations . . . 308
14.3.8.5 Logical Set Operations . . . 310
14.3.8.6 Set Operations . . . 311
14.3.8.7 Sequence Operation . . . 311
14.3.8.8 Sequence and Map Modification Operation . . . 312
14.3.8.9 Map Operations . . . 313
14.3.8.10 Compose and Iterate . . . 315

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.9 Quantified Expressions . . . 316
14.3.10 Iota Expression . . . 317
14.3.11 Set Expressions . . . 318
14.3.12 Sequence Expressions . . . 319
14.3.13 MapExpressions . . . 320
14.3.14 Tuple Constructor . . . 321
14.3.15 Record Expressions . . . 322
14.3.16 Apply Expressions . . . 323
14.3.17 Lambda Expressions . . . 324
14.3.18 Is Expressions . . . 326
14.3.19 Names . . . 326
14.3.20 Literals . . . 326
14.3.21 Auxiliary Well-formedness Predicates . . . 327
14.4 State Designators . . . 327
14.5 Statements . . . 328
14.5.1 Local Binding Statements . . . 329
14.5.2 Block and Assignment Statements . . . 330
14.5.3 Conditional Statements . . . 332
14.5.4 Loop Statements . . . 333
14.5.5 Non-Deterministic Statement . . . 334
14.5.6 Call and Return Statements . . . 335
14.5.7 Exception Handling Statements . . . 335
14.5.8 Identity Statements . . . 337
14.6 Patterns and Bindings . . . 337
14.6.1 Patterns . . . 338
14.6.1.1 Pattern Characteristic Predicates . . . 338
14.6.1.2 Relaxation of Pattern Characteristic Predicates . . . 340
14.6.1.3 Well-formedness of Patterns . . . 340
14.6.2 Bindings . . . 343
14.6.3 Value Environment . . . 346
14.7 Auxiliary Functions . . . 347
14.7.1 Dependency Relations . . . 347
14.7.1.1 Free and Defined Names . . . 348
14.7.1.2 Syntactical Sub-Components . . . 356
14.7.1.3 The Definitional Basis of Types . . . 361
14.7.2 Syntax Transformations . . . 361
14.7.3 Substitutions . . . 363
14.7.4 Indirectly Defined Functions . . . 368
14.7.5 Classification Functions . . . 371

A Extensions . . . 372

B Tool Conformity . . . 373
B.1 Semantic conformity of tools . . . 373

C Modules . . . 375
C.1 Overview . . . 375
C.2 Requirements of Modularization . . . 375
C.2.1 Language Facilities . . . 375
C.2.1.0.1 Syntactic Separation . . . 375
C.2.1.0.2 Explicit Import . . . 375
C.2.1.0.3 Explicit Export . . . 375
C.2.1.0.4 Parameterization . . . 376
C.2.2 Semantics . . . 376
C.3 Extant Approaches to Structuring Specifications . . . 376
C.3.1 The Syntactic Approach . . . 376
C.3.2 The Z Approach . . . 376

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

C.3.3 The Manchester Approach (Fitzgerald & Jones) . . . 376
C.3.4 The VVSL Approach (Middelburg) . . . 377
C.3.5 The RAISE Approach . . . 377

D Cross-references . . . 379
D.1 Cross References for the Dynamic Semantics . . . 379
D.1.1 Naming and Typesetting Conventions Used . . . 379
D.1.2 Listing of Functions/Predicates: Alphabetic (uses) . . . 380
D.2 Cross References for the Concrete Syntax . . . 385
D.3 Cross References for the Abstract Syntax . . . 388
D.4 Cross References for the Syntax Mapping . . . 390
D.5 Cross References for the Static Semantics . . . 393

E Bibliography . . . 397

Figures

1 Structure of the syntax mapping . . . 210

Tables

1 Character set . . . 187

2 Interchange syntax: representation of symbols . . . 192

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

International Standard ISO/IEC 13817-1 was prepared by Joint Technical Committee, ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

ISO/IEC 13817 consists of the following parts, under the general title *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language*:

— *Part 1: Base language*

Additional parts will specify modules and the development method.

Annexes A to E of this part of ISO/IEC 13817 are for information only.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Introduction

Historical background of the Vienna Development Method (VDM)

VDM was developed at the IBM Laboratory in Vienna. The Laboratory came into existence in 1961 when Professor Heinz Zemanek of the Technical University in Vienna decided to move his whole group to an industrial home.[1)] They had previously developed a computer called Mailüfterl at the Technical University. From 1958 the group had been increasingly involved in software projects including the construction of one of the early compilers for the ALGOL 60 programming language. As time went on they found it difficult to get adequate support for their projects and eventually joined IBM. In the mid-1960s, IBM decided to develop a new programming language for which the ambition was to replace both FORTRAN and COBOL. The language, which was at first called New Programming Language (until the National Physical Laboratories in the UK objected to the acronym — the language became known as PL/I), was clearly going to be large and it was decided that it would be useful to try to apply formal techniques to its description.

Based on their own work — and influenced by research work by Cal Elgot, Peter Landin and John McCarthy — the Vienna group developed an operational semantics definition of PL/I which they called ULD-3 ('Universal Language Description'; ULD-2 was the name applied to the IBM Hursley contribution to this effort — the language itself was being developed mainly from Hursley along with the early compilers; ULD-1 was a term applied to the natural language description of the language). The description of PL/I in ULD-3 style ran through three versions. These are very large documents. Operational semantics is now seen as unnecessarily complicated as compared to denotational semantics. However, to make the principles of denotational semantics applicable to a language like PL/I with arbitrary transfer of control, procedures as arguments, complicated tasking, etc. required major theoretical break-throughs and a considerable mathematical apparatus not available at the time. The effort of the formal definition uncovered many language problems early and had a substantial influence on the shape of the language.

Towards the end of the 1960s serious attempts were made to use the ULD-3 description as the basis of compiler designs. Many problems were uncovered: in general, one could say that the over-detailed mechanistic features of the operational semantics definition considerably complicated the task of proving that compiling algorithms were correct. But again one should be clear that the work was a technical achievement: a series of papers was published that described how various programming language concepts could be mapped into implementations which could be proved correct from the description (e.g. in JL71). In addition, a series of proposals was made which could simplify the task of developing compilers from a semantic description. One of these was an early form of an exit construct (HJ70) which led to an interesting difference between the Vienna flavour of denotational semantics and that used in Oxford. Another VDM-like idea that arose at this time was Peter Lucas' *twin machine* proof (Luc68), and subsequently the observation that the ghost variable treatment in the twin machine could be replaced by retrieve functions (Jon70) as a simpler way of proving most cases of data development are correct. It is worth noting that Lucas' *twin machine* idea has been re-invented several times since: the generalization of retrieve functions to relations can be seen as equivalent to twin machines with invariants.

Hans Beckič spent some time in England with Peter Landin at Queen Mary College and was the person who first pushed the Vienna group in the direction of denotational semantics. (Until his untimely death in 1982, Hans Beckič

[1)] This is not intended to be a history of the Vienna Laboratory: citations are limited to those concerning VDM itself.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

had published relatively little of his scientific research; some of his important papers were published posthumously in Bek84.) Another crucial stimulus was the visit to the Vienna laboratory by Dana Scott in 1969 (see dBS69).

During the period from 1971 to 1973, the Vienna group was diverted into other activities not really related to formal description. Cliff Jones at this time went back to the Hursley Laboratory and worked on a *functional* language description (ACJ72) and other aspects of what has become known as VDM. In particular he published a development of Earley's recogniser (Jon72) which is one of the first reports to use data reification. In late 1972 and throughout '73 and '74 the Vienna group (Cliff Jones returned and Dines Bjørner was recruited) had the opportunity to work on a PL/I compiler for what was then a very novel machine architecture. They of course decided to base their development for the compiler on a formal description of the programming language. PL/I was then undergoing ECMA/ANSI standardization (ANS76). The Vienna group chose to write a denotational semantics for PL/I (BBH$^+$74); this is the origin of the VDM[1)] work on language description techniques.

During the same period, at the IBM Laboratory in Hursley, an investigation into the use of Meta-IV to formaliy describe five of the major languages supported by IBM was carried out. The languages were PL/I, BASIC, FORTRAN, APL and COBOL. Sketches for parts of FORTRAN and APL were written, and a full description of minimal BASIC was produced. This work ceased when the language work was moved from Hursley.

Cliff Jones and Dines Bjørner took upon themselves the task of making sure that something other than technical reports existed to describe the work that had gone on on the language aspects of VDM: Be78 is a first book-length description of that work. In ESRI, Cliff Jones also developed the work on those aspects of VDM not specifically related to compiler development and the first book on what is now generally thought of as VDM is Jon80. Both of these books have now been superceded: the language description work is best accessed in Be82 and — in its second edition — the non-language work is best seen in Jon90 and also in AI91.

Within IBM, from 1978, a number of projects used VDM (for other than language descriptions). It was during this period that specifications of several large systems were carried out at the Böblingen Laboratory. (These included a file system for DOS and an fault report tracking system; a data reification for the file system was attempted to show that the proof techniques were viable in an industrial environment.) This work was carried out as part of a technology transfer program. Later the IBM Program Product Development Centre in Rome became involved, and work was done to formally specify a hotel management system. This work showed that VDM was suitable for large-scale projects; unfortunately little has been published on this experience.

Dines Bjørner's group at the Technical University of Denmark strenuously pursued the use of VDM for language description and he and his colleagues were responsible for descriptions of the CHILL programming Language and a major effort to document the semantics of the Ada programming language (BO80).

Language work was also continued at Leicester University where a formal definition of the full Pascal language was written (AH82) and later a formal definition of the programming language Modula-2, which became a Draft International Standard (AeWO).

The non-language, specification, aspects of VDM were taken up by the STL laboratory in Harlow and, partly because of their industrial push, the British Standards Institute (BSI) was persuaded to establish a standardization activity. This activity has not been easy because of the differences between the pressures of those interested in the language description aspects of VDM and those who are more interested in pre- and post-conditions, data reification and operation decomposition. It is to the credit of both the BSI and ISO Standards committee that they have managed to bear in mind the requirements of both types of user and come up with a standard that embraces such a wide scope of technical ideas. STL was responsible for funding the first formal semantics of VDM-SL and the work that was done at Manchester University by Brian Monahan (Mon87) was used as the starting point of the formal description of the specification language. A new formal description based on this work was produced because of the necessity of merging together the two aspects of the specification language.

The outcome of the standardization effort initiated by STL through the *British Standards Institution (BSI)* was the formation of a panel (BSI IST/5/-/50) whose membership was also open to participants from non-British organizations.

[1)] It is worth getting some acronyms sorted out: VDL stands for Vienna Description Language and was a term used to describe the operational semantics (ULD-3) notation; VDM stands for Vienna Development Method and relates to the post-1973 work; the specification language part of VDM is sometimes known as 'Meta-IV'; it is now known as VDM-SL.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

In 1991 the need for a VDM-SL standard was also recognized by *ISO/IEC JTC1*[2)] by the formation of a Working Group, SC22/WG19. The actual work on the standard is done by the members of the BSI Panel.

Work on the Standard encouraged the building of computer-based tools to support both the language and the development method. Adelard produced 'SpecBox', a tool that provided syntax and type checking of VDM specifications. Manchester University and the Rutherford Appleton Laboratories wrote 'mural' (JJLM91), a prototype tool that supports both the specification language and the development method, providing a proof engine to help with data reification and operation decomposition. The Technical University of Denmark and the National Physical Laboratory have configured the Cornell Synthesizer (a structured editor tool) to support VDM-SL with some type-checking facilities — this tool was used to produce the static semantics of VDM-SL. IFAD have produced a tool for syntax and type checking together with an interpreter for an executable subset to allow rapid prototyping.

During its history, the Vienna Development Method, together with its specification languages has had a profound influence on both the specification of programming languages and the specification and development of systems. The ideas in VDM have influenced several other specification languages including RAISE, COLD-K and VVSL.

[2)] Joint Technical Committee 1 of the International Standards Organization and the International Electro-technical Commission.

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language —

Part 1:
Base language

1 Scope

This part of ISO/IEC 13817 specifies the model based specification language VDM-SL (Vienna Development Method — Specification Language). It specifies:

- two representations: the mathematical and interchange;
- the syntax;
- the static semantics;
- the dynamic semantics;
- conformity for specifications and tools.

It does not specify:

- the proof obligations;
- the reification rules;
- the size or complexity of a specification that will exceed the capacity of any specific data processing system or the capacity of a particular tool, nor the actions to be taken when the corresponding limits are exceeded;
- the minimal requirements of a data processing system that is capable of supporting an implementation of a tool;
- the method that tools use for reporting errors.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

2 Normative References

The following standards contain provisions which, through reference in this text, constitute requirements of this part of ISO/IEC 13817. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 13817 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO 8859-1:1987[1)], *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1.*

ISO/IEC 14977:1996, *Syntactic Metalanguage — Extended BNF.*

1) Currently under revision.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3 Definitions

For the purposes of this part of ISO/IEC 13817, the following definitions apply.

3.1 conforming specification: A specification which is written in the language defined by this part of ISO/IEC 13817 and which obeys the conformity clauses for specifications in this part of ISO/IEC 13817.

3.2 conforming tool: A tool which processes conforming specifications and which obeys the conformity clauses for tools in this part of ISO/IEC 13817.

3.3 consistent specification: A specification that satisfies either the mathematical or interchange concrete syntax and can be translated by the syntax mapping into a representation that uses the core abstract syntax and for which the dynamic semantics defines at least one model.

3.4 core abstract syntax: The core abstract syntax gives a second abstract version of the underlying structure of the language that uses a reduced number of constructs. Translation between the outer abstract syntax and the core abstract syntax is, (in general) explicitly defined by a mapping function (see clause 12). In a few cases, where the core abstract syntax is not clearly related to the outer abstract syntax, translation or technical notes (annotations) are used to explain the translation.

3.5 dynamic semantics: Dynamic semantics are the rules for the derivation of the models of a specification that satisfies either the mathematical or interchange concrete syntax and can be translated by the mapping function into a representation that uses the core abstract syntax. The dynamic semantics are expressed using a mathematical notation that is defined in clause 5. The meaning of a consistent specification is given by a set of models that the specification defines.

3.6 error: A fault in a specification which means the specification could be a meaningless specification or the specification is inconsistent.

3.7 extension: An addition to the requirements of this part of ISO/IEC 13817 that does not invalidate any specification conforming with this standard except by prohibiting the use of one or more particular spellings of identifiers. This implies that any new semantic definitions that are introduced must be be consistent with those in this part of ISO/IEC 13817.

3.8 inconsistent specification: A specification that is not consistent — the set of models defined for the specification as defined by the dynamic semantics is empty.

3.9 interchange concrete syntax: The interchange concrete syntax is an alternative set of lexical rules from the Lexis (see clause 9.7) the representation of which uses standard symbols.

3.10 mathematical concrete syntax: The mathematical concrete syntax is a set of grammatical rules defining the syntactic structure of the language in terms of other concrete syntax rules, or lexical rules from the Lexis (see clause 9.7).

3.11 meaningless specification: A specification which cannot be translated by the syntax mapping to the core abstract syntax and thus for which no semantics can be given.

3.12 model: A mathematical construction designated as the meaning of a specification (or part of such specification) as defined by the dynamic semantics.

3.13 outer abstract syntax: The outer abstract syntax gives an abstract version of the underlying structure of the language. Translation between the concrete and abstract syntax is implicit. In a few cases, where the abstract syntax is not clearly related to the concrete syntax, translation or technical notes (annotations) are added to explain any apparent divergence.

3.14 proof obligations: The set of proofs that are necessary to show that a specification is semantically correct.

3.15 reification rules: A set of rules by which it is possible to express that one specification satisfy another more abstract specification. The first specification can then be considered an implementation of the latter.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

3.16 specification: A description of an entity in a formal language.

3.17 static semantics: Static semantics are the rules expressed in the form of truth-valued functions (i.e. the 'well-formedness' predicates) on components of the abstract syntax that are to be applied by a tool to determine the legality of a specification written in VDM-SL. The static semantics are expressed in VDM-SL that conforms to the dynamic semantics. The static semantics are a subset of the dynamic semantics that can be checked automatically (by tools).

3.18 system: Software for controlling the execution of tools.

3.19 syntax-conformant: A specification that is not a meaningless specification.

3.20 testable tool: A tool is testable if it is able to process specifications in such a manner that a user does not have to further influence its processing.

3.21 tool: The entire computing system which enables the specification user to manipulate specifications, in general consisting both of hardware and of the relevant associated software.

3.22 undefined specification: A specification that is not a meaningless specification, but it has not yet been determined whether it is a consistent specification or an inconsistent specification.

3.1 Structure of Formal Definition

This part of ISO/IEC 13817 uses two formal notations and natural language to state its requirements. For the most part the form and meaning of VDM-SL specifications is expressed using a formal definition accompanied by natural language normative statements and notes. In general, the formal definition of the VDM specification language (VDM-SL) is divided into five parts: the mathematical concrete syntax and the interchange concrete syntax that describe the representation of a specification; the outer abstract syntax and the core abstract syntax that describe abstract representations of a specification — these two abstract specifications are linked by a mapping function that translates the outer abstract syntax into the core abstract syntax; and the dynamic and static semantics. The mapping of either of the concrete representations of a specification to the outer abstract syntax can be performed by constructing an appropriate parse tree representation and removing those lexenes that are providing punctuation.

The mathematical notation used to define the dynamic semantics is defined in this part of ISO/IEC 13817 (see clause 5).

Where necessary, parts of the VDM-SL are explained by means of natural language comments labelled as annotations.

3.2 Conventions

The following conventions are used in this part of ISO/IEC 13817 and shall be interpreted as described below.

3.2.1 Informative text

At certain points in this part of ISO/IEC 13817, informative, rather than normative text appears. In particular, such text often appears immediately after sub-clause headings (before the specification of concrete syntax or static semantics), and within the confines of clauses where it is demarcated as notes, clarifications, and changes, as exemplified below.

NOTE — Notes are usually printed in a smaller font than normative text.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

4 Conformity

4.1 Specifications

A VDM-SL specification conforms to the requirements of this part of ISO/IEC 13817 if and only if it is derivable according to the syntactic rules of either

clause 9 or

clause 10

and is syntax-conformant or consistent with respect to

clause 6,

clause 7,

clause 8,

clause 11 and

clause 12.

NOTE — For recommendations of how to handle extensions to VDM-SL and for tool conformance, see Annex A and Annex B.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5 Basic Mathematical Notation

This clause introduces the mathematical notation that will be the "meta-language" employed to define the dynamic semantics of VDM-SL at the set-theoretic level.

The formal definition of the semantics of VDM-SL is given in a model-based style by stating a function that maps VDM-SL specifications to members of a space of models that is rich enough to provide denotations for all possible specifications.

In this clause the basic mathematical notation used for defining this function is introduced. It will be basic in the sense that the operators defined in this clause will work upon unordered sets.

The construction is based on ZF set-theory by assuming the existence of certain basic sets and operations on sets and using these to build spaces of values — as well as operations for manipulating these — suitable for defining the semantics of VDM-SL.

The main purpose of this clause is to fix the notation at the set-theoretic level, which is going to be used in the following clauses. The logic notation and the basic set-theoretic notation will be fixed first. Then the notation for mathematical structures like Cartesian products, relations, functions, finite sequences, and finite mappings will be fixed in that order. Because it is necessary to use so-called transfinite induction in the construction of the domain universe (in the next clause) the ordinal numbers are introduced as a basis for understanding transfinite induction. Since it is also necessary to find the cardinality of infinite sets the cardinal numbers are also introduced. Finally the notation for structured expressions (e.g. if-then-else etc.) and for functions and predicates which are going to be used in the definition of the dynamic semantics are fixed.

5.1 Logic Notation

The logic employed in the semantics definition is two-valued predicate logic with equality. The following notation is used:

T, F	truth valued constants
$\neg P$	negation
$P \wedge Q$	conjunction
$P \vee Q$	disjunction
$P \Rightarrow Q$	implication
$P \Leftrightarrow Q$	equivalence
$P = Q$	equality

5.2 Basic Set Theory

All of the constructions in the semantics are given within a "naïve" set theory; that is, a rigorous, but informal theory of arbitrary sets. It is widely accepted that this naïve theory of sets can be formalised in several ways. One of the best known axiomatisations of sets is known as Zermelo-Fraenkel set theory (or ZF). It has been shown, in practice, that this theory is powerful enough to give a set-theoretical account of much of classical mathematics, and is hence adequate for the needs of the metalanguage. The aim of this clause is solely to fix the notation concerning sets.

The intuitive concept of a *set* as a collection of objects, called *elements* or *members* of the set, is the basis for the logic. The notation $a \in A$ means that a is an element of the set A. If a is not an element of A, this is written $a \notin A$. If A and B are sets, $A \subseteq B$ denotes *inclusion*, that is, that A is a *subset* of B, or, all the elements of A are also in B. *Equality* of the sets A and B, in symbols $A = B$, holds iff $A \subseteq B$ and $B \subseteq A$. If $A = B$ does not hold, $A \neq B$ is written. *Proper inclusion* is denoted by $A \subset B$, and means $A \subseteq B$ and $A \neq B$.

The empty set is denoted by $\{\ \}$and $\emptyset$; note that $\{\ \} \subseteq A$ for every set A.

The *set-theoretic operations* $\cup$, $\cap$, $\bigcup$, $\bigcap$, $\setminus$ (they are called *union*, *intersection*, *distributed union*, *distributed inter-*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

section and *difference*, respectively) have their accepted mathematical meaning. Note that $B \subseteq A$ is equivalent to $B = B \cap A$, which, in turn, is equivalent to $A = B \cup A$. If $A \cap B = \{\ \}$, A and B are said to be *disjoint.*

For any set A, $\mathbb{P}(A)$ denotes the powerset of A, i.e. the set of all subsets of A. Similarly, $\mathbb{F}(A)$ denotes the set of all finite subsets of A.

Within the mathematical notation two different ways will be used to construct sets. Either the elements will be enumerated and the construction will look like $\{a_1, a_2, \ldots, a_n\}$. Alternatively the set will be constructed in a more implicit manner by means of a set comprehension which looks like $\{\ f(a) \mid a \in A \cdot P(a)\}$. Both of these constructors have their usual meaning.

Quantification over well-defined sets is also used. Universal quantification will be denoted by $\forall a{\in}A \cdot P$. Existential quantification will be denoted by $\exists a{\in}A \cdot P$. Finally, exist-unique quantification will be denoted by $\exists!a{\in}A \cdot P$. In all three cases the accepted rules for variable binding apply for the quantifiers. The range, A, should be a well-defined (but unrestricted) set.

A *family* $(a_i \mid i{\in}I)$ of elements of a set A forms a function $\varphi : I \to A$ such that $\varphi(i) = a_i$ for all $i{\in}I$. The set of all such families will be denoted by $I \to A$. For a family $\varphi = (a_i \mid i{\in}I)$ the image of I under φ will be denoted by $\{a_i \mid i{\in}I\}$.

The built-in operator, $\underline{\text{card}}(A)$, is used to get the cardinality of a set A. Normally this will be used for finite sets, however in a few places it is also used for possibly infinite sets. This will be further explained in a clause about cardinal numbers (see 5.9).

Finally, some basic domains (such as Booleans ($\mathbb{B}$), the natural numbers ($\mathbb{N}$), integers ($\mathbb{Z}$), the rational numbers ($\mathbb{Q}$), reals ($\mathbb{R}$)) are used and appropriate operators on them (such as plus (+), minus (−), multiplication, (×), less than (<), etc.). These basic operators will be used in their usual infix notation, but a few will also be used as built-in prefix operators (e.g. $\underline{\text{abs}}$, the absolute value).

5.3 Cartesian Products

The (binary) Cartesian product of two sets A, B is defined by:

$$A \times B = \{(a, b) \mid a \in A, b \in B\}$$

where the pairing operator (-, -) satisfies the property:

$$\forall a, b, c, d \cdot (a, b) = (c, d) \Leftrightarrow (a = c) \wedge (b = d)$$

The (generalized) Cartesian product of n sets $A_1, \ldots, A_n$ is defined by:

$$\bigtimes_{i=1}^{n} A_i = \{\ (a_1, \ldots, a_n) \mid\ a_1 \in A_1, \ldots\ , a_n \in A_n\ \}$$

Here the *tupling* operator (-, . . . , -) has the following property:

$$\forall a_1, a_1{}' \in A_1, \ldots, \forall a_n, a_n{}' \in A_n \cdot$$

$$(a_1, \ldots, a_n) = (a_1{}', \ldots, a_n{}') \Longleftrightarrow (a_1 = a_1{}') \wedge \ \ldots \wedge\ (a_n = a_n{}')$$

In addition inequality holds for tuples with different lengths:

$$\forall t_n \in \bigtimes_{i=1}^{n} A_i, t_m \in \bigtimes_{j=1}^{m} A_j \cdot n \neq m \Rightarrow t_n \neq t_m$$

If $A_1 = \ \ldots\ = A_n = A$ then $\bigtimes_{i=1}^{n} A_i$ will by denoted by A^n and called n'th Cartesian power of A.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

For an n-ary Cartesian product $\bigtimes_{i=1}^{n} A_i$ there is a family of n *projections* $\{\pi_1, \ldots, \pi_n\}$ such that for all $(a_i, \ldots, a_n) \in \bigtimes_{i=1}^{n} A_i$ and for all $k = 1, \ldots, n$:

$$\pi_k((a_1, \ldots, a_n)) = a_k$$

The tupling operator (-, . . . , -) and the projection functions have the following property:

$$\forall t \in \bigtimes_{i=1}^{n} A_i \cdot t = (\pi_1(t), \ldots, \pi_n(t))$$

The notation for the tupling operator and the projections is in a sense *polymorphic*. In fact these operators should be indexed by their size n and appropriate sets but for notational convenience this will not be done.

5.4 Binary Relations and Functions

A *binary relation* between sets A and B is an arbitrary subset of the Cartesian product $A \times B$.

In the case when $A = B$ a binary relation between A and B will be called a binary relation on A.

For sets A and B, $A \rightharpoonup B$ denotes the set of all *partial functions* from A to B. This set is defined by:

$$A \rightharpoonup B = \{ f \mid f \in \mathbb{P}(A \times B) \cdot \forall (a, b_1), (a, b_2) \in f \cdot b_1 = b_2 \}.$$

In addition to writing $f \in A \rightharpoonup B$ the more common notation $f : A \rightharpoonup B$ will also be used.

For every function $f : A \rightharpoonup B$, two sets called the *domain* and the *range* of f, respectively are defined by:

$$\delta_0(f) = \{a \mid a \in A \cdot \exists b \in B \cdot (a, b) \in f\}$$
$$\delta_1(f) = \{b \mid b \in B \cdot \exists a \in A \cdot (a, b) \in f\}.$$

For a function $f : A \rightharpoonup B$ and an element $a \in \delta_0(f)$ the *application* of f to a is defined to be the unique element $b \in \delta_1(f)$ such that $(a, b) \in f$. The notation $f(a)$ is used for function application.

For a function $f : A \rightharpoonup B$ and a set $A' \subseteq A$ a *restriction* of f to A' is a function $f\lceil A' : A' \rightharpoonup B$ such that $\delta_0(f\lceil A') = \delta_0(f) \cap A'$ and for all $a \in \delta_0(f\lceil A') \cdot f\lceil A'(a) = f(a)$.

A function $f : A \rightharpoonup B$ is called *injective* iff for all $a, a' \in \delta_0(f)$ if $a \neq a'$ then $f(a) \neq f(a')$.

The set of all *total functions* between sets A and B is a subset $A \to B \subseteq A \rightharpoonup B$ defined by:

$$A \to B = \{f \mid f : A \rightharpoonup B \cdot \delta_0(f) = A\}.$$

A total function $f : A \to B$ is called *bijective* iff it is injective and $\delta_1(f) = B$.

Given two partial functions $f : A \rightharpoonup B$ and $g : B \rightharpoonup C$, their *composition* is a function $g \circ f : A \rightharpoonup C$ defined by:

$$g \circ f = \{(a, g(f(a))) \mid a \in \delta_0(f) \cdot f(a) \in \delta_0(g)\}$$

If f and g are total functions then $g \circ f$ is total as well.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Functions are defined in typed λ-notation[1]. In order to define the meaning of a λ-expression the concepts of free and bound variables and substitution will briefly be introduced.

In an expression of the form $\lambda x{\in}A.e$ all occurrences of the variable x in the expression e are bound occurrences. All occurrences of variables in an expression that are not bound are free occurrences. $e[e'/x]$ is the expression resulting from substituting free occurrences of variable x in expression e with expression e'.

The meaning of $\lambda x{\in}A.e$ is defined as: if $e[a/x] \in B$ whenever $a \in A$ then $\lambda x{\in}A.e \in A \to B$ and denotes the function:

$$\{(a, b) \mid a \in A \cdot b = e[a/x]\}.$$

5.5 Finite Sequences

For a set A, $\mathbb{L}(A)$ denotes the set of all finite sequences of elements of A. It is defined by:

$$\mathbb{L}(A) = \bigcup_{n \in \omega} (\{1, \ldots, n\} \to A)$$

where ω is the first infinite ordinal, i.e. the set of all natural numbers (see also clause 5.7). Let $[\,]$ denote the empty sequence.

The set of all non-empty finite sequences of elements of A, $\mathbb{L}_1(A)$ is defined by:

$$\mathbb{L}_1(A) = \mathbb{L}(A) \backslash \{[\,]\}.$$

Sequences can be defined by using notations for defining functions, by enumeration of their elements, and by sequence comprehension. For $a_1, \ldots, a_n \in A$,

$$[\,a_1, \ldots, a_n\,]$$

is the sequence in $\mathbb{L}(A)$ equal to $(a_i \mid i \in \{1, \ldots, n\})$.

For every sequence $s \in \mathbb{L}(A)$, there exists a unique natural number $\underline{\text{len}}(s)$ such that $\{1, \ldots, \underline{\text{len}}(s)\} = \delta_0(s)$. This gives a function

$$\underline{\text{len}} : \mathbb{L}(A) \to \omega,$$

which for a given sequence returns its *length*.

Since a sequence $s \in \mathbb{L}(A)$ is a function, it can be applied to any element of its domain $\{1, \ldots, \underline{\text{len}}(s)\}$. Thus there is the (partial) *projection* function

$$_(_) : \mathbb{L}(A) \times \omega \xrightarrow{\sim} A$$

which maps $(s, n) \in \mathbb{L}(A) \times \omega$ to $s(n)$ when $1 \leq n \leq \underline{\text{len}}(s)$.

Now the definition by sequence comprehension can be stated: suppose f: $\omega \xrightarrow{\sim} A$, $D \in \mathbb{F}(\omega)$ such that $D \subseteq \delta_0(f)$, and P is a unary predicate on ω, then:

$$s = [f(i) \mid i{\in}\ D{\cdot}\ P(i)]$$

[1] In the definition of the dynamic semantics there will be places where the type will be omitted where it is "obvious" (e.g. $\lambda env \in ENV$ *body* will be abbreviated to λenv . *body*). This convention is used for notational convenience and it is considered that the mnemonics used (for variable identifiers) provide the reader with sufficient knowledge.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

defines a sequence (in $\mathbb{L}(A)$) of length $\underline{\text{card}}D'$ such that:

$$\forall i \in I \cdot s(i) = f(\varphi(i)),$$

where $I = \{1, \ldots, \underline{\text{card}}(D')\}$, $D' = \{i \in D \mid P(i)\}$, and $\varphi \in I \rightarrow D'$, is a function such that $\forall i, j \in I \cdot i < j \Rightarrow \varphi(i) < \varphi(j)$.

The following standard operators are also used on finite sequences:

$\underline{\text{hd}} : \mathbb{L}_1(A) \rightarrow A$
$\underline{\text{hd}}(s) = s(1)$

$\underline{\text{tl}} : \mathbb{L}_1(A) \rightarrow \mathbb{L}(A)$
$\underline{\text{tl}}(s) = [\, s(i) \mid i \in \{\, 2, \ldots, \underline{\text{len}}(s)\,\}\,]$

$\underline{\text{inds}} : \mathbb{L}(A) \rightarrow \mathbb{F}(\omega)$
$\underline{\text{inds}}(s) = \{\, 1, \ldots, \underline{\text{len}}(s)\,\}$

$\underline{\text{elems}} : \mathbb{L}(A) \rightarrow \mathbb{F}(A)$
$\underline{\text{elems}}(s) = \{\, s(i) \mid i \in \underline{\text{inds}}(s)\,\}$

$\underline{\text{join}} : \mathbb{L}(A) \times \mathbb{L}(A) \rightarrow \mathbb{L}(A)$
$\underline{\text{join}}([a_1, \ldots, a_n], [b_1, \ldots, b_m]) = [a_1, \ldots, a_n, b_1, \ldots, b_m]$

$\underline{\text{Join}} : \mathbb{L}(\mathbb{L}(A)) \rightarrow \mathbb{L}(A)$
$\underline{\text{Join}}(l_1, \ldots, l_n) = \underline{\text{join}}(l_1, \underline{\text{join}}(l_2, \ldots \underline{\text{join}}(l_{n\text{-}1}, l_n) \ldots))$.

5.6 Finite Mappings

For sets A and B, $\mathbb{M}(A, B)$ denotes the set of all finite mappings from A to B:

$$\mathbb{M}(A, B) = \{m : A \xrightarrow{\sim} B \mid \delta_0(m) \text{ is finite}\}.$$

Mappings can be defined by enumeration and by map comprehension. For $a_1, \ldots, a_n \in A$, where the a_i's are distinct, and $b_1, \ldots, b_n \in B$,

$$\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\},$$

named m, is the mapping in $\mathbb{M}(A, B)$ such that

$$\delta_0(m) = \{a_1, \ldots, a_n\} \wedge\ \forall i \in \{1, \ldots, n\} \cdot m(a_i) = b_i.$$

For $f : A \xrightarrow{\sim} B$, $A' \in \mathbb{F}(A)$, such that $A' \subseteq \delta_0(f)$ and P is a unary predicate on A,

$$m = \{a \mapsto f(a) \mid a \in A' \cdot P(a)\}$$

is the mapping in $\mathbb{M}(A, B)$ such that $m(a) = f(a)$ if $a \in A'$ and $P(a)$ is true and the set $\{a \mid a \in A' \cdot P(a)\}$ is finite. $m(a)$ is undefined otherwise. The empty map is denoted by $\{\mapsto\}$.

The following standard operators are also used on finite mappings:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$\underline{\text{dom}} : \mathbb{M}(A, B) \rightarrow \mathbb{F}(A)$
$\underline{\text{dom}}(m) = \delta_0(m)$

$\underline{\text{rng}} : \mathbb{M}(A, B) \rightarrow \mathbb{F}(B)$
$\underline{\text{rng}}(m) = \delta_1(m)$

$\underline{\text{restrict}} : \mathbb{M}(A, B) \times \mathbb{P}(A) \rightarrow \mathbb{M}(A, B)$
$\underline{\text{restrict}}(m, s) = \{a \mapsto m(a) \mid a \in (\underline{\text{dom}}(m) \cap s)\}$

$\underline{\text{subtract}} : \mathbb{M}(A, B) \times \mathbb{P}(A) \rightarrow \mathbb{M}(A, B)$
$\underline{\text{subtract}}(m, s) = \{a \mapsto m(a) \mid a \in (\underline{\text{dom}}(m) \backslash s)\}$

$\underline{\text{merge}} : \mathbb{M}(A, B) \times \mathbb{M}(A, B) \xrightarrow{\sim} \mathbb{M}(A, B)$
$\delta_0(\underline{\text{merge}}) = \{(m_1, m_2) \mid m_1, m_2 \in \mathbb{M}(A, B) \cdot \forall a \in (\underline{\text{dom}}(m_1) \cap \underline{\text{dom}}(\boldsymbol{m_2})) \cdot m_1(a) = m_2(a)\}$
$\underline{\text{merge}}(m_1, m_2) = m_1 \cup m_2$

$\underline{\text{Merge}} : \mathbb{F}(\mathbb{M}(A, B)) \xrightarrow{\sim} \mathbb{M}(A, B)$
$\delta_0(\underline{\text{Merge}}) = \{ms \mid ms \in \mathbb{F}(\mathbb{M}(A, B)) \cdot$
$\quad \forall m_1, m_2 \in ms \cdot \forall a \in \underline{\text{dom}}(m_1) \cap \underline{\text{dom}}(m_2) \cdot m_1(\boldsymbol{a}) = m_2(a)\}$
$\underline{\text{Merge}}(ms) = \bigcup ms$

$\underline{\text{overwrite}} : \mathbb{M}(A, B) \times \mathbb{M}(A, B) \rightarrow \mathbb{M}(A, B)$
$\underline{\text{overwrite}}(m_1, m_2) = \underline{\text{merge}}(\underline{\text{subtract}}(m_1, \underline{\text{dom}}(m_2)), m_2).$

A map abbreviation

Finally, the semantic definitions use $\mathbb{E}(A)$ as an abbreviation for $\mathbb{M}(Id', A)$ where Id' is the domain of all identifiers (i.e. *Id*, *OldId*, and *TypeVarId*).

5.7 Ordinal Numbers

Ordinal numbers constitute an extension into the infinite of the order properties of the natural numbers sequence:

$0 \leq 1 \leq 2 \leq \ldots$

A new number ω that comes after all of the natural numbers is affixed and the sequence continues counting from ω.

$0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots$

Because there is no reason to stop at any point, the sequence continues indefinitely as:

$0, 1, 2, \ldots, \omega, \omega + 1, \omega + 2, \ldots, \omega * 2, \omega * 2 + 1, \omega * 2 + 2, \ldots$

The main use of ordinal numbers in mathematics is in labelling sets to systematize constructions and proofs.

In set theory (End72) the natural numbers are represented as follows:

$$\begin{aligned} 0 &= \{\} \\ S(n) &= n \cup \{n\}, \end{aligned}$$

where S is a so-called *successor* operation.

The usual ordering on natural numbers can be defined by:

$n \leq m$ iff $n \in m$ or $n = m$

The *ordinal numbers*, are simply sets which can be ordered, similar to natural numbers using the ordering introduced above. To define them formally the auxiliary notion of $\in$-transitivity is introduced. A set S is called $\in$-transitive iff

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

for all x and y if $x \in y \in S$ then $x \in S$. A set α is called an *ordinal number*, or simply *ordinal*, if it is $\in$-transitive and all its elements are $\in$-transitive as well.

Natural numbers are a special case of ordinals.

For ordinals, as for natural numbers, the successor operation $S(x) = x \cup \{x\}$ will also be used. However, there is a difference between natural numbers and arbitrary ordinals. Every natural number (except 0) is of the form $S(n)$ for some n. This is no longer true in the case of arbitrary ordinal numbers. There are two kinds of (non-zero) ordinals — *successor ordinals* which are (as the non-zero natural numbers) of the form $S(\alpha)$ for some ordinal α, and *limit ordinals* which are not successor ordinals.

The simplest example of a limit ordinal is the set ω of all natural numbers.

5.8 Definition by Transfinite Induction

Functions with the set of naturals as a domain may be defined using induction. A typical example of an inductive definition is that of the factorial function:

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) * n! \end{aligned}$$

Here the factorial function !: $\omega \to \omega$ is defined using ! itself in the definition. It is a well-known fact that such definitions are valid under reasonable constraints.

Because ordinal numbers are a generalization of naturals there exists a generalization of the induction definition method. It is called *definition by transfinite induction*, or *transfinite recursion*. It can be formulated as follows.

Let α be a non-zero ordinal and A be a class. Suppose that a is an element of A, $F : A \to A$ is a function and $G : B \to A$ where $B = \{f \mid f : \beta \to A \text{ for some } \beta < \alpha\}$. Then there exists a unique function $H : \alpha \to A$ such that:

a) $H(0) = a$.

b) $H(S(\beta)) = F(H(\beta))$ for every β such that $S(\beta) < \alpha$.

c) $H(\beta) = G(H\lceil\beta)$ for every limit ordinal $\beta < \alpha$.

5.9 Cardinality and Cardinal Numbers

Two sets A and B are said to be *equipotent*, or *have the same cardinality*, if there exists a bijection $f : A \to B$.

To measure cardinality of sets the nature of "canonical sets having a given cardinality" is introduced. They are called *cardinal numbers*.

Cardinal numbers are special ordinals. Namely, an ordinal α is called a *cardinal number*, or simply a *cardinal* if it is not equipotent with any $\beta < \alpha$ (i.e. $\beta \leq \alpha$ and $\beta \neq \alpha$). It is easy to see that all natural numbers and ω are cardinals.

Assuming the so called axiom of choice, it can be formally proved that for every set S (regardless of whether or not it is finite) there exists a unique cardinal number $\underline{\text{card}}(S)$, called the *cardinality* of S, such that S and $\underline{\text{card}}(S)$ are equipotent. The axiom of choice simply says that for any family of non-empty sets one can pick an element from every set of that family and collect the elements in a resulting set.

A set S is said to be *finite* if $\underline{\text{card}}(S) < \omega$; *countable* if $\underline{\text{card}}(S) = \omega$; and *uncountable* if $\omega < \underline{\text{card}}(S)$.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.10 Structured Expressions

A number of structured constructs are used in the semantics definition.

The **if** expression has the usual meaning:

if T **then** e_1 **else** e_2 = e_1

if F **then** e_1 **else** e_2 = e_2.

When complicated tests are used with conjunctions or disjunctions it can be seen as a syntactic shorthand in the following way:

if $P \wedge Q$
then *cons*
else *altn*

it will correspond to:

if P
then if Q
  then *cons*
  else *altn*
else *altn*

For disjunction the corresponding explanation is:

if $P \vee Q$
then *cons*
else *altn*

it will correspond to:

if $\neg P$
then if $\neg Q$
  then *altn*
  else *cons*
else *cons*

In a few places it has been decided to use locally defined partial functions, and in these cases the else parts have been omitted. However, it is always ensured that the partial function is defined with a given argument before it is applied.

Another way of considering this is to think of a conditional and/or logic (originally presented in McC61) which are used in many modern programming languages.

The **let** construct

let $x = e$ **in** e'

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

can be read as a shorthand for:

$(\lambda x : X.e')(e)$.

The **cases** construct:

cases e :
$c_{11}, \ldots, c_{1n} \rightarrow e_1$
...
$c_{i1}, \ldots, c_{il} \rightarrow e_i$
...
$A_SET \rightarrow e_j$
...
$c_{m1}, \ldots, c_{mk} \rightarrow e_m$

where the c's are constant values, has the value e_i iff $e \in \{c_{i1}, \ldots, c_{il}\}$. In addition all c_{ij}'s must be distinct. In a few cases this construct is also used with a named set of tokens defined as: $A_SET = \{ c_1, c_2, \ldots, c_n \}$.

The **let** and **cases** constructs are also used in forms that facilitate the decomposition of tagged values (clause 8 contains the formal definitions for *MkTag*, *StripTag*, and *ShowTag* if they are required for the understanding of the following). The expression:

let $MkTag(t, x) = c$ **in** e'

corresponds to:

let $x = StripTag(e)$ **in** e'

if $ShowTag(e) = t$. It is undefined otherwise.

Similarly the expression:

cases e :
...
$MkTag(t, x) \rightarrow e'$
...

corresponds to:

cases $ShowTag(e)$:
...
$t \rightarrow$ **let** $x = StripTag(e)$ **in** e'
...

For notational convenience this kind of tagging is used for both syntactic and semantic entities. This is done in order to avoid translating the core abstract syntax (CAS) into the semantic domains (i.e. CAS $\subseteq$ VAL) as it was done for the STC/VDM-RL in Mon86. However, this trivial translation of the core abstract syntax to its semantical counterpart is not necessary, and does not improve the clarity of the definition. This means that all composite type definitions in the core abstract syntax are considered by the dynamic semantics as tagged values produced by means of *MkTag*. Thus, the value, *MkTag*(*t, x*), can correspond to a type definition in CAS like:

$t :: some_selector : x$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.11 Semantic Function and Predicate Definitions

The vast majority of the semantic functions are presented in the form:

5. *FunctionName* : *Type*
5.1 *FunctionName*(*arguments*) $\triangleq$
.2 *Function-definition*

Annotations to *FunctionName*:
5. Function-annotations
End of annotations

A few auxiliary functions are expressed implicitly by a number of axioms

FunctionName : *Type*
6. *FunctionName* must satisfy :
(1) *Axiom(s)*

Annotations to *FunctionName*:
6. Function-annotations
End of annotations

Meta-predicates (i.e. two-valued predicates from the level of the definition of the semantics) are presented like:

7. *PredicateName* : **Pred**(*Type*)
7.1 *PredicateName*(*arguments*) $\triangleq$
.2 *Predicate-definition*

In the above, *Predicate-definition* is composed from basic predicates (such as equality and set-membership) by means of connectives listed in clause 5.1.

Predicates in this part of ISO/IEC 13817 are not semantically considered as truth-valued functions. However (for example) a predicate P will be used as:

if $P(\ldots)$
then ...
else ...

instead of

if $P(\ldots) = True()$
then ...
else ...

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5.12 Use of Recursion

The dynamic semantics definition has a recursive nature, and therefore a justification about its soundness must be made. However, in this case recursion is only used over the abstract syntax tree which is finite. Thus, it can be proven by means of induction over the height of the tree that it is sound. Another way of explaining this, is to say that for any given abstract syntax the recursion can be entirely unfolded, leading to a non-recursive definition.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6 Core Abstract Syntax

The definition of dynamic semantics as specified in clause 8 is based on the core abstract syntax presented in this clause.

In this clause an annotated version of the core abstract syntax is given. The annotations firstly explain the abbreviations used, then the chosen abstraction, and finally (in some cases) very briefly highlight features of the semantics.

6.1 Document

$Document = Definitions$

Document is the start symbol in the core abstract syntax. It is simply a collection of 'definitions'.

6.2 Definitions

$$
\begin{array}{lll}
Definitions :: & typem & : ValueId \xrightarrow{m} TypeDef \\
& expofnm & : ValueId \mid PreId \mid PostId \xrightarrow{m} ExplPolyFnDef \\
& exmofnm & : ValueId \mid PreId \mid PostId \mid InitId \mid InvId \xrightarrow{m} ExplFnDef \\
& impofnm & : ValueId \xrightarrow{m} ImplPolyFnDef \\
& immofnm & : ValueId \xrightarrow{m} ImplFnDef \\
& valuem & : Pattern \xrightarrow{m} ValDef \\
& explopm & : ValueId \xrightarrow{m} ExplOpDef \\
& implopm & : ValueId \xrightarrow{m} ImplOpDef \\
& state & : [StateDef]
\end{array}
$$

Definitions consists of a number of collections of different 'definitions' and possibly a 'definition' of a state. The collections define types, *typem*; functions (polymorphic functions are defined either explicitly, *expofnm*, or implicitly, *impofnm*; and monomorphic functions are defined either explicitly, *exmofnm*, or implicitly, *immofnm*); values, *valuem*, and operations, *explopm* and *implopm*, respectively.

6.2.1 Type Definitions

$$
\begin{array}{lll}
TypeDef :: & shape & : Type \\
& inv & : InvId \times Lambda
\end{array}
$$

TypeDef is a 'type definition'. It consists of a type, *shape*, and an invariant, *inv*, on that type. The invariant has two components; an invariant identifier (e.g. *inv-A*), and a lambda expression representing the body of the invariant (e.g. $\lambda\, a : A \cdot \mathsf{true}$). This lambda expression should represent a truth-valued function from the defining type. If the type definition is recursive then the invariant need not be defined recursively over the structure. Semantically such recursive definitions are interpreted as the least fixed point and the invariant is taken into account in the iterative process of finding this least fixed point. The dynamic semantics disallows the invariant from being loosely specified.

$$
\begin{array}{l}
Type = BasicType \mid CompositeType \mid ProductType \mid UnionType \mid OptionalType \mid \\
\qquad SetType \mid SeqType \mid MapType \mid FnType \mid TypeId \mid TypeVar \mid QuoteType
\end{array}
$$

Type is either a basic type, a composite type, a union type, a set type, a sequence type, a mapping type, a function type, a type identifier, an optional type, a type variable, a product type, or a quotation type. Type variables can only be used inside a function type used in the definition of a polymorphic function.

$BasicType :: bt$: NATONE | NAT | INTEGER | RAT | REAL | BOOLEAN | CHAR | TOKEN | UNIT

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

BasicType is either a numeric type (the positive natural numbers, all natural numbers, integers, the rationals, and the reals), the Boolean type, characters, tokens, or a unit type (a singleton set containing NIL as its only element). All these basic types are treated as flat domains in the semantics (see clause 7.2.1).

CompositeType :: *id* : *ValueId*
fields : *Field**

CompositeType is a composite type, which is considered a record. It consists of a tag, *id*, and a sequence of types, *fields*, representing the fields of the record. It is semantically interpreted as smashed record space.

Field :: *sel* : [*ValueId*]
type : *Type*

Field possibly contains an identifier, *sel*, (which will be a selector function) and a type, *type*.

ProductType :: *fields* : *Type**

ProductType is a product type. It consists of a sequence of types, *fields*, representing the fields of the product. It is semantically interpreted as smashed Cartesian product.

UnionType :: *tps* : *Type*-set

UnionType is a union type. It consists of a set of types, *tps*. The union operator is interpreted as normal set-theoretic union.

OptionalType :: *tp* : *Type*

OptionalType is an optional type. It consists of a type, *tp*. It is understood as a shorthand for *Type* ∪ { NIL }.

SetType :: *elemtp* : *Type*

SetType is a set type. It consists of a type, *elemtp*. It is understood as the set of all *finite* subsets of the type, *Type*.

SeqType = *Seq*0*Type* | *Seq*1*Type*

SeqType is a sequence type. It is either a possibly empty sequence type or a non-empty sequence type.

*Seq*0*Type* :: *elemtp* : *Type*

*Seq*0*Type* is a possibly empty sequence type. It consists of a type, *elemtp*. It is understood as the set of all *finite* sequences with elements of the type, *Type*.

*Seq*1*Type* :: *elemtp* : *Type*

*Seq*1*Type* is a non-empty sequence type. It consists of a type, *elemtp*. It is understood as the set of all non-empty *finite* sequences with elements of type, *Type*.

MapType = *GeneralMapType* | *InjectiveMapType*

MapType is a map type. It is either a general map type[1)] or an injective map type.

GeneralMapType :: *dom* : *Type*
rng : *Type*

[1)] The term *general* is used here in contrast to *injective* in the sense that a general map type cannot always be inverted.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

GeneralMapType is a general map type. It consists of a domain type, *dom* and a range type, *rng*. It is understood as the set of all *finite* mappings from the *dom* type to the *rng* type.

InjectiveMapType :: *dom* : *Type*
rng : *Type*

InjectiveMapType is an injective map type. It consists of a domain type, *dom*, and a range type, *rng*. It is understood as the set of all *finite injective* mappings from the *dom* type to the *rng* type.

FnType :: *dom* : *Type*
rng : *Type*

FnType is a function type. It consists of a domain type, *dom*, and a range type, *rng*. It is understood as the set of all partial functions from the *dom* type to the *rng* type.

TypeId :: *id* : *ValueId*

TypeId is a type identifier. It consists of an identifier, *id*. The identifier used here must be defined in another type definition.

TypeVar :: *var* : *TypeVarId*

TypeVar is a type variable. It consists of an identifier, *var*. It is a type variable which can only be used in polymorphic function definitions.

QuoteType :: *lit* : *QuoteLit*

QuoteType is a quotation type. It consists of a quote literal, *lit*. It is semantically interpreted as a singleton set of this quote literal (i.e. a token).

6.2.2 State Definition

StateDef :: *stid* : *ValueId*
tp : *ValueId*
init : *ValueId* × *Expr*

StateDef is a state definition. It consists of the name of the state, *stid*, the name of the type of the state, *tp*, and a truth-valued function which defines the possible initial states *init*.

6.2.3 Value Definitions

ValDef :: *type* : [*Type*]
val : *Expr*

ValDef is a 'value definition'. It consists of an expression, *val*, and possibly a type, *type*. The expression designates a value while the type restricts the value.

6.2.4 Function Definitions

ExplPolyFnDef :: *tpparms* : *TypeVar*$^+$
polytp : *Type*
pre : *Expr*
body : *Lambda*
kind : TOTAL | PARTIAL

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ExplPolyFnDef is an 'explicit polymorphic function definition'. It consists of a non-empty sequence of type variables, *tpparms*, a type expression (using these type variables), *polytp*, a pre-condition restricting the domain of definition, *pre*, a lambda expression representing the body, *body*, and information about whether the defined function should be total or partial with respect to the pre-condition, *kind*.

ExplFnDef :: *domtp* : *Type*
restp : *Type*
pre : *Expr*
body : *Lambda*
kind : TOTAL | PARTIAL

ExplFnDef is an 'explicit monomorphic function definition'. It consists of a domain type, *domtp*, a result type, *restp*, a pre-condition restricting the domain of definition, *pre*, a lambda expression representing the body of the function, *body*, and information about whether the defined function should be total or partial with respect to the pre-condition, *kind*. Note that if the function is Curried the *restp* type will be a function type.

ImplPolyFnDef :: *tpparms* : *TypeVar*$^+$
heading : *ImplFnHeading*
pre : *Expr*
post : *Expr*

ImplPolyFnDef is an 'implicit polymorphic function definition'. It consists of a non-empty sequence of type variables, *tpparms*, an implicit function heading, *heading*, a pre-condition, *pre*, and a post-condition, *post*.

ImplFnDef :: *heading* : *ImplFnHeading*
pre : *Expr*
post : *Expr*

ImplFnDef is an 'implicit monomorphic function definition'. It consists of an implicit function heading, *heading*, and two expressions representing the pre-condition, *pre*, and the post-condition, *post*.

ImplFnHeading :: *par* : *Par*
restp : *Type*
resum : *ValueId*

ImplFnHeading is an 'implicit function heading'. It consists of a parameter (an input identifier and its corresponding type), *par*, a result type, *restp*, and a value identifier used to designate the result of evaluating the function, *resum*.

Par :: *id* : *ValueId*
type : *Type*

Par is a 'parameter'. It consists of an identifier, *id*, and its corresponding type, *type*.

Observe that implicitly defined functions cannot be Curried.

6.2.5 Operation Definitions

Observe that:

a) Operations cannot be Curried but they can take functions as arguments and return functions as result.

b) Operations cannot be polymorphic.

ExplOpDef :: *heading* : *OpHeading*
body : *Stmt*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ExplOpDef is an 'explicit operation definition'. It consists of an operation heading, *heading*, and a statement which represents the body of the operation, *body*.

ImplOpDef :: *heading* : *OpHeading*
body : *OpPost*

ImplOpDef is an 'implicit operation definition'. It consists of an operation heading, *heading* and a post-condition body, *body*.

OpHeading :: *par* : *Par*
restp : [*Type*]
pre : *Expr*
ext : *ExtVarInf*-set

OpHeading is an 'operation heading'. It consists of a parameter (an input identifier and its corresponding type), *par*, a result type, *restp*, a pre-condition restricting the domain of definition, *pre*, and a set of externals (parts of the state which are used by the operation), *ext*.

ExtVarInf :: *mode* : *ReadWriteMode*
rest : *ValueId*
tp : *Type*

ExtVarInf is an 'external variable information'. It consists of a mode of the external, *mode*, the name of it, *rest*, and the type of it, *tp*.

ReadWriteMode = READ | READWRITE

ReadWriteMode describes whether a part of the state (an external) is only read or whether it is also written to.

OpPost :: *return* : [*ValueId*]
post : *Expr*

OpPost is the body of an 'implicitly defined operation'. It consists of an optional identifier denoting the result of the operation, *return*, and an expression representing the post-condition, *post*.

6.3 Expressions

Expr = *LetExpr* | *LetBeSTExpr* | *DefExpr* |
IfExpr | *CasesExpr* |
UnaryExpr | *BinaryExpr* |
QuantExpr | *IotaExpr* |
SetEnumeration | *SetComprehension* | *SetRange* |
SeqEnumeration | *SeqComprehension* | *SubSequence* |
MapEnumeration | *MapComprehension* | *TupleConstructor* |
RecordConstructor | *RecordModifier* | *Apply* |
FieldSelect | *FctTypeInst* | *Lambda* |
IsExpr | *Literal* | *Id* | *OldId*

Expr is an expression.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.3.1 Local Binding Expressions

$$\begin{aligned} \mathit{LetExpr} :: \mathit{vals} &: \mathit{Pattern} \xrightarrow{m} \mathit{ValDef} \\ \mathit{explfns} &: \mathit{ValueId} \xrightarrow{m} \mathit{ExplFnDef} \\ \mathit{implfns} &: \mathit{ValueId} \xrightarrow{m} \mathit{ImplFnDef} \\ \mathit{in} &: \mathit{Expr} \end{aligned}$$

LetExpr is a let expression. It consists of a map of 'value definitions', *vals*, a map of 'explicit (monomorphic) function definitions, *explfns*, a map of 'implicit (monomorphic) function definitions', *implfns*, and an expression in which these 'definitions' must be visible, *in*. The 'definitions' in *defs*, *explfns*, and *implfns* can be mutually recursive.

$$\begin{aligned} \mathit{LetBeSTExpr} :: \mathit{bind} &: \mathit{Bind} \\ \mathit{st} &: \mathit{Expr} \\ \mathit{in} &: \mathit{Expr} \end{aligned}$$

LetBeSTExpr is a 'let-be-such-that expression'. It consists of a binding, *bind*, an expression (restricting the possibly bindings further), *st*, and an expression in which the bindings of the pattern identifiers from *bind* must be visible, *in*.

$$\begin{aligned} \mathit{DefExpr} :: \mathit{lhs} &: \mathit{Pattern} \\ \mathit{rhs} &: \mathit{Expr} \\ \mathit{in} &: \mathit{Expr} \end{aligned}$$

DefExpr is a 'local definition expression'. It consists of a left hand side pattern, *lhs*, a right hand side expression (possibly reading the state), *rhs*, and an expression in which these bindings must be visible, *in*.

6.3.2 Conditional Expressions

$$\begin{aligned} \mathit{IfExpr} :: \mathit{test} &: \mathit{Expr} \\ \mathit{cons} &: \mathit{Expr} \\ \mathit{altn} &: \mathit{Expr} \end{aligned}$$

IfExpr is an 'if-then-else expression'. It consists of a test expression, *test*, a consequence expression, *cons*, and an alternative expression, *altn*.

$$\begin{aligned} \mathit{CasesExpr} :: \mathit{sel} &: \mathit{Expr} \\ \mathit{altns} &: \mathit{CaseAltn}^{+} \end{aligned}$$

CasesExpr is a 'cases expression'. It consists of a selector expression, *sel*, and a non-empty sequence of case alternatives, *altns*.

$$\begin{aligned} \mathit{CaseAltn} :: \mathit{match} &: \mathit{Pattern} \\ \mathit{body} &: \mathit{Expr} \end{aligned}$$

CaseAltn is a 'case expression alternative'. It consists of a pattern, *match*, and an expression, *body*. The OTHERWISE alternative from the concrete syntax is represented by a "don't care" pattern.

6.3.3 Unary Expressions

$$\begin{aligned} \mathit{UnaryExpr} :: \mathit{opr} &: \mathit{UnaryOp} \\ \mathit{arg} &: \mathit{Expr} \end{aligned}$$

UnaryExpr is an 'unary expression'. It consists of an unary operator, *opr*, and an expression, *arg*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

UnaryOp = NUMPLUS | NUMMINUS | NUMABS | FLOOR |
NOT |
SETCARD | SETPOWER | SETDISTRUNION | SETDISTRINTERSECT |
SEQHEAD | SEQTAIL | SEQLEN | SEQELEMS | SEQINDICES | SEQDISTRCONC |
MAPDOM | MAPRNG | MAPINVERSE | MAPDISTRMERGE

UnaryOp is an 'unary operator'.

6.3.4 Binary Expressions

BinaryExpr :: *left* : *Expr*
opr : *BinaryOp*
right : *Expr*

BinaryExpr is a 'binary expression'. It consists of a left hand side expression, *left*, a binary operator, *opr*, and a right hand side expression, *right*.

BinaryOp = NUMPLUS | NUMMINUS | NUMMULT | NUMDIV | INTDIV | NUMREM | NUMMOD |
NUMLT | NUMLE | NUMGT | NUMGE | EQ | NE |
OR | AND | IMPLY | EQUIV |
INSET | NOTINSET | SUBSET | PROPERSUBSET |
SETUNION | SETINTERSECT | SETDIFFERENCE |
SEQCONC | MAPORSEQMOD | MAPMERGE |
MAPDOMRESTRTO | MAPDOMRESTRBY | MAPRNGRESTRTO | MAPRNGRESTRBY |
COMPOSE | ITERATE

BinaryOp is a 'binary operator'.

6.3.5 Quantified Expressions

QuantExpr = *AllOrExistsExpr* | *ExistsUniqueExpr*

QuantExpr is a quantified expression. It is either a universal quantified expression or an existential quantified expression or it is a unique quantified expression.

AllOrExistsExpr :: *quant* : *Quantifier*
bind : *Bind*-set
pred : *Expr*

AllOrExistsExpr is either a universal quantified expression or an existential quantified expression. It consists of a quantifier, *quant*, (i.e., 'for all' or 'exists'), a set of bindings, *bind*, and an expression which is the predicate which the quantification covers, *pred*.

Quantifier = ALL | EXISTS

Quantifier is a 'quantifier'. It is either a 'for all quantifier' or an 'exists quantifier'.

ExistsUniqueExpr :: *bind* : *Bind*
pred : *Expr*

ExistsUniqueExpr is a unique quantified expression. It consists of a binding, *bind*, and an expression which is the predicate which thc quantification covers, *pred*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.3.6 Iota Expression

IotaExpr :: *bind* : *Bind*
pred : *Expr*

IotaExpr is an 'iota expression'. It consists of a binding, *bind*, and a predicate expression, *pred*. Semantically it corresponds to selecting the unique element which fulfils an expression like $\exists!bind \cdot pred$.

6.3.7 Set Expressions

SetEnumeration :: *els* : *Expr**

SetEnumeration is a set enumeration expression which simply consists of a sequence[1)] of element expressions, *els*.

SetComprehension :: *elem* : *Expr*
binds : *Bind*-set
pred : *Expr*

SetComprehension is a 'set comprehension expression'. It consists of an element expression, *elem*, a set of bindings, *bind*, and a predicate expression, *pred*. For any possible combination of the bindings which additionally satisfies the predicate, *pred*, the element expression, *elem*, is evaluated.

SetRange :: *lb* : *Expr*
ub : *Expr*

SetRange is a 'set range expression'. It consists of a lower bound expression, *lb*, and an upper bound expression, *ub*. Both these expressions should evaluate to numeric values, and if so this construct will denote the set of integers between and including the lower and upper bound.

6.3.8 Sequence Expressions

SeqEnumeration :: *els* : *Expr**

SeqEnumeration is a 'sequence enumeration expression'. It is simply a sequence of element expressions, *els*.

SeqComprehension :: *elem* : *Expr*
bind : *Bind*
pred : *Expr*

SeqComprehension is a 'sequence comprehension expression'. It consists of an element expression, *elem*, a binding (which must be of a single identifier), *bind*, and a predicate expression, *pred*. For any possible binding which additionally satisfies the predicate, *pred*, the element expression, *elem*, is evaluated and ordered according to the binding value which may, therefore, be only numeric.

SubSequence :: *sequence* : *Expr*
from : *Expr*
to : *Expr*

SubSequence is a 'sub-sequence expression'. It consists of a sequence expression, *sequence*, a lower bound expression, *from*, and an upper bound expression, *to*. Note that the lower and upper bound must be numeric values.

[1)]The order of these expressions is not important for the semantics, but it cannot be represented as a set because a loosely specified expression can occur more than once (and evaluate to different results).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.3.9 Map Expressions

MapEnumeration :: *els* : *Maplet**

MapEnumeration is a 'map enumeration expression'. It consists of a set of 'maplets' (pairs of domain element and range element), *els*.

Maplet :: *dom* : *Expr*
rng : *Expr*

Maplet is a 'maplet'. It is a pair of a domain element, *dom*, and a range element, *rng* (which are used in a map).

MapComprehension :: *elem* : *Maplet*
bind : *Bind*-set
pred : *Expr*

MapComprehension is a 'map comprehension expression'. It consists of an element 'maplet', *elem*, a set of bindings, *bind*, and a predicate expression, *pred*. The expression denotes a map which for any possible combination of the bindings which satisfies the predicate, *pred*, the element maplet, *elem*, is evaluated.

6.3.10 Tuple Constructor

TupleConstructor :: *fields* : *Expr*$^+$

TupleConstructor is a 'tuple constructor expression'. It consists of a sequence of expressions representing the fields of the tuple.

6.3.11 Record Expressions

RecordConstructor :: *tag* : *MkId*
fields : *Expr**

RecordConstructor is a 'record constructor expression'. It consists of an identifier, *tag*, and a sequence of expressions representing the fields of the record, *fields*.

RecordModifier :: *rec* : *Expr*
modifiers : *ValueId* $\xrightarrow{m}$ *Expr*

RecordModifier is a 'record modifier expression'. It consists of a 'record expression', *rec*, and a map of modifiers (from selector identifiers to element expressions), *modifiers*[1)]. The expression takes a record, *rec*, and modifies those fields that are present in the domain of *modifiers* with the corresponding element expressions.

6.3.12 Apply Expressions

Apply :: *fct* : *Expr*
arg : *Expr*

Apply is an 'application expression'. It consists of an expression which is either a (monomorphic) function, a map, or a sequence, *fct*, and an expression representing the argument which must be applied, *arg*.

FieldSelect :: *record* : *Expr*
field : *ValueId*

[1)] In Jon86 these are called μ-functions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

FieldSelect is a 'field select expression'. It consists of a record expression, *record*, and a selector field identifier, *field*.

$FctTypeInst :: id : Id$
$\quad tpinst : Type^+$

FctTypeInst is an expression where a polymorphic function is instantiated. It consists of a polymorphic function identifier, *id*, and a non-empty sequence of types (to instantiate the polymorphic function's type variables), *tpinst*.

6.3.13 Lambda Expression

$Lambda :: parm : Par$
$\quad body : Expr$

Lambda is a 'monomorphic lambda expression'. It consists of a parameter (an identifier and its type), *parm*, and an expression, *body*.

6.3.14 Is Expression

$IsExpr :: tag : ValueId \mid BasicType$
$\quad arg : Expr$

IsExpr is an 'is-expression'. It consists of either a type tag or a basic type, *type*, and an argument expression, *arg*. It denotes a Boolean expression which tests whether or not *arg* is of the type, *type*.

6.3.15 Literals

$Literal = NumLit \mid BoolLit \mid CharLit \mid TextLit \mid QuoteLit \mid NilLit$

Literal is a literal. It is either a numeric literal, a Boolean literal, a character literal, a text literal, a quote literal, or a nil literal.

$NumLit :: val : \mathbb{R}$

NumLit is a 'numeric literal'. It is any numeric value.

$BoolLit :: val : \mathbb{B}$

BoolLit is a 'Boolean literal'. It is either TRUE or FALSE.

$CharLit :: val :$ CHAR

CharLit is a 'character literal'. It is a character.

$TextLit :: val : CharLit^*$

TextLit is a 'text literal'. It is a sequence of 'character literals'.

$QuoteLit :: val : CharLit^*$

QuoteLit is a 'quote literal'. It is represented as a token.

$NilLit :: val :$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

NilLit is a 'nil literal'. It is simply NIL. Note that the basic type UNIT is also equal to {NIL}.

6.3.16 Identifiers

The identifiers are partitioned into several non-overlapping, disjoint parts by means of tagging. This is also done in the Outer Abstract Syntax (OAS), but a few extra name spaces used internally by the dynamic semantics are also included below.

Id = *ValueId* | *PreId* | *PostId* | *InvId* | *InitId* | *MkId* | *IsId* |
StateTypeId | *StateConstId* | *RecSelId* | *RecModId* | *RecConsId* | *TypeEnvId* | *TagEnvId*

Id is an identifier which is represented as a tagged token.

ValueId :: token

PreId :: token

PostId :: token

InvId :: token

InitId :: token

MkId :: token

IsId :: token

StateTypeId :: token

StateConstId ::

RecSelId :: token × token

RecModId :: token

RecConsId :: token

TypeEnvId ::

TagEnvId ::

TypeVarId :: token

OldId :: token

OldId is an identifier which has been 'hooked' in a post-condition of an operation (the value of a state component before the execution of the operation).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.4 State Designators

StateDesignator = *ValueId* | *FieldRef* | *MapOrSeqRef*

StateDesignator is a 'state designator'. It is either an identifier, a 'field reference' or a reference to either a map or a sequence. This component is used referring to a specific part of the state.

FieldRef :: *sd* : *StateDesignator*
sel : *ValueId*

FieldRef is a 'field reference' to part of a state. It consists of a state designator, *sd*, and a selector identifier, *sel*. The state designator, *sd*, must designate a record value, and the selector identifier, *sel*, must designate a selector from that record.

MapOrSeqRef :: *sd* : *StateDesignator*
sel : *Expr*

MapOrSeqRef is a reference to a part of a state which is either a map or a sequence. It consists of a state designator, *sd*, and a selector expression *sel*. The state designator, *sd*, must designate either a map value or a sequence value, and the selector expression, *sel*, must designate either an index in the sequence or an element in the domain of the map.

6.5 Statements

Stmt = *LetStmt* | *LetBeSTStmt* | *DefStmt* |
Block | *Sequence* | *Assign* | *IfStmt* | *CasesStmt* |
SeqForLoop | *SetForLoop* | *IndexForLoop* | *WhileLoop* |
NonDetStmt | *Call* | *ReturnStmt* |
HandlerStmt | *Exit* |
IDENT

Stmt is a 'statement'. It is either one of the ones defined below or an identity statement.

6.5.1 Local Binding Statements

LetStmt :: *defs* : *Pattern* $\xrightarrow{m}$ *ValDef*
explfns : *ValueId* $\xrightarrow{m}$ *ExplFnDef*
implfns : *ValueId* $\xrightarrow{m}$ *ImplFnDef*
in : *Stmt*

LetStmt is a 'let statement'. It consists of a map of 'value definitions', *defs*, a map of 'explicit (monomorphic) function definitions', *explfns*, a map of 'implicit (monomorphic) function definitions', *implfns*, and a statement in which these 'definitions' must be visible, *in*. The 'definitions' in *defs*, *explfns*, and *implfns* can be mutually recursive.

LetBeSTStmt :: *bind* : *Bind*
st : *Expr*
in : *Stmt*

LetBeSTStmt is a 'let-be-such-that statement'. It consists of a binding, *bind*, an expression (possibly restricting the bindings further), *st*, and a statement in which the binding of the pattern identifiers from *bind* must be visible, *in*.

DefStmt :: *lhs* : *Pattern*
rhs : *Expr* | *Call*
in : *Stmt*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

DefStmt is a 'local definition statement'. It consists of a left hand side pattern, *lhs*, a right hand side expression (possibly reading the state) or call of a value returning operation, *rhs*, and a statement in which a binding of the pattern identifiers from the *lhs* pattern (matched against the value of the *rhs*) must be visible, *in*.

6.5.2 Block and Assignment Statements

Block :: *var* : *ValueId*
init : [*Expr* | *Call*]
tp : *Type*
body : *Stmt*

Block is a 'block statement'. It consists of a local variable identifier, *var*, possibly an initial value of it (either by an expression or by calling a value returning operation), *init*, a type to which its value must belong to, *tp*, and a statement which represents the body of the block, *body*. It is here that a local variable (considered as a part of the state) can be declared.

Sequence :: *stmts* : *Stmt**

Sequence is a 'sequence statement' It consists of a sequence of statements, *stmts*.

Assign :: *lhs* : *StateDesignator*
rhs : *Expr* | *Call*

Assign is an 'assign statement'. It consists of a left hand side state designator (a reference to a part of the state), *lhs*, and a right hand side expression or call of a value-returning operation, *rhs*.

6.5.3 Conditional Statements

IfStmt :: *test* : *Expr*
cons : *Stmt*
altn : *Stmt*

IfStmt is an 'if-then-else statement'. It consists of a test expression, *test*, a consequence statement, *cons*, and an alternative statement, *altn*.

CasesStmt :: *sel* : *Expr*
altns : *CaseStmtAltn*$^+$

CasesStmt is a 'cases statement'. It consists of a selector expression, *sel*, and a non-empty sequence of case alternatives, *altns*.

CaseStmtAltn :: *match* : *Pattern*
body : *Stmt*

CaseStmtAltn is a 'case statement alternative'. It consists of a pattern, *match*, (which is matched against the selector value), and a statement which represents the body of the alternative, *body*. The OTHERWISE alternative is represented by a "don't care" pattern.

6.5.4 Loop Statements

SeqForLoop :: *cv* : *Pattern*
dirn : *LoopDirection*
seq : *Expr*
body : *Stmt*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SeqForLoop is a 'for-loop statement over a sequence value'. It consists of a pattern (to be matched against each element in the sequence), *cv*, information about the loop direction, *dirn*, an expression (denoting a sequence), *seq*, and a statement which represents the body of the for-loop, *body*. The sequence value is static in the sense that it is evaluated only once, and not on each turn of the loop.

LoopDirection = FORWARDS | BACKWARDS

SetForLoop :: *cv* : *Pattern*
set : *Expr*
body : *Stmt*

SetForLoop is a 'for-loop statement over a set value'. It consists of a pattern (to be matched against each element in the set), *cv*, an expression (denoting a set), *set*, and a statement which represents the body of the for-loop, *body*. The set value is static in the sense that it is evaluated only once, and not on each turn of the loop.

IndexForLoop :: *cv* : *ValueId*
lb : *Expr*
ub : *Expr*
by : *Expr*
body : *Stmt*

IndexForLoop is an integer increment/decrement 'for-loop statement'. It consists of an identifier, *cv*, a lower bound expression, *lb*, an upper bound expression, *ub*, a by expression (indicating the step size), *by*, and a statement representing the body of the for-loop, *body*. Both the lower- and upper bounds and the step size are static in the sense that they are evaluated only once and not on each turn of the loop.

WhileLoop :: *test* : *Expr*
body : *Stmt*

WhileLoop is a 'while loop statement'. It consists of a test expression, *test*, and a statement, which represents the body of the loop, *body*.

6.5.5 Non-Deterministic Sequences

NonDetStmt :: *stmts* : $Stmt^+$

NonDetStmt is a 'non-deterministic statement'. It consists of a non-empty sequence of statements[1)], *stmts*. It denotes an arbitrary ordering of execution of these statements.

6.5.6 Call and Return Statements

Call :: *oprt* : *ValueId*
arg : *Expr*
state : [*StateDesignator*]

Call is a statement calling an operation. It consists of an identifier (the name of the operation), *oprt*, an argument expression, *arg*, and an optional state designator indicating which part of the state it is acting upon, *state* (it will only be nilif the entire specification has no state component).

ReturnStmt :: *expr* : [*Expr*]

ReturnStmt is a 'return statement'. It consists of an expression, *expr*, whose value must be returned.

[1)]The order of these statements is not important for the semantics, but it cannot be represented as a set because a statement can occur more than once.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.5.7 Exception Handling Statements

HandlerStmt = *Always* | *TrapStmt* | *RecTrapStmt*

HandlerStmt is a 'handler statement'. It takes care of trapping exits in three different ways.

Always :: *post* : *Stmt*
body : *Stmt*

Always is an 'always statement'. It consists of a post statement, *post* and a body statement, *body*. The *post* part is always evaluated after the evaluation of the *body*, even if it terminated with an exit.

TrapStmt :: *pat* : *Pattern*
post : *Stmt*
body : *Stmt*

TrapStmt is a 'trap statement'. It consists of a pattern, *pat*, (to be matched against the exit value), a post statement, *post*, and a body statement, *body*. The *post* part will be evaluated only if the *body* statement terminates with an exit, and the pattern matches the exit value. Then the evaluation of *post* will take place in an environment which has been enriched by the bindings from the pattern matching.

RecTrapStmt :: *traps* : $Pattern \xrightarrow{m} Stmt$
body : *Stmt*

RecTrapStmt is a 'recursive trap statement'. It consists of a mapping of 'traps', *traps*, and a body statement, *body*. If the *body* statement terminates with an error, and the exit value matches any of the patterns in the domain of the map of traps, then the corresponding trap statement (in the map) will be evaluated. Exits from this evaluation will then be trapped recursively.

Exit :: *expr* : [*Expr*]

Exit is an exit statement. It consists of an optional expression, *expr*, whose value must be returned.

6.6 Patterns and Bindings

Pattern = *PatternId* | *MatchVal* | *SeqPattern* | *TuplePattern* |
RecordPattern | *SetPattern* | *ConstrPattern*

Pattern is a pattern. It is either a 'pattern identifier', a 'matching value', a 'sequence pattern', a 'tuple pattern', a 'record pattern', a 'set pattern', or a 'constrained pattern'.

PatternId :: *id* : [*ValueId*]

PatternId is a 'pattern identifier'. It is either simply an identifier or a "don't-care".

MatchVal :: *val* : *Expr*

MatchVal is a 'matching value' (used inside a pattern). It is simply an expression.

SeqPattern = *SeqEnumPattern* | *SeqConcPattern*

SeqPattern is a 'sequence pattern'. It is either a 'sequence enumeration pattern' or a 'sequence concatenation pattern'.

SeqEnumPattern :: *els* : $Pattern^*$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SeqEnumPattern is a 'sequence enumeration pattern'. It consists of a sequence of patterns, *els*.

SeqConcPattern :: *lp* : *Pattern*
rp : *Pattern*

SeqConcPattern is a 'sequence concatenation pattern'. It consists of a left pattern, *lp*, and a right pattern, *rp*. These two patterns must be concatenated together (corresponding to a binary expression with the sequence concatenation operator).

TuplePattern :: *fields* : $Pattern^+$

TuplePattern is a 'tuple pattern' It consists of a sequence of patterns, *fields* (corresponding to a tuple constructor).

RecordPattern :: *tag* : *MkId*
fields : $Pattern^*$

RecordPattern is a 'record pattern'. It consists of an identifier, *tag*, and a sequence of patterns acting as fields of the record, *fields*, (corresponding to a record constructor expression).

SetPattern = *SetEnumPattern* | *SetUnionPattern*

SetPattern is a 'set pattern'. It is either a 'set enumeration pattern' or a 'set union pattern'.

SetEnumPattern :: *els* : $Pattern^+$

SetEnumPattern is a 'set enumeration pattern'. It consist of a sequence of patterns, *els*, (corresponding to a set enumeration expression).

SetUnionPattern :: *lp* : *Pattern*
rp : *Pattern*

SetUnionPattern is a 'set union pattern'. It consists of two patterns, *lp*, and *rp* (corresponding to a binary expression with a set union operator).

ConstrPattern = *SetConstrPattern* | *TypeConstrPattern*

ConstrPattern is a 'constrained pattern'. It is either a 'set constrained pattern' or a 'type constrained pattern'.

SetConstrPattern :: *pat* : *Pattern*
set : *Expr*

SetConstrPattern a 'set constrained pattern'. It consists of a pattern, *pat*, and a set expression, *set*.

TypeConstrPattern :: *pat* : *Pattern*
type : *Type*

TypeConstrPattern is a 'type constrained pattern'. It consists of a pattern, *pat*, and a type, *type*.

Bind = *SetBind* | *TypeBind*

Bind is a 'binding'. It is either a 'set binding' or a 'type binding'.

SetBind :: *pat* : *Pattern*
set : *Expr*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SetBind is a 'set binding'. It consists of a pattern, *pat*, and an expression which must denote a set, *set*. Semantically a 'set bind' will denote the collection of all possible value environments resulting from matching the pattern against all elements of the set.

TypeBind :: *pat* : *Pattern*
type : *Type*

TypeBind is a type binding. It consists of a pattern, *pat*, and a type, *type*. Semantically a 'type bind' will denote the collection of all possible value environments resulting from matching the pattern against all elements belonging to the type.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7 Dynamic Semantic Domains

This clause first introduces the domain universe and then defines domain operators — operators which can be used in the definition of the dynamic semantics[1)]. A number of semantic domains which will be used in the definition of the dynamic semantics are then defined. These are all based on the domain universe.

7.1 The Domain Universe

The domain universe which is introduced in this clause provides denotations for all values expressible in VDM-SL. This clause will first introduce the basic theory behind cpo's, and then a number of cpo operators will be presented. Then a universe of cpo's is built using transfinite induction and the defined cpo operators. In order to be able to interpret invariants semantically a universe of domains is finally constructed. This last clause also defines new domain operators[2)] — operators which will be used in the definition of the dynamic semantics.

7.1.1 Basic Definitions

A standard way of formally interpreting recursive definitions of elements of a domain is to equip the domain with a partial ordering, and then define an element to be the least fixed point of the function involved in the recursive definition. This works in a particularly smooth way if the domain forms a chain-complete partial order, and the function involved is continuous. This clause reviews the related basic concepts which underlie and motivate the construction of the VDM-SL domain universe.

7.1.1.1 Complete Partial Orders and Fixed Point Definitions

Definition 7.1 (Partial Ordering) A binary relation $\sqsubseteq$ on D is called a *partial ordering* on D iff it is:

a) Reflexive: for all $a \in D$, $a \sqsubseteq a$.

b) Antisymmetric: for all $a, b \in D$, $a \sqsubseteq b$ and $b \sqsubseteq a$ imply $a = b$.

c) Transitive: for all $a, b, c \in D$, $a \sqsubseteq b$ and $b \sqsubseteq c$ imply $a \sqsubseteq c$.

Definition 7.2 (Partially ordered set) A *partially ordered set* A is a pair $(|A|, \sqsubseteq_A)$ where $|A|$ is a set and $\sqsubseteq_A$ is a partial ordering on $|A|$.

Let $A = (|A|, \sqsubseteq_A)$ be a partially ordered set in the following definitions.

Definition 7.3 (Upper bound) An *upper bound* of a set $X \subseteq |A|$ is an element $a \in |A|$ such that for all $x \in X$, $x \sqsubseteq_A a$.

Definition 7.4 (Least upper bound) A *least upper bound* of a set $X \subseteq |A|$ in A is an element $\bigsqcup_A X \in |A|$ such that $\bigsqcup_A X$ is an upper bound of X, and for any upper bound a of X, $\bigsqcup_A X \sqsubseteq_A a$.

Definition 7.5 (Least element (bottom)) An element $\bot_A = \bigsqcup_A \emptyset$, if it exists, is called the *least element* of A.

Definition 7.6 (Countable chain) A sequence $(a_i \mid i \in \omega)$ of elements of $|A|$ is called a *countable chain* in A if it is increasing (i.e. $a_0 \sqsubseteq_A a_1 \sqsubseteq_A \ldots$).

Definition 7.7 (Complete partial order) A is called a *complete partial order*, or a *cpo*, if it has a least element and for each countable chain of its elements there exists a least upper bound of the chain in A.

1) The domain universe construction presented here is inspired by TW90.

2) These operators are simple extensions of the operators at the cpo level.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Definition 7.8 (Continuous function) Given two cpo's A and B, a function : $f: |A| \rightarrow |B|$ is *continuous* if it preserves the least upper bounds of all countable chains in A, i.e. for any chain $(a_i \mid i \in \omega)$:

$$f(\bigsqcup_A \{a_i \mid i \in \omega\}) = \bigsqcup_B \{f(a_i) \mid i \in \omega\}.$$

Proposition 7.9 For any cpo A and continuous function $f: |A| \rightarrow |A|$, the least upper bound:

$$\bigsqcup_A (f^n(\bot_A) \mid n \in \omega)$$

of the chain $(f^n(\bot_A) \mid n \in \omega)$, where:

— $f^0(\bot_A) = \bot_A$, and

— $f^{n+1}(\bot_A) = f(f^n(\bot_A))$ for $n \geq 0$,

is the least fixed point of the function f, i.e. the least solution to the recursive equation

$$x = f(x).$$

The least fixed point of f is written as $\mathsf{Y}f$ where Y is a least fixed point operator yielding the least solution.

7.1.1.2 Operators on Complete Partial Orders

A number of operators on cpo's which will be used in the construction of the domain universe are given in the following definitions.

Basic cpo's are formed by *lifting* sets of elements.

Definition 7.10 (Lifting) For any set S, the result of *lifting* S is a cpo $S_\bot$ defined by:

— $|S_\bot| = S \cup \{\bot\}$

— for $a_1, a_2 \in |S_\bot|$, $a_1 \sqsubseteq_{S_\bot} a_2$ iff $a_1 = \bot$ or $a_1 = a_2$.

A cpo with an ordering defined as above is called a *flat* cpo.

In general, the set-theoretic union of two cpo's A and B, defined to be a partially ordered set:

$$(|A| \cup |B|, \sqsubseteq_A \cup \sqsubseteq_B)$$

does not have to be a cpo itself. Therefore the notion of *union-compatibility* of cpo's is introduced. The more general case is given here because it will be needed in the construction of the cpo universe.

Definition 7.11 (Union-compatible cpo's) Let $\mathcal{A}$ be a family of cpo's. The family $\mathcal{A}$ is *union-compatible* if:

$$\bigcup \mathcal{A} = (\bigcup \{|A| \mid A \in \mathcal{A}\}, \bigcup \{\sqsubseteq_A \mid A \in \mathcal{A}\})$$

is a cpo.

Definition 7.12 (Union) Let $\mathcal{A}$ be a union-compatible family of cpo's with for all A in $\mathcal{A}$ $\bot_A = \bot$. Then the *union*, $\mathcal{U}_{CPO}(\mathcal{A})$, is defined by

— $|\mathcal{U}_{CPO}(\mathcal{A})| = \bigcup \{|A| \mid A \in \mathcal{A}\}$

— $\sqsubseteq_{\mathcal{U}_{CPO}(\mathcal{A})} = \bigcup \{\sqsubseteq_A \mid A \in \mathcal{A}\}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The finite subset cpo operator is also partial because it is defined only for flat cpo's.

Definition 7.13 (Finite subsets) Let A be a flat cpo with $\bot = \bot_A$. Then the cpo of its *finite subsets*, $\mathcal{S}_{CPO}(A)$, is defined as follows:

— $|\mathcal{S}_{CPO}(A)| = \mathbb{F}(|A| \setminus \{\bot\}) \cup \{\bot\}$,

— for $s_1, s_2 \in |\mathcal{S}_{CPO}(A)|$, $s_1 \sqsubseteq_{\mathcal{S}_{CPO}(A)} s_2$ iff $s_1 = \bot$ or $s_1 = s_2$.

Definition 7.14 (Cartesian product) Let $A_1, \ldots, A_n$ be cpo's with $\bot = \bot_{A_1} = \ldots = \bot_{A_n}$. Then their *smashed Cartesian product*, $\mathcal{P}_{CPO}(A_1, \ldots, A_n)$, is defined by:

— $|\mathcal{P}_{CPO}(A_1, \ldots, A_n)| = \bigtimes_{i=1}^{n} (|A_i| \setminus \{\bot\}) \cup \{\bot\}$

— $\bot \sqsubseteq_{\mathcal{P}_{CPO}(A_1,\ldots,A_n)} p$ for all $p \in |\mathcal{P}_{CPO}(A_1, \ldots, A_n)|$, and
for $(a_1, \ldots, a_n), (a'_1, \ldots, a'_n) \in |\mathcal{P}_{CPO}(A_1, \ldots, A_n)|$,
$(a_1, \ldots, a_n) \sqsubseteq_{\mathcal{P}_{CPO}(A_1,\ldots,A_n)} (a'_1, \ldots, a'_n)$ iff $a_1 \sqsubseteq_{A_1} a'_1$ and ... and $a_n \sqsubseteq_{A_n} a'_n$.

Definition 7.15 (Finite Sequences) Let A be a cpo with $\bot = \bot_A$. The cpo of finite sequences of elements of A, $\mathcal{L}_{CPO}(A)$, is defined by

— $|\mathcal{L}_{CPO}(A)| = \mathbb{L}(|A| \setminus \{\bot\}) \cup \{\bot\}$

— $\bot \sqsubseteq_{\mathcal{L}_{CPO}(A)} l$ for all $l \in |\mathcal{L}_{CPO}(A)|$, and
for $l_1, l_2 \in |\mathcal{L}_{CPO}(A)| \setminus \{\bot\}$, $l_1 \sqsubseteq_{\mathcal{L}_{CPO}(A)} l_2$ iff $\underline{\text{len}}(l_1) = \underline{\text{len}}(l_2)$ and
for $i \in \{1, \ldots, \underline{\text{len}}(l_1)\}$, $l_1(i) \sqsubseteq_A l_2(i)$.

Definition 7.16 (Record space) Let $id \in Id$ be a VDM-SL identifier, and let $A_1, \ldots, A_n$ be cpo's with $\bot = \bot_{A_1} = \ldots = \bot_{A_n}$. Then the *smashed record cpo*, $\mathcal{R}^{id}_{CPO}(A_1, \ldots, A_n)$, is defined by:

— $|\mathcal{R}^{id}_{CPO}(A_1, \ldots, A_n)| = (\{id\} \times \bigtimes_{i=1}^{n} (|A_i| \setminus \{\bot\})) \cup \{\bot\}$

— $\bot \sqsubseteq_{\mathcal{R}^{id}_{CPO}(A_1,\ldots,A_n)} r$ for all $r \in |\mathcal{R}^{id}_{CPO}(A_1, \ldots, A_n)|$, and
for $(id, a_1, \ldots, a_n), (id, a'_1, \ldots, a'_n) \in |\mathcal{R}^{id}_{CPO}(A_1, \ldots, A_n)|$,
$(id, a_1, \ldots, a_n) \sqsubseteq_{\mathcal{R}^{id}_{CPO}(A_1,\ldots,A_n)} (id, a'_1, \ldots, a'_n)$ iff $a_1 \sqsubseteq_{A_1} a'_1$ and ... and $a_n \sqsubseteq_{A_n} a'_n$.

The following operator on cpo's corresponds to the finite mapping constructor of VDM-SL. The domain of a mapping is required to be a flat cpo.

Definition 7.17 (Mapping space) Let A be a flat cpo and B be a cpo with $\bot = \bot_A = \bot_B$. Then the *cpo of smashed mappings* from A to B, $\mathcal{M}_{CPO}(A, B)$, is defined as follows:

— $|\mathcal{M}_{CPO}(A, B)| = \mathbb{M}(|A| \setminus \{\bot\}, |B| \setminus \{\bot\}) \cup \{\bot\}$

— $\bot \sqsubseteq_{\mathcal{M}_{CPO}(A,B)} m$ for all $m \in |\mathcal{M}_{CPO}(A, B)|$, and
for $m_1, m_2 \in |\mathcal{M}_{CPO}(A, B)| \setminus \{\bot\}$, $m_1 \sqsubseteq_{\mathcal{M}_{CPO}(A,B)} m_2$ iff
$\delta_0(m_1) = \delta_0(m_2)$ and for all $a \in \delta_0(m_1)$, $m_1(a) \sqsubseteq_B m_2(a)$.

The next operator corresponds to the function space constructor $D_1 \to D_2$. The definition is like the usual definition for cpo's with the addition of $\bot$ as a new least element.

Definition 7.18 (Function space) Let A and B be cpo's with $\bot = \bot_A = \bot_B$. Then the *cpo of functions* from A to B, $\mathcal{F}_{CPO}(A, B)$, is defined by:

— $|\mathcal{F}_{CPO}(A, B)| = (|A| \to |B|) \cup \{\bot\}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— $\bot \sqsubseteq_{\mathcal{F}_{CPO}(A,B)} f$ for all $f \in |\mathcal{F}_{CPO}(A,B)|$, and for $f, g \in |\mathcal{F}_{CPO}(A,B)| \setminus \{\bot\}$, $f \sqsubseteq_{\mathcal{F}_{CPO}(A,B)} g$ iff for all $a \in A$, $f(a) \sqsubseteq_B g(a)$.

The last operator on cpo's required is *tagging*, which produces an isomorphic copy of a cpo by attaching a tag to its elements.

Definition 7.19 (Tagging) Let A be a cpo with $\bot = \bot_A$. For any $t \in TAG$, *tagging* A with t yields a cpo $\mathcal{T}^t_{CPO}(A)$ defined as follows:

— $|\mathcal{T}^t_{CPO}(A)| = \{(t, a) \mid a \in (|A| \setminus \{\bot\})\} \cup \{\bot\}$

— $\bot \sqsubseteq_{\mathcal{T}^t_{CPO}(A)} e$ for all $e \in |\mathcal{T}^t_{CPO}(A)|$, and
for $(t, a_1), (t, a_2) \in |\mathcal{T}^t_{CPO}(A)|$, $(t, a_1) \sqsubseteq_{\mathcal{T}^t_{CPO}(A)} (t, a_2)$ iff $a_1 \sqsubseteq_A a_2$.

Here TAG is taken to be an abbreviation:

$$TAG = \{\text{'}bool\text{'}, \text{'}char\text{'}, \text{'}nil\text{'}, \text{'}token\text{'}, \text{'}num\text{'}, \text{'}quot\text{'}, \text{'}set\text{'}, \text{'}tuple\text{'}, \text{'}seq\text{'}, \text{'}record\text{'}, \text{'}map\text{'}, \text{'}fun\text{'}\} \cup QUOTE$$

where $QUOTE$ is a countably infinite set of uniquely identified quotations to which the user of VDM-SL can get access.

7.1.2 A Universe of Complete Partial Orders

In this clause a universe of cpo's will be constructed. The details of the construction will be slightly different from the definitions given in TW90 but the underlying principle is exactly the same.

The universe of cpo's is a family of cpo's which contains all the basic cpo's (corresponding to the basic domains of VDM-SL) and is closed under all the operators defined in the previous clause as well as under unions of countably many, union-compatible sets of cpo's. The construction starts with the family of basic cpo's and proceeds by closing it under the operators. Then, the unions of all countably many, union-compatible sets of cpo's in the family are added. Unfortunately, since not all the operators defined (notably, the function space operator) preserve the unions added, the family of cpo's obtained is not closed under the operators. Hence, it is necessary to iterate the procedure again, and again, and .... Fortunately, the length of this iteration can be bounded: it is not necessary to iterate further than to the first uncountable ordinal, ω_1.

Construction 7.20 (Cpo universe) The basic cpo's are:

$$\begin{array}{lcl}
\mathsf{Bool\text{-}cpo} & = & \mathcal{T}^{\text{'}bool\text{'}}_{CPO}(\mathbb{B}_\bot) \\
\mathsf{char\text{-}cpo} & = & \mathcal{T}^{\text{'}char\text{'}}_{CPO}(CHAR_\bot) \\
\mathsf{nil\text{-}cpo} & = & \mathcal{T}^{\text{'}nil\text{'}}_{CPO}(\{\underline{\mathrm{nil}}\}_\bot) \\
\mathsf{token\text{-}cpo} & = & \mathcal{T}^{\text{'}token\text{'}}_{CPO}(QUOTE_\bot) \\
\mathsf{NUM\text{-}CPOS} & = & \{\mathcal{T}^{\text{'}num\text{'}}_{CPO}(\mathbb{N}_\bot), \mathcal{T}^{\text{'}num\text{'}}_{CPO}(\mathbb{N}_{1\bot}), \mathcal{T}^{\text{'}num\text{'}}_{CPO}(\mathbb{Z}_\bot), \mathcal{T}^{\text{'}num\text{'}}_{CPO}(\mathbb{Q}_\bot), \mathcal{T}^{\text{'}num\text{'}}_{CPO}(\mathbb{R}_\bot)\} \\
\mathsf{QUOTE\text{-}CPOS} & = & \{\mathcal{T}^{\text{'}quot\text{'}}_{CPO}(\{q\}_\bot) \mid q \in QUOTE\}
\end{array}$$

$CHAR$ is simply a finite set containing an encoding for all the characters that can be supplied by the user of VDM-SL.

A transfinite (but bounded) cumulative hierarchy U_α, $\alpha < \omega_1$, of families of cpo's is defined by:

— $U_0 = \{\mathsf{Bool\text{-}cpo}, \mathsf{char\text{-}cpo}, \mathsf{nil\text{-}cpo}, \mathsf{token\text{-}cpo}\} \cup \mathsf{NUM\text{-}CPOS} \cup \mathsf{QUOTE\text{-}CPOS} \cup \{\emptyset_\bot\}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— for any ordinal $\alpha < \omega_1$,

$$\begin{aligned} U_{\alpha+1} = U_\alpha \;&\cup\; \{\mathcal{U}_{CPO}(\{D_1,\ldots,D_n\}) \mid D_1,\ldots,D_n \in U_\alpha \wedge D_1,\ldots,D_n \text{ is a union-compatible family}\} \\ &\cup\; \{\mathcal{T}^{\text{'set'}}_{CPO}(\mathcal{S}_{CPO}(D)) \mid D \in U_\alpha \wedge D \text{ is flat}\} \\ &\cup\; \{\mathcal{T}^{\text{'tuple'}}_{CPO}(\mathcal{P}_{CPO}(D_1,\ldots,D_n)) \mid D_1,\ldots,D_n \in U_\alpha\} \\ &\cup\; \{\mathcal{T}^{\text{'seq'}}_{CPO}(\mathcal{L}_{CPO}(D)) \mid D \in U_\alpha\} \\ &\cup\; \{\mathcal{T}^{\text{'record'}}_{CPO}(\mathcal{R}^{id}_{CPO}(D_1,\ldots,D_n)) \mid id \in Id \wedge D_1,\ldots,D_n \in U_\alpha\} \\ &\cup\; \{\mathcal{T}^{\text{'map'}}_{CPO}(\mathcal{M}_{CPO}(D_1,D_2)) \mid D_1,D_2 \in U_\alpha \wedge D_1 \text{ is flat}\} \\ &\cup\; \{\mathcal{T}^{\text{'fun'}}_{CPO}(\mathcal{F}_{CPO}(D_1,D_2)) \mid D_1,D_2 \in U_\alpha\} \\ &\cup\; \{\mathcal{T}^{q}_{CPO}(D) \mid D \in U_\alpha, q \in QUOTE\}. \end{aligned}$$

— for any limit ordinal $\alpha < \omega_1$, U_α is the family of the unions of all countably many, union-compatible sets of cpo's in $\bigcup_{\beta<\alpha} U_\beta$.

Finally, the cpo universe is defined as the union of the families in the above hierarchy:

$$CPO = \bigcup_{\alpha<\omega_1} U_\alpha.$$

Proposition 7.21 The cpo universe *CPO* as defined above contains the basic cpo's, has a least element[1)], and is closed under all the operators on cpo's defined in clause 7.1.1.2 as well as under unions of countably many, union-compatible sets of cpo's. The proof for this proposition is given in TW90.

7.1.3 A Universe of Domains

The universe of cpo's, *CPO*, is rich enough for interpreting all the domain constructors of VDM as well as certain recursive domain equations over these constructors. However, *CPO* does not support the interpretation of invariant constrained domains. A solution to this problem is given in the following.

Definition 7.22 (VDM domain) A *VDM domain* is a pair:

$$A = ((|A|, \sqsubseteq_A), \|A\|)$$

where $(|A|, \sqsubseteq_A)$ is a cpo (from *CPO*) and $\|A\| \subseteq |A| \setminus \{\bot_A\}$.

Having defined the concept of a VDM domain, the last step of universe construction is:

Construction 7.23 (Domain universe) The universe of domains for VDM, *DOM*, is defined by:

$$DOM = \{((|A|, \sqsubseteq_A), \|A\|) \mid (|A|, \sqsubseteq_A) \in CPO \wedge \|A\| \subseteq |A| \setminus \{\bot_A\}\}.$$

It is now necessary to define the operators on *DOM* necessary for interpreting the VDM domain constructors. Each of these — except for the operator for modelling invariant constraints corresponds to one of the operators defined on cpo's in clause 7.1.1.2. In fact, they are all related to their counterparts on cpo's in that they behave in exactly the same way on the cpo part of their arguments — they are said to *extend* the cpo operators.

Let $A = ((|A|, \sqsubseteq_A), \|A\|)$ be a domain. In the following the selector function $\mathcal{D}(A)$ will denote a cpo $(|A|, \sqsubseteq_A)$ (i.e. $\mathcal{D}$ is a function from *DOM* to *CPO*). Similarly given a domain A, $|A|$ and $\|A\|$ can be considered as selection functions. $\mathcal{D}^*$ denotes the extension of $\mathcal{D}$ to the case of sequences of domains.

Definition 7.24 (VDM domain operators) For each of the operators on *CPO* defined in clause 7.1.1.2, its extension to a *domain operator* on *DOM* is:

[1)] The least element is $\emptyset_\bot$, with respect to the traditional subset inclusion ordering taken pairwise both on the carrier set and the partial order.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$\mathsf{Bool\text{-}dom} = (\mathsf{Bool\text{-}cpo}, \{\text{`}bool\text{'}\} \times \mathbb{B})$

$\mathsf{char\text{-}dom} = (\mathsf{char\text{-}cpo}, \{\text{`}char\text{'}\} \times CHAR)$

$\mathsf{nil\text{-}dom} = (\mathsf{nil\text{-}cpo}, \{\text{`}nil\text{'}\} \times \{\underline{\mathrm{nil}}\})$

$\mathsf{token\text{-}dom} = (\mathsf{token\text{-}cpo}, \{\text{`}token\text{'}\} \times QUOTE)$

$\mathsf{nat\text{-}dom} = (\mathcal{T}_{CPO}^{\text{`}num\text{'}}(\mathbb{N}_\perp), \{\text{`}num\text{'}\} \times \mathbb{N})$

$\mathsf{natone\text{-}dom} = (\mathcal{T}_{CPO}^{\text{`}num\text{'}}(\mathbb{N}_{1\perp}), \{\text{`}num\text{'}\} \times \mathbb{N}_1)$

$\mathsf{int\text{-}dom} = (\mathcal{T}_{CPO}^{\text{`}num\text{'}}(\mathbb{Z}_\perp), \{\text{`}num\text{'}\} \times \mathbb{Z})$

$\mathsf{rat\text{-}dom} = (\mathcal{T}_{CPO}^{\text{`}num\text{'}}(\mathbb{Q}_\perp), \{\text{`}num\text{'}\} \times \mathbb{Q})$

$\mathsf{real\text{-}dom} = (\mathcal{T}_{CPO}^{\text{`}num\text{'}}(\mathbb{R}_\perp), \{\text{`}num\text{'}\} \times \mathbb{R})$

$\mathsf{QUOTE\text{-}DOMS} = \{(\mathcal{T}_{CPO}^{\text{`}quot\text{'}}(\{q\}_\perp), \{(\text{`}quot\text{'}, q)\}) \mid q \in QUOTE\}$

$\mathcal{U}(\mathcal{A}) = (\mathcal{U}_{CPO}(\{\mathcal{D}(A) \mid A \in \mathcal{A}\}), \bigcup\{\|A\| \mid A \in \mathcal{A}\})$

$\mathcal{S}(A) = (\mathcal{T}_{CPO}^{\text{`}set\text{'}}(\mathcal{S}_{CPO}(\mathcal{D}(A)), \{\text{`}set\text{'}\} \times \mathbb{F}(\|A\|))$

$\mathcal{P}(A_1, \ldots, A_n) = (\mathcal{T}_{CPO}^{\text{`}tuple\text{'}}(\mathcal{P}_{CPO}(\mathcal{D}^*(A_1, \ldots, A_n))), \{\text{`}tuple\text{'}\} \times \bigtimes_{i=1}^{n} \|A_i\|)$

$\mathcal{L}(A) = (\mathcal{T}_{CPO}^{\text{`}seq\text{'}}(\mathcal{L}_{CPO}(\mathcal{D}(A))), \{\text{`}seq\text{'}\} \times \mathbb{L}(\|A\|))$

$\mathcal{R}^{id}(A_1, \ldots, A_n) = (\mathcal{T}_{CPO}^{\text{`}record\text{'}}(\mathcal{R}_{CPO}^{id}(\mathcal{D}^*(A_1, \ldots, A_n)), \{\text{`}record\text{'}\} \times (\{id\} \times \bigtimes_{i=1}^{n} \|A_i\|))$

$\mathcal{M}(A, B) = (\mathcal{T}_{CPO}^{\text{`}map\text{'}}(\mathcal{M}_{CPO}(\mathcal{D}(A), \mathcal{D}(B))), \{\text{`}map\text{'}\} \times \mathbb{M}(\|A\|, \|B\|))$

$\mathcal{F}(A, B) = (\mathcal{T}_{CPO}^{\text{`}fun\text{'}}(\mathcal{F}_{CPO}(\mathcal{D}(A), \mathcal{D}(B))),$

$\qquad \{\text{`}fun\text{'}\} \times \{f : |A| \to |B| \mid \forall a \in \|A\| \cdot f(a) \in |B| \wedge \forall a \in (|A| \backslash \|A\|) \cdot f(a) = \perp\})$

$\mathcal{T}^t(A) = (\mathcal{T}_{CPO}^t(\mathcal{D}(A)), \{t\} \times \|A\|).$

Representing the invariant as the set of values for which it holds, the operator for invariant constraint is given by:

Definition 7.25 (Invariant constraint) Let A be a VDM domain and let S be a set of values. Then the *invariant constrained* domain $\mathcal{I}(S, A)$ is defined by:

$$\mathcal{I}(S, A) = (\mathcal{D}(A), \|A\| \cap S).$$

Definition 7.26 (Direct summand) A cpo A is a *direct summand* of a cpo B, written $A \sqsubseteq_{\mathsf{DS}} B$, if for some cpo D such that $|A| \cap |D| = \{\perp\}$, $B = \mathcal{U}(\{A, D\})$.

Definition 7.27 (Subdomain) A domain A is a *subdomain* of a domain B, written $A \sqsubseteq_{DOM} B$, if:

— $(|A|, \sqsubseteq_A) \sqsubseteq_{\mathsf{DS}} (|B|, \sqsubseteq_B)$, and,

— $\|A\| \subseteq \|B\|$.

From TW90 it follows that any set of possibly recursive domain definitions may be interpreted in the universe DOM in the usual way (i.e. the recursive set of equations on domains has a least solution in DOM) provided that:

a) The partial operators (i.e. $\mathcal{S}, \mathcal{M}$, and $\mathcal{U}$) are applied only to domains within their domain of definition.

b) The function space is not involved in an essentially recursive way.

Notice that the function space operator may be freely applied to domains defined in a recursive way, as well as be used as already defined domains in recursive definitions of other domains. In this way it is legal to make a type definition producing binary trees with functions on the leaves.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2 The Semantic Domains

This clause introduces the semantic domains which are used in the definition of the dynamic semantics. It should be noted that the domains defined in this clause are meta-domains, i.e. they are used in the definition of the dynamic semantics of VDM-SL. Thus, these domains will not necessarily have the order-theoretic properties required of VDM domains as they have been defined in the previous clause.

This clause will first present the basic semantic domains which form the basis for the 'pure' models which a specification can denote. These models (ENV_{PURE})[1)] are called 'pure' because they do not contain any of the extra values which are used for defining the dynamic semantics. After that the extended semantic domains which are used in the definition of the dynamic semantics are given. This is followed by an outline of the semantic domains used for the evaluation functions (producing definers and evaluators for the different kinds of constructs). Finally, a number of auxiliary semantic domains are listed.

7.2.1 Basic Semantic Domains

The semantic domains presented in this clause serve as the basis for the definition of the dynamic semantics. These domains are built on top of the domain universe constructed in the previous clause. Some domains can be considered as extensions of the domain universe while others can be considered as named subsets of the domain universe.

1. $ENV_{PURE} = \mathbb{E}(VAL \cup DOM \cup OPVAL \cup POLYVAL)$
2. $VAL = \bigcup\{ |A| \mid ((|A|, \sqsubseteq_A), \|A\|) \in DOM \}$
3. $DOM = \{ ((|A|, \sqsubseteq_A), \|A\|)$
   $\mid (|A|, \sqsubseteq_A) \in CPO, \|A\| \subseteq |A| \setminus \{\bot\} \}$
4. $OPVAL = \{ \{ (i_v, i_st, o_st, o_v, m)$
   $\mid i_v \in \|t_1\| \cup \{\bot\}, i_st, o_st \in (\|st\| \cup \{\bot, \underline{\text{nil}}\}),$
   $o_v \in (\|t_2\| \cup \{\bot, \underline{\text{nil}}\}), m \in MODE \}$
   $\mid t_1, t_2, st \in FLATDOM \}$
5. $FLATDOM = \{ d \mid d \in DOM \cdot \forall a_1, a_2 \in |d| \cdot a_1 \sqsubseteq_d a_2 \Leftrightarrow a_1 = \bot \vee a_1 = a_2 \}$
6. $FLATVAL = \bigcup\{ |d| \mid d \in FLATDOM \}$
7. $POLYVAL = \mathbb{L}_1(DOM) \to VAL$
8. $POLYDOM = \mathbb{L}_1(DOM) \to DOM$
9. $MODE = \{ \underline{\text{cont}}, \underline{\text{exit}}, \underline{\text{ret}} \}$

Annotations to the Basic Semantic Domains:

1. ENV_{PURE}: all 'pure' environments which map identifiers to their denotation. The values in the range can either be (monomorphic) values, domains, operations or polymorphic functions. The (monomorphic) values and the domains come directly from the domain universe, while the others can be considered as built on top of the domain universe.
2. *VAL*: all values constructed in the domain universe. The user of VDM-SL may define values in this value universe by means of the cpo operators. Of course, such definitions may be recursive. Since the value universe, *VAL*, as a whole does not have a cpo structure, in order to interpret such a recursive definition it is not sufficient to know that it is to define an element in *VAL*. It has to be pointed out which of the component cpo's (domains) the definition is to be interpreted in. This is taken into account in the definition of the dynamic semantics in places where the standard least fixed point technique is used.
3. *DOM*: all domains constructed in the domain universe.
4. *OPVAL*: all operation values built on top of the domain universe. Notice how operations are interpreted using a relational style. An operation value is a set of tuples (each with five elements). A tuple contains an input value, state values before and after the evaluation of an operation, an output value[2)], and a flag indicating the mode of the operation[3)]. Do also notice that bottom, $\bot$, is allowed as a normal value. This is necessary in order to give an erratic nondeterministic interpretation.

[1)] A model for a specification can also be viewed as an environment for its 'outside' world and therefore the term ENV_{PURE} is used.
[2)] Notice that if an operation does not return a value, this is modelled in the dynamic semantics by the return of nil.
[3)] State, input and output values must all be flat (since sets containing non-flat elements are not contained within the domain universe).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

5. *FLATDOM*: all domains which fulfill the flatness criterion (i.e. $FLATDOM \subseteq DOM$)[4].
6. *FLATVAL*: all flat values (i.e. $FLATVAL \subseteq VAL$).
7. *POLYVAL*: all polymorphic functions[1]. The polymorphic functions are represented as functions from a list of domains to values. The argument domains correspond to an actual instantiation of the function's type variables.
8. *POLYDOM*: all domains for the polymorphic functions.
9. *MODE*: a mode is used in the denotation of operations. The mode indicated whether an execution of an operation ended with an error flag (an exit quote), or a value was returned from the operation (a ret quote) or no value was returned (by cont showing that execution can continue).

End of annotations

[4] All domains involving the function space operator are ruled out in this way.

[1] The kind of polymorphism which is present is called "first order explicit polymorphism".

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.2 Extended Semantic Domains

This clause introduces a number of extended semantic domains which are used in the definition of the dynamic semantics. Values from these domains are however, not 'visible' at the outermost level (i.e. the set of models which an entire specification denotes).

10. $ENV = ENV_{PURE} \cup \mathbb{E}(MOD_FUN \cup SEL_FUN \cup CONS_FUN \cup STATE_FUN \cup TAG_ENV \cup LOC_ENV)$
11. $MOD_FUN = \mathbb{E}(VAL) \rightarrow RECORD_VAL \rightarrow VAL$
12. $SEL_FUN = RECORD_VAL \rightarrow VAL$
13. $CONS_FUN = \mathbb{L}(VAL) \rightarrow RECORD_VAL_{\perp}$
14. $STATE_FUN = \mathbb{F}(ExtVarInf) \rightarrow (VAL \cup \{\underline{\text{nil}}\} \times VAL \cup \{\underline{\text{nil}}\}) \rightarrow \mathbb{B}$
15. $TAG_ENV = \mathbb{E}(CompositeType)$
16. $LOC_ENV = \mathbb{E}(DOM)$

Annotations to Extended Semantic Domains:

10. *ENV*: all possible (extended) environments which are used in the definition of the dynamic semantics. The extension to the 'pure' environments consists of modifier functions (used for record values), selector functions (also used for record values), and a function for checking the state constancy (used to ensure that read components of the state are left unchanged by operations). In addition a tag environment (used to check that the tags of composite types in the entire specification fit together) and a local environment (used for type checking local variables inside explicitly defined operations).
11. *MOD_FUN*: all modifier functions used for record values[1)]. The 'pure' environment is expanded with modifier functions for all record types and they will all belong to this domain.
12. *SEL_FUN*: all selector functions used for record values. The 'pure' environment is expanded with selector functions for all record types and they will all belong to this domain.
13. *CONS_FUN*: all constructor functions used for record values. The 'pure' environment is expanded with constructor functions for all record types and they will all belong to this domain.
14. *STATE_FUN*: a function checking the state constancy. There will only be one function used from this domain for each specification. It is used to ensure that read components of the state are left unchanged by operations. Note that *ExtVarInf* is defined in clause 6.2.5 in clause 6.
15. *TAG_ENV*: a local environment which is embedded in the global environment, *ENV*. This 'tag environment' contains information about the composite types which are used in the type definitions part of the entire specification. All composite types which are used in other parts of the specification must also be defined in the type definition part. Thus, this local environment is used to ensure this. Note that *CompositeType* is defined in clause 6.2.1 in clause 6.
16. *LOC_ENV*: a local environment which is embedded in the environment, *ENV*. This domain environment contains information about the type of the temporary variables used inside block statements in operations. Thus this domain is present in order to type-check usages of local variables in operations.

End of annotations

[1)] The domain for record values called *RECORD_VAL* is defined in clause 7.2.4.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.3 Semantic Domains for Evaluation Functions

This clause defines a number of combinations of the basic semantic domains and the extended semantic domains which are often used in the definition of the dynamic semantics. They are called definers and evaluators. Because of the presence of looseness within VDM-SL there will also exist loose definers and loose evaluators[1]. Thus, a definer can be considered as something which changes the 'current' environment and an evaluator can be considered as the semantic counterpart of a piece of syntax.

17. $LDef = \mathbb{P}(Def)$
18. $Def = ENV \to (ENV \cup \{\underline{\text{err}}\})$
19. $TEval = ENV \to (DOM \cup \{\underline{\text{err}}\})$
20. $LSEval = \mathbb{P}(SEval)$
21. $SEval = ENV \to (ENV \times MODE \times (VAL \cup \{\underline{\text{nil}}\}))$
22. $LPatEval = \mathbb{P}(PatEval)$
23. $PatEval = VAL \to ENV \to (VEnv \cup \{\underline{\text{err}}, \underline{\text{unmatch}}\})$
24. $LBindEval = \mathbb{P}(BindEval)$
25. $BindEval = ENV \to \mathbb{P}(VEnv \cup \{\underline{\text{err}}\})$
26. $VEnv = \mathbb{E}(VAL)$
27. $LEEval = \mathbb{P}(EEval)$
28. $EEval = ENV \to VAL$
29. $LPEval = \mathbb{P}(PEval)$
30. $PEval = ENV \to POLYVAL$
31. $PDomEval = ENV \to POLYDOM$

Annotations to the Semantic Evaluation Domains:

17. *LDef*: all loose definers (i.e. a set of definers).
18. *Def*: all definers are total functions on environments which may return an error flag.
19. *TEval*: all type evaluators are total functions from environments to either a domain or an error flag. Notice that type definitions cannot be loosely specified, and therefore there is no domain for loose type evaluators.
20. *LSEval*: all loose statement evaluators (i.e. a set of statement evaluators).
21. *SEval*: all statement evaluators are total functions from environments to a three-tuple, which contains evaluation information. The first component is a (possibly) changed environment (even though only state parts are changed, whole environments are returned). The second component is a flag indicating whether evaluation ended normally (with (ret) or without (cont) a value) or an exit was encountered (exit). The third component is either a value or a flag indicating that evaluation did not return any value. This value component is used for operations which either exit or return a value.
22. *LPatEval*: all loose pattern evaluators (i.e. a set of pattern evaluators).
23. *PatEval*: all pattern evaluators are total (Curried) functions from values (to be matched against the pattern) to a total function, from environments to a value environment or two different kinds of flags. unmatch is a flag which indicate that the matching simply failed in the given environment. err is an error flag which is used if a real error has occurred in the pattern (i.e. if the pattern contains a match value which evaluates to bottom).
24. *LBindEval*: all loose binding evaluators (i.e. a set of binding evaluators).
25. *BindEval*: all binding evaluators are total functions from environments to a (possibly empty) set of value environments[2] or an error flag. The value environments correspond to pattern evaluators matching against values belonging to a type or a set (in the case of loose specification of the set value different binding evaluators will be present in a loose binding evaluator). The error flag is used if a 'real' error has occurred: Either the pattern have contained a match value which yielded bottom or the set expression does not yield a set (or a type is erroneous).
26. *VEnv*: all value environments are mappings from identifiers to values. An auxiliary domain used only for clarity (i.e. $Venv \subseteq ENV$).

[1] Notice that this way of modelling looseness implies that definers and evaluators are deterministic.

[2] The returning value environments could just as well be described with the domain ENV, but for clarity the auxiliary domain $VEnv$ is introduced.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

27. *LEEval*: all loose expression evaluators (i.e. sets of expression evaluators).
28. *EEval*: all expression evaluators are total functions from environments to values.
29. *LPEval*: all loose polymorphic function evaluators (i.e. sets of polymorphic function evaluators).
30. *PEval*: all polymorphic function evaluators are total functions from environments to polymorphic function values.
31. *PDomEval*: all polymorphic domain evaluators are total functions from environments to polymorphic function domains. There is no way the evaluation can return an error in this case, so no explicit error flag is used.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.2.4 Auxiliary Semantic Domains

In the definition of the dynamic semantics a number of abbreviations for subsets of the sets of values in the universe, *VAL* are used. An abbreviation for record parts of domains in *DOM* will be used in the definition. It should be noted that the domain universe operators (from clause 7.1.3) supply the values (and the domain) with the appropriate tags (e.g. '*map*'). These auxiliary semantic domains are listed here.

32. $FUN_VAL = \bigcup\{ \|\mathcal{F}(d_1, d_2)\| \mid d_1,\ d_2 \in DOM\}$
33. $MAP_VAL = \bigcup\{ \|\mathcal{M}(d_1, d_2)\| \mid d_1 \in FLATDOM,\ d_2 \in DOM\}$
34. $LIST_VAL = \bigcup\{ \|\mathcal{L}(d)\| \mid d \in DOM\}$
35. $SET_VAL = \bigcup\{ \|\mathcal{S}(d)\| \mid d \in FLATDOM\}$
36. $TUPLE_VAL = \bigcup\{ \|\mathcal{P}(d_1, \ldots, d_n)\| \mid d_1,\ldots,d_n \in DOM,\ n \in \omega\}$
37. $RECORD_VAL = \bigcup\{ \|\mathcal{R}^{id}(d_1, \ldots, d_n)\| \mid id \in Id,\ d_1,\ldots,d_n \in DOM,\ n \in \omega\}$
38. $RECORD_VAL_\perp = RECORD_VAL \cup \{\perp\}$
39. $NUM_VAL = \|\text{real-dom}\|$
40. $BOOL_VAL = \|\text{Bool-dom}\|$
41. $RECORD_DOM = \{\mathcal{R}^{id}(d_1,\ldots,d_n) \mid id \in Id,\ d_1,\ldots,d_n \in DOM,\ n \in \omega\}$
42. $QUASI_CPO = \{(\bigcup\{|A| \mid A \in F\}, \bigcup\{\sqsubseteq_A \mid A \in F\}) \mid F \in \mathbb{F}(CPO)\}$
43. $QUASI_DOM = \{((|A|, \sqsubseteq_A), \|A\|) \mid (|A|, \sqsubseteq_A) \in QUASI_CPO \cdot \|A\| \subseteq |A| \setminus \{\perp\}\}$

Annotations to the Auxiliary Semantic Domains:

32. *FUN_VAL*: all total functions.
33. *MAP_VAL*: all mapping values.
34. *LIST_VAL*: all list (or sequence) values.
35. *SET_VAL*: all set values.
36. *TUPLE_VAL*: all tuple values (Cartesian products).
37. *RECORD_VAL*: all record values.
38. $RECORD_VAL_\perp$: all record values and bottom.
39. *NUM_VAL*: all numeric values (the reals).
40. *BOOL_VAL*: all Boolean values.
41. *RECORD_DOM*: all record domains.
42. *QUASI_CPO*: all pairs with set of values in the first component and their ordering in the second component. Thus, the elements of this domain can be CPO's, but are not necessarily so.
43. *QUASI_DOM*: all pairs with a quasi-cpo in the first component and a restriction set (modelling values fulfilling an invariant) in the second component. Thus, the elements of this domain can belong to *DOM*, but do not necessarily do so.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8 The Dynamic Semantics

This clause is organized as follows: First a summary of the overall structure of this definition of the VDM-SL language semantics is given. Then the semantic definition itself is presented in clauses following the structure of the core abstract syntax. After this a number of different kinds of auxiliary functions are introduced.

The semantics of a syntactic specification written in VDM-SL is given by the *SemSpec* function. A syntactic specification is simply a collection of definitions. The semantics of such a collection of definitions is the (possibly infinite) set of models fulfilling all the definitions. A model will fulfil a collection of definitions if it provides appropriate denotations for all the definitions[1)].

The semantics of a syntactic specification is a set of models because the definitions may be loosely specified. Loose specification arises because a specification generally needs to be stated at a higher level of abstraction than that of the final implementation of the system. Thus, when a construct is loosely specified it will have more than one 'legal' implementation; legal in the sense that it implements the loosely specified construct according to certain implementation relations.

The *SemSpec* function is given in an implicit style by testing all possible models against the *IsAModelOf* predicate. This is in contrast to the explicit style which is traditionally used for giving semantics to programming languages. In order to give a compositional semantics the explicit style requires an order in which the definitions must appear. The implicit style is used because VDM-SL does not have this property.

The *IsAModelOf* predicate shows how a syntactic specification is related to an already given (candidate) model (considered as an environment for the 'outside' world). This predicate is defined by various verification predicates which again are expressed in terms of evaluation functions. The *Verify*-predicates check whether or not the given model is possible for different kinds of constructs. Most of the *Eval*-functions yield a set of value evaluators because of possible underdeterminedness of functions, expressions and patterns. Typically, this affects the verification predicates in that the value in the given model must be a member of the set of value evaluators (applied with the model as an environment) returned by the specific evaluation function. Prior to calling verification predicates, *IsAModelOf* will expand the given model with constructs which are implicitly defined. The model is not expected to contain a denotation for these constructs (e.g. selector functions in record types), so the model will not make the denotations of such constructs visible to the 'outside' world. If the syntactic specification contains the definition of a record type then the model should contain the corresponding domain for that record type, but not the selector functions used to take components out of that domain. Finally the definition contains a number of auxiliary functions and predicates.

The semantic functions and predicates used in the dynamic semantics have been provided with annotations explaining the purpose of the function/predicate and also explaining what specific parts of the function/predicate are meant to do. In a few places, extra examples are presented. The purpose of these examples is to illustrate the complexity which needs to be taken into account in the formal definition. Therefore, the examples can be read advantageously in parallel with understanding the formal description. Most of the examples focus on the effect looseness has on the semantics. This has been done because experience have shown that this part is particularly difficult to understand fully without examples.

8.1 Document

The top level abstract syntax of VDM-SL is a *Document*. A document is a specification which consists of a collection of definitions.

1. $SemSpec : Document \to \mathbb{P}(ENV_{PURE})$

1.1 $SemSpec(doc) \triangleq$
.2 $\quad \{ env \mid env \in ENV_{PURE} \cdot IsAModelOf(env, doc) \}$

Annotations to *SemSpec*:

[1)] Thus, a model can also be considered as the environment which the collection of definitions provides for the 'outside' world.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

1. This is the main function of the dynamic semantics. It takes a syntactic document (a specification) and yields the set of all possible models for a document. It does so in an implicit style by testing all possible models against the *IsAModelOf* predicate. This is in contrast to the explicit style which traditionally is used for giving semantics to for example programming languages. In order to give a compositional semantics the explicit style requires an order in which the definitions must appear. The implicit style is necessary because VDM-SL does not have this property.

End of annotations

2. $IsAModelOf$: **Pred**($ENV_{PURE} \times Document$)

2.1 $IsAModelOf(env, defs) \triangleq$
.2 **let** $MkTag('Definitions', t, epf, ef, ipf, if, v, eo, io, s) = defs$ **in**
.3 $DisjointIds(t, epf, ef, ipf, if, v, eo, io, s, env) \wedge$
.4 **let** $exp_env = ExpandImplDefs(t, s)(env)$ **in**
.5 $exp_env \neq \underline{\text{err}} \wedge$
.6 $VerifyTypes(t, exp_env) \wedge$
.7 $VerifyExplPolyFns(epf, exp_env) \wedge$
.8 $VerifyExplFns(ef, exp_env) \wedge$
.9 $VerifyImplPolyFns(ipf, exp_env) \wedge$
.10 $VerifyImplFns(if, exp_env) \wedge$
.11 $VerifyValues(v, exp_env) \wedge$
.12 $VerifyExplOps(eo, exp_env) \wedge$
.13 $VerifyImplOps(io, exp_env) \wedge$
.14 $VerifyStateDef(s, exp_env)$

Annotations to *IsAModelOf*:

2. This is the main predicate of the dynamic semantics. It is a predicate which states whether a syntactic definition is logically associated with a given model (*env*).
.3 All the globally defined constructs must be given disjoint names (i.e. no overloading is allowed). Thus, it is not possible in VDM-SL to use the same name for a type and for a function. This is ensured by the predicate *DisjointIds*.
.4 The given model, *env*, is expanded with constructs which are 'implicitly' defined. Thus, the model is not expected to provide inside information about, for instance, how to select components of record values. It is such 'implicit' definitions which are hidden from the 'outside' world.
.5 If this expansion process can go wrong with the given model, *env* is rejected.
.6–.14 The extended model must contain proper denotations for each of the constructs (e.g. types, functions).

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
3.    DisjointIds : Pred(𝔼(TypeDef) × 𝔼(ExplPolyFnDef) × 𝔼(ExplFnDef) ×
                         𝔼(ImplPolyFnDef) × 𝔼(ImplFnDef) ×
                         𝕄(Pattern, ValDef) × 𝔼(ExplOpDef) × 𝔼(ImplOpDef) ×
                         (StateDef ∪ {nil}) × ENV_PURE )
3.1   DisjointIds(m1, m2, m3, m4, m5, pm, m6, m7, sd, env) ≜
 .2     let s1 = dom(m1),
 .3         s2 = dom(m2),
 .4         s3 = dom(m3),
 .5         s4 = dom(m4),
 .6         s5 = dom(m5),
 .7         s6 = dom(m6),
 .8         s7 = dom(m7),
 .9         ps = ⋃{ ColIdSet(p) | p ∈ dom(pm) } in
 .10    let set_l = [ s1, s2, s3, s4, s5, s6, s7, ps],
 .11        st_ids = if sd = nil
 .12                 then { }
 .13                 else let MkTag('StateDef', (MkTag('ValueId', stid), _, _)) = sd in
 .14                        { MkTag('StateTypeId', stid), MkTag('InitId', stid) }
 .15    in
 .16       (dom(env) = ⋃elems(set_l) ∪ st_ids) ∧
 .17       (∀i, j ∈ inds(set_l) · i ≠ j ⇒ set_l(i) ∩ set_l(j) = { }) ∧
 .18       (∀id1, id2 ∈ dom(env) · (ShowTag(id1) = 'StateTypeId' ∧
 .19                                ShowTag(id2) = 'StateTypeId') ⇒ id1 = id2 ∧
 .20         ShowTag(id1) ∈ { 'ValueId', 'PreId', 'PostId', 'InvId', 'InitId', 'StateTypeId'})
```

Annotations to *DisjointIds*:

3. This auxiliary predicate is used to ensure that all definitions are uniquely identified, and that the domain of the model, *env*, is identical to the collection of all these identifiers.

.2–.8 Identifiers for all of the different kinds of constructs (e.g. type definitions, function definitions) are collected.

.9 Pattern identifiers from all value definitions are collected. The left hand side of a value definition can be an arbitrary pattern and therefore a more complex expression must be used here.

.11–.14 The identifiers related to the state definition are gathered.

.16 The given model must contain exactly all of the identifiers collected in its domain.

.17 All the identifiers in the different collection of syntactic definitions must be disjoint.

.18–.19 There must at the most be one identifier tagged as being the state type of the definition.

.20 The identifiers in a 'pure' environment may only be value identifiers, pre- and post-condition identifiers, invariant identifiers, initialisation identifiers and a state type identifier.

End of annotations

8.2 Definitions

A VDM-SL document (or specification) consists of a collection of definitions. This clause presents the top-level functions for the different kinds of definitions which can be used. In some cases there are both verification predicates and evaluation functions for a definition kind, and in other cases only verification predicates are presented here. The evaluation functions are presented in this clause if they exist (i.e. in some cases the verification predicate uses evaluation functions for the components of the definition instead of an evaluation function for the entire definition.

8.2.1 Type Definitions

This clause presents the verification predicate for type definitions and the corresponding evaluation functions for type definitions and types.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2.1.1 The Verification Predicate for Types

4. *VerifyTypes* : **Pred**(𝔼(*TypeDef*) × *ENV*)

4.1 *VerifyTypes*(*tpdef_m*, *env*) ≜
.2 **if** dom(*tpdef_m*) ⊆ dom(*env*) ∧ ∀*id* ∈ dom(*tpdef_m*) · *env*(*id*) ∈ *DOM*
.3 **then let** *env'* = *EvalTypeDef*(*tpdef_m*)(subtract(*env*, dom(*tpdef_m*))) **in**
.4 **if** *env'* = err
.5 **then** F
.6 **else** ∀*id* ∈ dom(*tpdef_m*) · *env*(*id*) = *env'*(*id*)
.7 **else** F

Annotations to *VerifyTypes*:

4. This predicate checks whether the given environment, *env*, contains proper denotations for all the syntactic type definitions. Note that types cannot be loosely specified.
.2 All names of the type definitions must belong to the domain of the environment and all of them must be bound to domains in the environment.
.3 The collection of all type definitions is evaluated in the given environment where the denotations for the type definitions have been removed.
.4–.5 If the evaluation reports an error, the given environment can be rejected as a possible model for the syntactic specification.
.6 Otherwise, all names of the type definitions must be bound to the same domain in the given environment and in the environment resulting from the evaluation of the type definitions.

End of annotations

An example of a recursive type definition with an invariant

Before proceeding with the semantics of type expressions, a small example is shown, illustrating how the least fixed point is calculated. The concrete syntax for this example looks like:

$T = T \times T \mid \mathbb{N}$

inv $t \triangleq$
  if *is*-$\mathbb{N}(t)$
  then true
  else let $(t_1, t_2) = t$ in
    $Greatest(t_1) \leq Smallest(t_2)$

where *Greatest* and *Smallest* are simple auxiliary functions with the domain $T \times T \mid \mathbb{N}$, providing respectively the greatest and the smallest number contained in the argument tree. Note that it is not necessary in the invariant to call recursively on t_1 and t_2: This is to do with the way the least fixed point for such a type definition is calculated.

In this case the first four steps of the calculation will look like:

(1) $T = (\emptyset_\perp, \{\ \})$
(2) $T = (\mathbb{N}_\perp, \mathbb{N})$
(3) $T = ((\mathbb{N} \mid \mathbb{N} \times \mathbb{N})_\perp, \mathbb{N} \cup T_1)$
(4) $T = ((\mathbb{N} \mid \mathbb{N} \times \mathbb{N} \mid (\mathbb{N} \times \mathbb{N}) \times \mathbb{N} \mid \mathbb{N} \times (\mathbb{N} \times \mathbb{N}) \mid (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}))_\perp,$
$\mathbb{N} \cup T_1 \cup T_2 \cup T_3 \cup T_4)$

where T_1, T_2, T_3, and T_4 correspond to the denotations from the following concrete type definitions:

$T_1 = \mathbb{N} \times \mathbb{N}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$\mathsf{inv}\ (t_1, t_2) \triangleq t_1 \leq t_2$

$T_2 = (\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$

$\mathsf{inv}\ (t_1, t_2) \triangleq \mathit{Greatest}(t_1) \leq t_2$

$T_3 = \mathbb{N} \times (\mathbb{N} \times \mathbb{N})$

$\mathsf{inv}\ (t_1, t_2) \triangleq t_1 \leq \mathit{Smallest}(t_2)$

$T_4 = (\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})$

$\mathsf{inv}\ (t_1, t_2) \triangleq \mathit{Greatest}(t_1) \leq \mathit{Smallest}(t_2)$

□

8.2.1.2 Evaluation Functions for Types

```
5.    EvalTypeDef : IE(TypeDef) → Def
5.1   EvalTypeDef(td_m)(env) ≙
 .2    let id_s = dom(td_m) in
 .3      let tev_m = { id ↦ EvalType(SelShape(td_m(id))) | id ∈ id_s } in
 .4        let inv_tf_s = EvalExplDef(ColInvFns(td_m)) in
 .5          if card(inv_tf_s) = 1
 .6          then let itf = Iota(inv_tf_s) in
 .7                 let d_tf =
 .8                     λappr ∈ DomApprox_err .
 .9                     if appr = err
 .10                    then err
 .11                    else let m =
 .12                            { id ↦ let d = tev_m(id)(overwrite(env, appr)),
 .13                                       env' = overwrite(env, appr) in
 .14                                   if d = err ∨ env' = err
 .15                                   then err
 .16                                   else let inv = itf(env')(SelInvId(td_m(id))) in
 .17                                          I({ e ∈ ||d|| | inv(e) = True() }, d)
 .18                            | id ∈ id_s } in
 .19                          if ∃id ∈ id_s · m(id) = err
 .20                          then err
 .21                          else m in
 .22                  if IsCont_DomApprox_err(d_tf)
 .23                  then let d_m = Y(d_tf) in
 .24                         if d_m = err
 .25                         then err
 .26                         else overwrite(env, d_m)
 .27                  else err
 .28         else err
```

Annotations to *EvalTypeDef*:

5. The function *EvalTypeDef* gives meaning to a set of possibly mutually recursive type definitions. The invariants which the definitions contain are included in the least fixed point iteration.

.4–.5 The invariants are evaluated and checked to ensured that they are not loosely specified.

.7–.21 A function $d_tf : \mathit{DomApprox}_{\underline{\mathrm{err}}} \rightarrow \mathit{DomApprox}_{\underline{\mathrm{err}}}$ is being defined. The set $\mathit{DomApprox}_{\underline{\mathrm{err}}} = \mathit{DomApprox} \cup \{\underline{\mathrm{err}}\}$, where $\mathit{DomApprox} = id_s \rightarrow \mathit{DOM}$ (see .2) has a structure of a cpo with an ordering defined by:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

— for $f, g \in DomApprox\ f \sqsubseteq g$ iff $\forall id \in id_s\ f(id) \sqsubseteq_{DOM} g(id)$

— $\lambda id : id_s\ .\ (\emptyset_\perp, \{\}) \sqsubseteq \underline{\text{err}}$.

The function $\lambda id : id_s\ .\ (\emptyset_\perp, \{\})$ is a bottom element. Note that err is uncomparable with any element of *DomApprox* except bottom.

.22–.26 If *d_tf* is continuous with respect to the ordering on $DomApprox_{\underline{\text{err}}}$, then the least fixed point of *d_tf* is calculated. If it is not equal to err, then the environment (supplied as an argument in line .1) is overwritten. Otherwise err is returned.

End of annotations

```
6.    EvalType : Type → TEval
6.1   EvalType(tp)(env) ≜
 .2    cases ShowTag(tp) :
 .3      'BasicType'          → EvalBasicType(tp)(env),
 .4      'CompositeType'      → EvalCompositeType(tp)(env),
 .5      'ProductType'        → EvalProductType(tp)(env),
 .6      'UnionType'          → EvalUnionType(tp)(env),
 .7      'OptionalType'       → EvalOptionalType(tp)(env),
 .8      'SetType'            → EvalSetType(tp)(env),
 .9      'Seq0Type'           → EvalSeq0Type(tp)(env),
 .10     'Seq1Type'           → EvalSeq1Type(tp)(env),
 .11     'GeneralMapType'     → EvalGeneralMapType(tp)(env),
 .12     'InjectiveMapType'   → EvalInjectiveMapType(tp)(env),
 .13     'FnType'             → EvalFnType(tp)(env),
 .14     'TypeId'             → EvalTypeId(tp)(env),
 .15     'TypeVar'            → EvalTypeVar(tp)(env),
 .16     'QuoteType'          → EvalQuoteType(tp)(env)
```

Annotations to *EvalType*:

6. This function gives meaning to syntactic types. The meaning is expressed by means of type evaluators (i.e. functions from *ENV* to $DOM \cup \{\underline{\text{err}}\}$).

.3–.16 The different kinds of type are evaluated by means of sub-functions.

End of annotations

```
7.    EvalBasicType : BasicType → TEval
7.1   EvalBasicType(MkTag('BasicType', tp))(env) ≜
 .2    cases tp :
 .3      NATONE  → natone-dom,
 .4      NAT     → nat-dom,
 .5      INTEGER → int-dom,
 .6      RAT     → rat-dom,
 .7      REAL    → real-dom,
 .8      BOOLEAN → Bool-dom,
 .9      CHAR    → char-dom,
 .10     TOKEN   → token-dom,
 .11     UNIT    → nil-dom
```

Annotations to *EvalBasicType*:

7. This function gives meaning to basic types.

.3–.11 The basic domains from the domain universe are used directly.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8. $EvalCompositeType : CompositeType \to TEval$

8.1 $EvalCompositeType(MkTag(\text{'CompositeType'}, (id, f_l)))(env) \triangleq$
.2 **let** $tp_l = ColTypeList(f_l)$ **in**
.3 **let** $d_l = EvalTypeList(tp_l)(env)$ **in**
.4 **if** $d_l = \underline{\text{err}}$
.5 **then** $\underline{\text{err}}$
.6 **else** $\mathcal{R}^{id}(d_l)$

Annotations to *EvalCompositeType*:

8. This function gives meaning to *CompositeTypes* which are tagged types with a list of fields (also called a record). Each of the fields have a type and possibly also a selector which can be used to select the corresponding part of the *CompositeType*.
.2 From the list of fields a list of the corresponding types is collected.
.3 The meaning of this list of types is found.
.4–.6 If an error is returned, an error is reported. Otherwise a record domain is constructed with the *id* tag and the list of domains.

End of annotations

9. $EvalProductType : ProductType \to TEval$

9.1 $EvalProductType(MkTag(\text{'ProductType'}, tp_l))(env) \triangleq$
.2 **let** $d_l = EvalTypeList(tp_l)(env)$ **in**
.3 **if** $d_l = \underline{\text{err}}$
.4 **then** $\underline{\text{err}}$
.5 **else** $\mathcal{P}(d_l)$

Annotations to *EvalProductType*:

9. This function gives meaning to *ProductTypes*, which are Cartesian products (interpreted as smashed products).
.2 The meaning of the list of types is found.
.3–.5 If an error is returned, an error is reported. Otherwise a product domain is constructed with the list of domains.

End of annotations

10. $EvalUnionType : UnionType \to TEval$

10.1 $EvalUnionType(MkTag(\text{'UnionType'}, t_s))(env) \triangleq$
.2 **let** $d_s = EvalTypeSet(t_s)(env)$ **in**
.3 **if** $d_s = \underline{\text{err}}$
.4 **then** $\underline{\text{err}}$
.5 **else let** $d = ((\bigcup\{|A| \mid A \in d_s\}, \bigcup\{\sqsubseteq_A \mid A \in d_s\}), \bigcup\{\|A\| \mid A \in d_s\})$ **in**
.6 **if** $IsCpo(d)$
.7 **then** d
.8 **else** $\underline{\text{err}}$

Annotations to *EvalUnionType*:

10. This function gives meaning to *UnionTypes*, which are types defined by a normal set-theoretic union of other types.
.2 The meaning of the set of types is found.
.3–.4 If an error is returned, an error is reported.
.5 A 'quasi'-domain is constructed. It corresponds to taking the union of the different domains. However, since these domains not necessarily are union-compatible it cannot be claimed that it is a real domain belonging to the domain universe.
.6–.8 If the 'quasi'-domain, d, forms a Cpo the domains have been union-compatible and d belongs to DOM.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Otherwise an error is reported.

End of annotations

11. *EvalOptionalType* : *OptionalType* → *TEval*

11.1 *EvalOptionalType*(*MkTag*('*OptionalType*', *t*))(*env*) ≜
.2 **let** *d* = *EvalType*(*t*)(*env*) **in**
.3 **if** *d* = err
.4 **then** err
.5 **else** $\mathcal{U}$(*d*, nil-dom)

Annotations to *EvalOptionalType*:

11. This function gives meaning to *OptionalTypes*.
.2 The meaning of the type, *t*, is found.
.3–.5 If an error was returned, an error is reported. Otherwise a union of the resulting domain and the basic domain containing nil is returned.

End of annotations

12. *EvalSetType* : *SetType* → *TEval*

12.1 *EvalSetType*(*MkTag*('*SetType*', *t*))(*env*) ≜
.2 **let** *d* = *EvalType*(*t*)(*env*) **in**
.3 **if** *d* = err ∨ *d* ∉ *FLATDOM*
.4 **then** err
.5 **else** $\mathcal{S}$(*d*)

Annotations to *EvalSetType*:

12. This function gives meaning to *SetTypes*, which are types of finite subsets.
.2 The meaning of the type, *t*, is found.
.3–.5 If an error is returned, an error is reported. This also happens if the meaning of *t* is a non-flat domain, *d*. Otherwise the domain of all finite subsets of *d* is returned.

End of annotations

13. *EvalSeq0Type* : *Seq0Type* → *TEval*

13.1 *EvalSeq0Type*(*MkTag*('*Seq0Type*', *t*))(*env*) ≜
.2 **let** *d* = *EvalType*(*t*)(*env*) **in**
.3 **if** *d* = err
.4 **then** err
.5 **else** $\mathcal{L}$(*d*)

Annotations to *EvalSeq0Type*:

13. This function gives meaning to *Seq0Types*, which are types of possibly empty sequences.
.2 The meaning of the type, *t*, is found.
.3–.5 If an error is returned, an error is reported. Otherwise the domain of all finite sequences of *d* is returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14. $EvalSeq1Type : Seq1Type \rightarrow TEval$

14.1 $EvalSeq1Type(MkTag(\text{'}Seq1Type\text{'}, t))(env) \triangleq$
.2 **let** $d = EvalType(MkTag(\text{'}Seq0Type\text{'}, t))(env)$ **in**
.3 **if** $d = \underline{\text{err}}$
.4 **then** $\underline{\text{err}}$
.5 **else** $\mathcal{I}(\{ s \in \|d\| \mid s \neq [\,] \}, d)$

Annotations to *EvalSeq1Type*:

14. This function gives meaning to *Seq1Type*s, which are types of non-empty sequences.
.2 The meaning of a type with possibly empty sequences of t, is found.
.3–.5 If an error is returned, an error is reported. Otherwise the domain of all non-empty finite sequences of d is returned by applying an invariant ensuring that the elements are non-empty.

End of annotations

15. $EvalGeneralMapType : GeneralMapType \rightarrow TEval$

15.1 $EvalGeneralMapType(MkTag(\text{'}GeneralMapType\text{'}, (t_1, t_2)))(env) \triangleq$
.2 **let** $d_1 = EvalType(t_1)(env)$,
.3 $d_2 = EvalType(t_2)(env)$ **in**
.4 **if** $d_1 = \underline{\text{err}} \vee d_2 = \underline{\text{err}} \vee d_1 \notin FLATDOM$
.5 **then** $\underline{\text{err}}$
.6 **else** $\mathcal{M}(d_1, d_2)$

Annotations to *EvalGeneralMapType*:

15. This function gives meaning to *GeneralMapType*s, which are types of finite mappings from a domain type to a range type.
.2–.3 The meaning of the domain type, t_1, and the range type, t_2, are found.
.4–.6 If an error is returned, an error is reported. Similarly, if the meaning of the domain type is a non-flat domain, an error is also reported. Otherwise a finite mapping domain with d_1 and d_2 is returned.

End of annotations

16. $EvalInjectiveMapType : InjectiveMapType \rightarrow TEval$

16.1 $EvalInjectiveMapType(MkTag(\text{'}InjectiveMapType\text{'}, (t_1, t_2)))(env) \triangleq$
.2 **let** $d = EvalType(MkTag(\text{'}GeneralMapType\text{'}, (t_1, t_2)))(env)$ **in**
.3 **if** $d = \underline{\text{err}}$
.4 **then** $\underline{\text{err}}$
.5 **else** $\mathcal{I}(\{ MkTag(\text{'}map\text{'}, m) \in \|d\| \mid Injective(m) \}, d)$

Annotations to *EvalInjectiveMapType*:

16. This function gives meaning to *InjectiveMapType*s which are types of finite mappings from domain type to a range type which have the property that they can be inverted (must be injective).
.2 The meaning of a general map type with t_1 and t_2 is found.
.3–.5 If an error is returned, an error is reported. Otherwise the domain of finite mappings from d_1 to d_2 with the injective property is returned by applying an invariant ensuring this property.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

17. $EvalFnType : FnType \to TEval$

17.1 $EvalFnType(MkTag(\text{'}FnType\text{'}, (t_1, t_2)))(env) \triangleq$
.2 **let** $d_1 = EvalType(t_1)(env)$,
.3 $d_2 = EvalType(t_2)(env)$ **in**
.4 **if** $d_1 = \underline{\text{err}} \vee d_2 = \underline{\text{err}}$
.5 **then** $\underline{\text{err}}$
.6 **else** $\mathcal{F}(d_1, d_2)$

Annotations to *EvalFnType*:
17. This function gives meaning to *FnTypes*, which are general (partial) function types.
.2–.3 The meaning of the domain type, t_1, and the range type, t_2 are found.
.4–.6 If an error is returned, an error is reported. Otherwise the domain of total functions from d_1 to d_2 is returned.

End of annotations

18. $EvalTypeId : TypeId \to TEval$

18.1 $EvalTypeId(MkTag(\text{'}TypeId\text{'}, id))(env) \triangleq$
.2 **if** $id \in \underline{\text{dom}}(env) \wedge env(id) \in DOM$
.3 **then** $env(id)$
.4 **else** $\underline{\text{err}}$

Annotations to *EvalTypeId*:
18. This function gives meaning to *TypeIds*.
.2–.4 If *id* belongs to the domain of the environment and the value looked up in the environment is a domain, then the value which has been looked up is returned. Otherwise an error is reported.

End of annotations

19. $EvalTypeVar : TypeVar \to TEval$

19.1 $EvalTypeVar(MkTag(\text{'}TypeVar\text{'}, id))(env) \triangleq$
.2 **if** $id \in \underline{\text{dom}}(env) \wedge env(id) \in DOM$
.3 **then** $env(id)$
.4 **else** $\underline{\text{err}}$

Annotations to *EvalTypeVar*:
19. This function gives meaning to *TypeVars*, which are type variables. These can only be introduced in the signature of polymorphic function definitions.
.2–.4 If *id* belongs to the domain of the environment and the value looked up in the environment is a domain, then the value which has been looked up is returned. Otherwise an error is reported.

End of annotations

20. $EvalQuoteType : QuoteType \to TEval$

20.1 $EvalQuoteType(MkTag(\text{'}QuoteType\text{'}, q))(env) \triangleq$
.2 $(\mathcal{T}_{CPO}^{\text{'}quot\text{'}}(\{q\}_\perp), \{MkTag(\text{'}quot\text{'}, q)\})$

Annotations to *EvalQuoteType*:
20. This function gives meaning to *QuoteTypes*, which can be viewed as singleton sets of quotations, i.e. strings of characters.
.2 The basic VDM domain with the given quotation, q, is returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
21.   EvalTypeList : 𝕃(Type) → ENV → 𝕃(DOM) ∪ {err}
21.1  EvalTypeList(t_l)(env) ≜
 .2    if t_l = [ ]
 .3    then [ ]
 .4    else let d   = EvalType(hd(t_l))(env),
 .5             d_l = EvalTypeList(tl(t_l))(env) in
 .6         if d = err ∨ d_l = err
 .7         then err
 .8         else join([ d], d_l)
```

Annotations to *EvalTypeList*:

21. This function gives meaning to a finite list of syntactic types.
.2–.3 If the list of types is empty, an empty list is returned.
.4–.5 The meaning of the first type in the list is found, and the meaning of the remaining list of types is found (recursively).
.6–.8 If either of these evaluations return an error, an error is reported. Otherwise the resulting list of domains is returned.

End of annotations

```
22.   EvalTypeSet : 𝔽(Type) → ENV → 𝔽(DOM) ∪ {err}
22.1  EvalTypeSet(t_s)(env) ≜
 .2    if t_s = { }
 .3    then { }
 .4    else let t = Choose(t_s) in
 .5         let d   = EvalType(t)(env),
 .6             d_s = EvalTypeSet(t_s\{ t})(env) in
 .7           if d = err ∨ d_s = err
 .8           then err
 .9           else { d} ∪ d_s
```

Annotations to *EvalTypeSet*:

22. This function gives meaning to a finite set of syntactic types.
.2–.3 If the set of types is empty, an empty set is returned.
.4 An arbitrary type, t, is chosen from the non-empty finite set, t_s.
.5–.6 The meaning of the chosen type is found, and the meaning of the set of the remaining types is found (recursively).
.7–.9 If either of these evaluations return an error, an error is reported. Otherwise the resulting set of domains is returned.

End of annotations

```
23.   EvalTypeMap : 𝔼(Type) → ENV → 𝔼(DOM) ∪ {err}
23.1  EvalTypeMap(t_m)(env) ≜
 .2    let id_s = dom(t_m) in
 .3      if ∃id ∈ id_s · EvalType(t_m(id))(env) = err
 .4      then err
 .5      else { id ↦ EvalType(t_m(id))(env) | id ∈ id_s}
```

Annotations to *EvalTypeMap*:

23. This function gives meaning to a finite map from identifiers to syntactic types.
.3–.4 If any of the type evaluators for the types in the map return an error, an error is reported.
.5 A map from identifiers to corresponding domains is created for all identifiers in the domain of the map.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2.2 State Definition

This clause presents the verification predicate for the state definition.

```
24.   VerifyStateDef : Pred((StateDef ∪ { nil }) × ENV)
24.1  VerifyStateDef(sd, env) ≙
 .2    if sd = nil
 .3    then GetStateTypeId(env) = nostate
 .4    else let MkTag('StateDef', (MkTag('ValueId', stid), tp, (id, expr))) = sd in
 .5           let initid = MkTag('InitId', stid),
 .6               st_d = GetStateDom(env),
 .7               btp = MkTag('BasicType', BOOLEAN),
 .8               d = EvalTypeId(tp)(env) in
 .9            if initid ∉ dom(env) ∨ d = err
 .10           then F
 .11           else let init_s = MakeFuncAbs(id, tp, btp, expr)(env) in
 .12                    st_d = d ∧ env(initid) ∈ init_s ∧ ShowTag(env(initid)) = 'fun' ∧
 .13                    (∀e ∈ ||d|| · StripTag(env(initid))(e) ∈ BOOL_VAL) ∧
 .14                    (∃e ∈ ||d|| · StripTag(env(initid))(e) = True())
```

Annotations to *VerifyStateDef*:

24. This predicate checks whether the given environment, *env*, contains a proper denotation for the syntactic state definition.

.2–.3 If there is no state present *GetStateTypeId* must yield nostate.

.4–.14 The state definition is decomposed into its components and it is required that looking up the state type, *st_d*, in the environment is equal to the result of evaluating the syntactic definition of the type of the state. In addition, it is required that the initialisation predicate is member of possible initial values from abstracting it into a truth-valued function. Notice how it is required in .13 that the initialisation predicate is defined for all values in the state domain and how line .14 express that at least one possible state value must exist initially.

End of annotations

8.2.3 Value Definitions

This clause presents the verification predicate for value definitions and the evaluation function which is used for both value definitions and explicitly defined functions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2.3.1 The Verification Predicate for Value Definitions

```
25.    VerifyValues : Pred(𝕄(Pattern, ValDef) × ENV)
25.1   VerifyValues(val_m, env) ≙
  .2    ∀pat ∈ dom(val_m) ·
  .3     let id_s = ColIdSet(pat) in
  .4      if id_s ⊆ dom(env) ∧ ∀id ∈ id_s · env(id) ∈ VAL
  .5      then let ldef = EvalExplDef(val_m) in
  .6             let env_s = { def(subtract(env, id_s)) | def ∈ ldef} in
  .7              ∃env' ∈ env_s ·
  .8               ∀id ∈ id_s ·
  .9                env(id) = env'(id) ∧
  .10               let MkTag('ValDef', (tp, _)) = env(id) in
  .11                if tp ≠ nil
  .12                then let d = EvalType(tp)(env) in
  .13                       d ≠ err ∧ env(id) ∈ ||d|| ∧ CheckTagEnv(env, tp)
  .14                else env(id) ≠ ⊥
  .15     else F
```

Annotations to *VerifyValues*:

25. This predicate checks whether the given environment contains proper denotations for the syntactic value definitions.

.3 The right hand side of value definitions can be general patterns and it is therefore necessary to collect all of the pattern identifiers which are globally declared by the value definitions.

.4 The names of all of the pattern identifiers must belong to the domain of the environment, and all of them must be ordinary values from the domain universe.

.5–.6 The collection of value definitions is evaluated in the given environment subtracted by the denotations of these value definitions. Because of looseness this results in a set of new environments.

.7–.14 There must exist an environment (resulting from the evaluation of the value definitions) such that the given environment provides satisfactory denotations to all pattern identifiers. This also includes a type-check if a type is present in the value definition. Furthermore, it is checked that the type does not contain any undefined composite types (this is done by *CheckTagEnv*).

End of annotations

An example combining looseness and recursion

Before proceeding with the semantics of recursive explicit value definitions, a small example is shown, illustrating the complexity which needs to be taken into account here. The aspect which this example emphasizes is how the fixed point calculation is carried out for a recursive definition which includes looseness (and thus has multiple least fixed points). The concrete syntax for this example is a variation of the traditional factorial function, and it looks like:

```
fac1 : ℕ → ℕ
fac1 (n) ≙
  if n = 0
  then let x ∈ {1, 2} in x
  else n × fac1(n − 1)
```

This is an explicit function definition using looseness in the body of the function. Such looseness is interpreted as underdeterminedness, implying that for each model of the function the same choice is made each time (thus ensuring referential transparency).

The explicit function definition above is transformed into a core abstract syntax form where the body component is a lambda expression corresponding to this concrete syntax:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$\lambda\, n : \mathbb{N} \cdot$
  if $n = 0$
  then let $x \in \{1, 2\}$ in x
  else $n \times fac_1(n - 1)$

When this syntactic expression is evaluated it will have a set of two expression evaluators as its denotation (using *EvalExpr*):

$$\{\, \lambda env\ .\ \lambda n\ .\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times fac_1(n - 1),\quad \lambda env\ .\ \lambda n\ .\ \textbf{if}\ n = 0\ \textbf{then}\ 2\ \textbf{else}\ n \times fac_1(n - 1)\,\}$$

For each of these two expression evaluators the least fixed point is calculated. The first calculation proceeds in the following way:

$$\begin{aligned}
&(1)\quad fac_1 = \lambda n.\ \bot\\
&(2)\quad fac_1 = \lambda n.\ \begin{cases} n = 0 \Rightarrow 1\\ n > 0 \Rightarrow \bot \end{cases}\\
&(3)\quad fac_1 = \lambda n.\ \begin{cases} n = 0 \Rightarrow 1\\ n = 1 \Rightarrow 1\\ n > 1 \Rightarrow \bot \end{cases}\\
&(4)\quad fac_1 = \lambda n.\ \begin{cases} n = 0 \Rightarrow 1\\ n = 1 \Rightarrow 1\\ n = 2 \Rightarrow 2\\ n > 2 \Rightarrow \bot \end{cases}\\
&(5)\quad fac_1 = \lambda n.\ \begin{cases} n = 0 \Rightarrow 1\\ n = 1 \Rightarrow 1\\ n = 2 \Rightarrow 2\\ n = 3 \Rightarrow 6\\ n > 3 \Rightarrow \bot \end{cases}\\
&\qquad\vdots
\end{aligned}$$

which corresponds to the unfolding of the normal factorial function. The second calculation proceeds in the following way:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$
\begin{array}{lll}
(1) & fac_1 & = \lambda n.\ \bot \\
(2) & fac_1 & = \lambda n. \left\{ \begin{array}{lll} n = 0 & \Rightarrow & 2 \\ n > 0 & \Rightarrow & \bot \end{array} \right. \\
(3) & fac_1 & = \lambda n. \left\{ \begin{array}{lll} n = 0 & \Rightarrow & 2 \\ n = 1 & \Rightarrow & 2 \\ n > 1 & \Rightarrow & \bot \end{array} \right. \\
(4) & fac_1 & = \lambda n. \left\{ \begin{array}{lll} n = 0 & \Rightarrow & 2 \\ n = 1 & \Rightarrow & 2 \\ n = 2 & \Rightarrow & 4 \\ n > 2 & \Rightarrow & \bot \end{array} \right. \\
(5) & fac_1 & = \lambda n. \left\{ \begin{array}{lll} n = 0 & \Rightarrow & 2 \\ n = 1 & \Rightarrow & 2 \\ n = 2 & \Rightarrow & 4 \\ n = 3 & \Rightarrow & 12 \\ n > 3 & \Rightarrow & \bot \end{array} \right. \\
 & \vdots &
\end{array}
$$

which corresponds to the unfolding of 2 times the normal factorial function. Thus the example used here has two least fixed points (one for each of the expression evaluators), therefore it illustrates how a least fixed point semantics can be dealt with in the presence of looseness.

□

A more complicated example

The example above introduced finite underdeterminedness, but infinite underdeterminedness can also be introduced. Consider:

$fac_2 : \mathbb{N} \to \mathbb{N}$

$fac_2\,(n) \triangleq$
  let $x \in \{1, 2\}$ in
  if $n = 0$
  then x
  else $x \times n \times fac_2\,(n - 1)$

Notice how the choice of x can be made on each recursive call of fac_2. Therefore, the choice of x depends on the actual value of the formal parameter n (i.e. $fac_2(2)$ may chose to bind x to 1 while $fac_2(1)$ may chose to bind x to 2 in the same model).

Just like before the explicit function definition is transformed into a core abstract syntax form. Here the corresponding concrete syntax will look like:

$\lambda\, n : \mathbb{N} \cdot$
  let $x \in \{1, 2\}$ in if $n = 0$
  then x
  else $x \times n \times fac_2(n - 1)$

When this syntactic expression is evaluated it will have an infinite set of expression evaluators of the form:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$\{\lambda env\ .\ \lambda n\ .$ **if** $n = 0$
then 1
else $n \times fac_2(n - 1)$,
$\lambda env\ .\ \lambda n\ .$ **if** $n = 0$
then 2
else $n \times fac_2(n - 1)\}$
$\lambda env\ .\ \lambda n\ .$ **if** $n = 0$
then 2
else if $n = 1$
then 4
else $n \times fac_2(n - 1)$,
...
$\lambda env\ .\ \lambda n\ .$ **if** $n = 0$
then 1
else $2 \times n \times fac_2(n - 1)\}$
$\lambda env\ .\ \lambda n\ .$ **if** $n = 0$
then 2
else $2 \times n \times fac_2(n - 1)\}$

This infinite set contains all functions for which the least fixed points can be described by[1]: $\lambda\, n \cdot 2^{n+1-k} \times fac(n)$ where *fac* is the normal factorial function and $0 \leq k \leq n + 1$. The first evaluator listed chooses 1 every time, whereas the second evaluator chooses 2 for $n = 0$ and 1 otherwise. The third evaluator chooses 2 for $n = 0$ and $n = 1$ and 1 otherwise and the last evaluators which are written in the set are ones where 2 is chosen (almost) all the time. Just like before the least fixed point will be found for each of these evaluators in the standard way.

In general there will be many evaluators which have the same upper bound. In this particular case each evaluator which chooses 2 once and 1 all other times will have the upper bound corresponding to twice the normal factorial function. However, notice that even though these have the same least upper bound they are incomparable least fixed points (they will have different graphs because of the different places where 2 was chosen). Here it means that $fac_2(2)$ may yield 2 (2!), 4 (2×2!), 8 (2×2×2!) and 16 (2×2×2×2!) in the different models for fac_2.

□

8.2.3.2 The Evaluation Function for Value Definitions

The function presented in this clause is made so general that monomorphic function definitions are also handled as a kind of value definitions.

[1] Note that the value of k may depend on n for each of the functions. Thus we really have an infinite set of functions despite the relatively "simple" expression used here to express it.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

26. $EvalExplDef : \mathbb{M}(Pattern,\ (ExplFnDef \cup ValDef)) \to LDef$

26.1 $EvalExplDef(def_m) \triangleq$
.2 **let** $p_s = \underline{\mathrm{dom}}(def_m)$ **in**
.3 **let** $lev_m = \{\, p \mapsto EvalExpr(SelBody(def_m(p))) \mid p \in p_s \,\}$ **in**
.4 **let** $ev_m_s = \{\, m \mid m \in \mathbb{M}(Pattern,\ EEval) \cdot$
.5 $\underline{\mathrm{dom}}(m) = p_s \wedge$
.6 $\forall p \in p_s \cdot m(p) \in lev_m(p) \,\}$ **in**
.7 **let** $plev_m = \{\, p \mapsto EvalPattern(p) \mid p \in p_s \,\}$ **in**
.8 **let** $pev_m_s = \{\, m \mid m \in \mathbb{M}(Pattern,\ PatEval) \cdot$
.9 $\underline{\mathrm{dom}}(m) = p_s \wedge$
.10 $\forall p \in p_s \cdot m(p) \in plev_m(p) \,\}$ **in**
.11 $\{\, \lambda env.$**let** $tf = \lambda appr \in ObjApprox_{\underline{\mathrm{err}}}$.
.12 **if** $appr = \underline{\mathrm{err}}$
.13 **then** $\underline{\mathrm{err}}$
.14 **else let** $venv_s = \{$ **let** $pev = pev_m(p),$
.15 $ev = ev_m(p),$
.16 $env' = \underline{\mathrm{overwrite}}(env,\ appr)$ **in**
.17 $pev(ev(env'))(env)$
.18 $\mid p \in p_s \,\}$ **in**
.19 **if** $\{\underline{\mathrm{err}},\ \underline{\mathrm{unmatch}}\} \subseteq venv_s \vee \neg Compatible(venv_s)$
.20 **then** $\underline{\mathrm{err}}$
.21 **else** $\underline{\mathrm{Merge}}(venv_s)$ **in**
.22 **if** $IsCont_{ObjApprox_{\underline{\mathrm{err}}}}(tf)$
.23 **then let** $lfp_m = \mathrm{Y}(tf)$ **in**
.24 **if** $lfp_m = \underline{\mathrm{err}}$
.25 **then** $\underline{\mathrm{err}}$
.26 **else** $\underline{\mathrm{overwrite}}(env,\ lfp_m)$
.27 **else** $\underline{\mathrm{err}}$
.28 $\mid pev_m \in pev_m_s,\ ev_m \in ev_m_s \,\}$

Annotations to *EvalExplDef*:

26. The function *EvalExplDef* gives meaning to a set of possibly mutually recursive explicit function and value definitions.

.11–.28 The set of definers (i.e. a loose definer) is constructed.

.11–.21 A function $tf : ObjApprox_{\underline{\mathrm{err}}} \to ObjApprox_{\underline{\mathrm{err}}}$ is defined. The set, $ObjApprox_{\underline{\mathrm{err}}}$, is a union $ObjApprox \cup \{\underline{\mathrm{err}}\}$, where $ObjApprox \subseteq DefId \to VAL$ and $DefId = \bigcup\{\, ColIdSet(p) \mid p \in p_s \,\}$ (see .2 of *EvalLocalDef*). The set, *ObjApprox*, has the following property:
$\forall id \in DefId \cdot \exists D_{id} \in DOM \cdot$
$\forall f \in ObjApprox \cdot f(id) \in |D_{id}|$.
The set $ObjApprox_{\underline{\mathrm{err}}}$ has a structure of a cpo with an ordering $\sqsubseteq$ defined by:

— for $f,g \in ObjApprox\ f \sqsubseteq g$ iff $\forall id \in DefId \cdot f(id) \sqsubseteq_{D_{id}} g(id)$
— $\bot \sqsubseteq \underline{\mathrm{err}}$

The function $\lambda id : DefId$. $\bot$ is a bottom element. Note that $\underline{\mathrm{err}}$ is uncomparable with any element of *ObjApprox* except bottom.

.22–.27 If *tf* is continuous with respect to the ordering on $ObjApprox_{\underline{\mathrm{err}}}$, then the least fixed point of *tf* is calculated. If it is different from $\underline{\mathrm{err}}$, then the current environment is overwritten. Otherwise an error is reported.

.28 The construction is indexed by all possible interpretations of patterns and expressions occurring in the definitions being elaborated.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2.4 Function Definitions

This clause presents all verification predicates for the different kind of function definitions. In addition the evaluation functions for polymorphic explicitly defined functions are given. The evaluation function for monomorphic explicitly defined functions can be found in the clause about value definitions.

8.2.4.1 Verification Predicates for Function Definitions

```
27.   VerifyExplPolyFns : Pred(𝔼(ExplPolyFnDef) × ENV)
27.1  VerifyExplPolyFns(epfd_m, env) ≜
 .2    let id_s = dom(epfd_m) in
 .3      if id_s ⊆ dom(env) ∧ ∀id ∈ id_s · env(id) ∈ POLYVAL
 .4      then let ldef = EvalPolyDef(epfd_m) in
 .5             let env_s = { def(subtract(env, id_s)) | def ∈ ldef}\{ err },
 .6                 p_lev_f = λid ∈ id_s.EvalExpr(SelPre(epfd_m(id))) in
 .7               let p_ev_f_s = PropE({ p_lev_f}) in
 .8                 let tp_l_m = { id ↦ ColTypeList(ColParList(epfd_m(id))) | id ∈ id_s },
 .9                     fn_d_m = { id ↦ EvalType(GetType(epfd_m(id))) | id ∈ id_s },
 .10                    id_l_m = { id ↦ ColIdList(ColParList(epfd_m(id))) | id ∈ id_s } in
 .11                ∃env' ∈ env_s ·
 .12                  ∃p_ev_f ∈ p_ev_f_s ·
 .13                    ∀id ∈ id_s ·
 .14                      ∀d_l ∈ 𝕃(DOM) ·
 .15                        len(d_l) = len(ColTypeVars(epfd_m(id))) ⇒
 .16                        (∀i ∈ inds(tp_l_m(id)) · CheckTagEnv(tp_l_m(id)(i))) ∧
 .17                        let env'' = ExtDomEnv(SelTypeVars(epfd_m(id)), d_l) in
 .18                          if env'' ≠ err
 .19                          then let d_m = { id ↦ EvalType(GetType(epfd_m(id)))
 .20                                         | id ∈ ids },
 .21                                   d_l_m = { id ↦ EvalTypeList(tp_l_m(id))
 .22                                                  (overwrite(env, env''))
 .23                                           | id ∈ id_s } in
 .24                               if env'(id) ∈ POLYVAL ∧
 .25                                 env'(id)(d_l) ∈ FUN_VAL ∧
 .26                                 d_m(id) ≠ err ∧env(id)(d_l) ∈ ||d_m(id)||
 .27                               then let id_l = id_l_m(id),
 .28                                        d_l' = d_l_m(id),
 .29                                        ev = p_ev_f(id),
 .30                                        kind = SelKind(epfd_m(id)),
 .31                                        f1 = env'(id)(d_l),
 .32                                        f2 = env(id)(d_l) in
 .33                                    VerifyCurFn(id_l, d_l', f1, f2, ev, kind, env)
 .34                               else F
 .35                          else F
 .36     else F
```

Annotations to *VerifyExplPolyFns*:

27. This predicate checks whether the given environment, *env*, contains proper denotations for the syntactic explicit polymorphic function definitions.

.3 All names of the polymorphic function definitions must belong to the domain of the environment and all of them must be bound to polymorphic values in the environment.

.4–.5 The collection of polymorphic function definitions is evaluated in the given environment where the denotation of these definitions have been removed. Because of looseness this results in a set of new environments, *env_s*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.6–.7 A collection of evaluators for the pre-conditions to the different polymorphic function definitions is gathered in *p_ev_f_s*.

.11–.33 There must exist an environment (resulting from the evaluation of the polymorphic function definitions) and there must exist a combination of the pre-conditions such that the given environment provides satisfactory denotations to all the polymorphic function definitions (they must be tested with all possible legal instantiations).

.16 All composite types used in the signature of all of the functions must be defined in the type definition part of the specification.

.27–.33 Given that the function value from the environment belongs to the signature type of each Curried function and the evaluation of the body results in a function value further verification is carried out by the auxiliary predicate *VerifyCurFn*.

End of annotations

28. $VerifyExplFns : \mathbf{Pred}(\mathbb{E}(ExplFnDef) \times ENV)$

28.1 $VerifyExplFns(efd_m,\ env) \triangleq$

.2 **let** $id_s = \underline{\mathrm{dom}}(efd_m)$ **in**

.3 **if** $id_s \subseteq \underline{\mathrm{dom}}(env) \wedge \forall id \in id_s \cdot env(id) \in FUN_VAL$

.4 **then let** $ldef = EvalExplDef(\{\ MkTag(\text{'PatternId'},\ id) \mapsto efd_m(id)$

.5 $\mid id \in id_s\ \})$ **in**

.6 **let** $env_s = \{\ def(\underline{\mathrm{subtract}}(env,\ id_s)) \mid def \in ldef\} \backslash \{\ \underline{\mathrm{err}}\ \},$

.7 $p_lev_f = \lambda id \in id_s.EvalExpr(SelPre(efd_m(id)))$ **in**

.8 **let** $p_ev_f_s = PropE(\{\ p_lev_f\ \})$ **in**

.9 **let** $tp_l_m = \{\ id \mapsto ColTypeList(ColParList(efd_m(id))) \mid id \in id_s\},$

.10 $fn_d_m = \{\ id \mapsto EvalType(GetType(efd_m(id))) \mid id \in id_s\},$

.11 $id_l_m = \{\ id \mapsto ColIdList(ColParList(efd_m(id))) \mid id \in id_s\}$ **in**

.12 **if** $\forall id \in id_s \cdot EvalTypeList(tp_l_m(id))(env) \neq \underline{\mathrm{err}} \wedge fn_d_m(id) \neq \underline{\mathrm{err}}$

.13 **then let** $d_l_m = \{\ id \mapsto EvalTypeList(tp_l_m(id))(env)$

.14 $\mid id \in id_s\}$ **in**

.15 $\exists env' \in env_s \cdot$

.16 $\exists p_ev_f \in p_ev_f_s \cdot$

.17 $\forall id \in id_s \cdot$

.18 $(\forall i \in \underline{\mathrm{inds}}(tp_l_m(id)) \cdot CheckTagEnv(tp_l_m(id)(i))) \wedge$

.19 **if** $env(id) \in \|fn_d_m(id)(env)\| \wedge env'(id) \in FUN_VAL$

.20 **then let** $id_l = id_l_m(id),$

.21 $d_l = d_l_m(id),$

.22 $ev = p_ev_f(id),$

.23 $kind = SelKind(efd_m(id)),$

.24 $f_1 = env'(id),$

.25 $f_2 = env(id)$ **in**

.26 $VerifyCurFn(id_l,\ d_l,\ f_1,\ f_2,\ ev,\ kind,\ env)$

.27 **else** F

.28 **else** F

.29 **else** F

Annotations to *VerifyExplFns*:

28. This predicate checks whether the given environment, *env*, contains proper denotations for the syntactic explicit (monomorphic) function definitions.

.3 All names of the function definitions must belong to the domain of the environment and all of them must be bound to monomorphic function values in the environment.

.4–.6 The collection of monomorphic function definitions is evaluated in the given environment subtracted by the denotation of these definitions. Because of looseness this results in a set of new environments, *env_s*.

.7–.8 A collection of evaluators for the pre-conditions of the different polymorphic function definitions is gathered in *p_ev_f_s*.

.15–.26 There must exist an environment (resulting from the evaluation of the monomorphic function definitions) and there must exist a combination of the pre-conditions such that the given environment provides

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

satisfactory denotations to all the monomorphic function definitions.

.18 All composite types used in the signature of all of the functions must be defined in the type definition part of the specification.

.19–.26 Given that the function value from the environment belongs to the signature type of each Curried function and the evaluation of the body resulted in a function value further verification is carried out by the auxiliary predicate *VerifyCurFn*.

End of annotations

```
29.   VerifyImplPolyFns : Pred(𝔼(ImplPolyFnDef) × ENV)
29.1  VerifyImplPolyFns(ipfd_m, env) ≜
 .2    let id_s = dom(ipfd_m) in
 .3      if id_s ⊆ dom(env) ∧ ∀id ∈ id_s · env(id) ∈ POLYVAL
 .4      then let pre_ev_m_s  = MakeEvalMapSet(ipfd_m)(λ(id, ipfd). SelPre(ipfd)),
 .5               post_ev_m_s = MakeEvalMapSet(ipfd_m)(λ(id, ipfd). SelPost(ipfd)),
 .6               pfn_m_s     = MakePolyFns(ipfd_m)(env) in
 .7             pfn_m_s ≠ err ∧
 .8             ∃pre_ev_m ∈ pre_ev_m_s ·
 .9               ∃post_ev_m ∈ post_ev_m_s ·
 .10                ∃pfn_m ∈ pfn_m_s ·
 .11                  ∀id ∈ id_s ·
 .12                    ∀d_l ∈ 𝕃(DOM) ·
 .13                      len(d_l) = len(ColTypeVars(ipfd_m(id))) ⇒
 .14                      env(id) = pfn_m(id) ∧
 .15                      ∀ext_m ∈ ExtImplPolyFnEnv(ipfd_m(id))(env)(d_l) ·
 .16                        ext_m ≠ err ∧
 .17                        let env' = overwrite(env, ext_m),
 .18                            ext = { SelRes(ipfd_m(id)) ↦
 .19                                    GenPolyApply(env(id), d_l, Iota(rng(ext_m))) } in
 .20                          let ext_env = overwrite(env', ext) in
 .21                            let pre  = pre_ev_m(id),
 .22                                post = post_ev_m(id) in
 .23                              if pre ∉ POLYVAL ∨ post ∉ POLYVAL
 .24                              then F
 .25                              else (pre(d_l)(env') = False()) ∨
 .26                                   (pre(d_l)(env') = True() ∧
 .27                                    post(d_l)(ext_env) = True())
 .28     else F
```

Annotations to *VerifyImplPolyFns*:

29. This predicate checks whether the given environment, *env*, contains proper denotations for the syntactic implicit polymorphic function definitions.

.3 All names of the polymorphic function definitions must belong to the domain of the environment and all of them must be bound to polymorphic values in the environment.

.4–.5 Two sets of maps of all possible combinations of expression evaluators for the pre and post-condition expressions are created. *pre_ev_m_s* and *post_ev_m_s* both have the type ℙ(𝔼(*EEval*)).

.6–.7 Given the signature of each implicit polymorphic function definition, all possible polymorphic function values are collected. An error is reported if an undefined composite type has been used in the signature.

.8–.27 There must exist a map of pre-conditions and a map of post-conditions such that the given environment provides satisfactory denotations to all of the polymorphic function definitions (they must be tested with all possible legal instantiations). Furthermore there must exist a combination of polymorphic function values such that all function names are bound to their corresponding value in the given environment.

.14–.20 The environment for evaluating the pre and post-conditions must be expanded with a binding of the 'argument identifier' to a possible argument value (this extension must be carried out for possible values in the domain of the function). In addition, the environment used for evaluating the post-condition must be extended with a binding of the 'result identifier' to the result of applying the function (this extension

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

must be carried out for possible values in the range of the function).

.21–.23 Each pre and post-condition pair in the maps must be bound to polymorphic function values.

.25–.27 Either the pre-condition is false and nothing is guaranteed about the post-condition, or both the pre-condition and the post-condition must be true.

End of annotations

30. $VerifyImplFns : \mathbf{Pred}(\mathbb{E}(ImplFnDef) \times ENV)$

30.1 $VerifyImplFns(ifd_m, env) \triangleq$

.2 **let** $id_s = \underline{\mathrm{dom}}(ifd_m)$ **in**

.3 **if** $id_s \subseteq \underline{\mathrm{dom}}(env) \wedge \forall id \in id_s \cdot env(id) \in FUN_VAL$

.4 **then let** $pre_ev_m_s = MakeEvalMapSet(ifd_m)(\lambda(id, ifd).\ SelPre(ifd))$,

.5 $post_ev_m_s = MakeEvalMapSet(ifd_m)(\lambda(id, ifd).\ SelPost(ifd))$,

.6 $app_ev_m_s = MakeEvalMapSet(ifd_m)(\lambda(id, ifd).\ GetFnApp(id, ifd))$,

.7 $ext_m_s = ExtImplFnsEnv(ifd_m)(env)$,

.8 $fn_m_s = MakeFns(ifd_m)(env)$ **in**

.9 $fn_m_s \neq \underline{\mathrm{err}} \wedge$

.10 $\exists pre_ev_m \in pre_ev_m_s \cdot$

.11 $\exists post_ev_m \in post_ev_m_s \cdot$

.12 $\exists app_ev_m \in app_ev_m_s \cdot$

.13 $\exists fn_m \in fn_m_s \cdot$

.14 $\forall ext_m \in ext_m_s \cdot$

.15 $\forall id \in id_s \cdot$

.16 $env(id) = fn_m(id) \wedge$

.17 **let** $env' = \underline{\mathrm{overwrite}}(env, ext_m(id))$ **in**

.18 **let** $ext = \{ SelRes(ifd_m(id)) \mapsto app_ev_m(id)(env') \}$ **in**

.19 **let** $ext_env = \underline{\mathrm{overwrite}}(env', ext)$ **in**

.20 $(pre_ev_m(id)(env') = False()) \vee$

.21 $(pre_ev_m(id)(env') = True() \wedge$

.22 $post_ev_m(id)(ext_env) = True())$

.23 **else** F

Annotations to *VerifyImplFns*:

30. This predicate checks whether the given environment contains proper denotations for the syntactic implicit function definitions.

.3 All names of the implicit function definitions must belong to the domain of the given environment, *env*, and all of them must be bound to (monomorphic) function values in the environment.

.4–.5 Two sets of maps of all possible combinations of expression evaluators for the pre and post-condition expressions are created. *pre_ev_m_s* and *post_ev_m_s* both have the type $\mathbb{P}(\mathbb{E}(EEval))$.

.6 This constructs a singleton set of a map from the names of the implicitly defined functions to the expression evaluator arising from applying the corresponding function with an actual parameter identical to the formal parameter.

.7 The environment for evaluating the pre and post-conditions must be expanded with a binding of the 'argument identifier' to a possible argument value. A map is created of all possible extensions for all possible values in the domain of the function.

.8 All implicitly defined (monomorphic) functions must belong to the type used in the signature of the function definition. *fn_m_s* is a set of all possible maps of (total) function values fulfilling the type of the function.

.9 If an error is reported from using the function type the entire model can be rejected.

.10–.22 A map of pre-conditions and a map of post-conditions must exist such that the given environment provides satisfactory denotations for all of the implicit function definitions for all possible argument values. It is also checked that the type signatures for the functions are satisfied.

.18–.19 The environment used for evaluating the post-condition must be extended with a binding of the 'result identifier' to the result of applying the function (this extension must be carried out for possible values in the range of the function).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.20 – .22 Either the pre-condition is false and nothing is guaranteed about the post-condition, or both the pre-condition and the post-condition must be true.

End of annotations

31. $VerifyCurFn$: **Pred**($\mathbb{L}(Id) \times (\mathbb{L}(DOM) \cup \{\underline{\text{err}}\}) \times FUN_VAL \times FUN_VAL \times EEval \times \{\text{TOTAL, PARTIAL}\} \times ENV$)

31.1 $VerifyCurFn(id_l,\ d_l,\ f_1,\ f_2,\ ev,\ kind,\ env) \triangleq$
.2 **if** $d_l \neq \underline{\text{err}} \wedge EqCurDoms(\underline{\text{len}}(d_l),\ f_1,\ f_2)$
.3 **then let** $v_l_s = \{\, l \mid l \in \mathbb{L}(VAL) \cdot \underline{\text{len}}(l) = \underline{\text{len}}(d_l) \wedge$
.4 $\forall i \in \underline{\text{inds}}(l) \cdot l(i) \in \|d_l(i)\|\,\}$ **in**
.5 $\forall v_l \in v_l_s \cdot$ **let** $env' = ExtValEnv(id_l,\ v_l)$ **in**
.6 $env' \neq \underline{\text{err}} \wedge$
.7 $ev(\underline{\text{overwrite}}(env,\ env')) = True() \Rightarrow$
.8 $((kind = \text{PARTIAL} \vee ApplyCurFn(f_1,\ v_l) \neq \bot) \wedge$
.9 $ApplyCurFn(f_1,\ v_l) = ApplyCurFn(f_2,\ v_l))$
.10 **else** F

Annotations to *VerifyCurFn*:

31. This predicate checks whether the given environment contains proper denotations for a Curried explicit function.
.2 No error must be reported in the evaluation of the list of Curried domains, and the two functions, f_1 and f_2, must be equally Curried (with similar domains).
.3 – .4 A set of all possible lists of (Curried) argument values is created.
.5 – .9 For all such lists of possible (Curried) argument values it must hold that if the pre-condition is fulfilled the two functions f_1 and f_2 must yield the same value, and if it is a total function the result must be different from $\bot$.
.7 It is checked that the pre-condition is fulfilled. The pre-condition evaluator, ev, must yield true for the given environment extended with all the argument bindings.
.8 If the function is total it must not yield $\bot$ if the pre-condition is fulfilled.

End of annotations

8.2.4.2 Evaluation Functions for Polymorphic Functions

32. $EvalPolyDef$: $\mathbb{E}(ExplPolyFnDef) \rightarrow LDef$

32.1 $EvalPolyDef(poly_m) \triangleq$
.2 **let** $id_s = \underline{\text{dom}}(poly_m)$ **in**
.3 **let** $lpolyev_m = \{\, id \mapsto MakePolyFnDef(SelPolyFn(poly_m(id))) \mid id \in id_s\,\}$ **in**
.4 **let** $polyev_m_s = \{\, m \in \mathbb{E}(PEval)$
.5 $\mid \underline{\text{dom}}(m) = id_s \cdot \forall id \in id_s \cdot m(id) \in lpolyev_m(id)\,\}$ **in**
.6 $\{\, \lambda env.$**let** $id_l_m = \{\, id \mapsto ColTypeVars(poly_m(id)) \mid id \in id_s\,\},$
.7 $tp_m = \{\, id \mapsto GetType(poly_m(id)) \mid id \in id_s\,\}$ **in**
.8 **let** $pfun = \lambda id \in id_s\ .$
.9 $PolyVals(EvalPolyType(id_l,\ tp_m(id))(env))$ **in**
.10 **let** $PolyApprox = PropErr(\{\, pfun\,\})$ **in**
.11 **let** $tf = \lambda app \in PolyApprox\ .$
.12 $\{\, id \mapsto polyev_m(id)(\underline{\text{overwrite}}(env,\ app)) \mid id \in id_s\,\}$ **in**
.13 **if** $IsCont_{PolyApprox}(tf)$
.14 **then let** $lfp_m = \Upsilon(tf)$ **in**
.15 $\underline{\text{overwrite}}(env,\ lfp_m)$
.16 **else** $\underline{\text{err}}$
.17 $\mid polyev_m \in polyev_m_s\,\}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annotations to *EvalPolyDef*:

32. The function *EvalPolyDef* gives meaning to a set of possibly mutually recursive polymorphic function definitions.

.6–.17 The set of definers (i.e. a loose definer) is constructed.

.10 For a given environment *env*, the set of approximations $PolyApprox \subseteq id_s \to POLYVAL$ is constructed. It has a structure of a cpo with an ordering defined by:

$$f \sqsubseteq g \text{ iff } \forall id \in id_s \cdot \forall d_l \in \mathbb{L}_1(DOM) \cdot f(id)(d_l) \sqsubseteq_{D_{id}^{d_l}} g(id)(d_l)$$

where $D_{id}^{d_l} = EvalPolyType(poly_m(id))(env)(d_l)$.

.11–.12 A function $tf : PolyApprox \to PolyApprox$ is defined.

.13–.16 If *tf* is continuous with respect to the ordering in *PolyApprox*, then the current environment is overwritten by the least fixed point of *tf*. Otherwise an error is reported.

.17 The whole construction is indexed by all possible mappings of polymorphic evaluators corresponding to the definitions.

End of annotations

33. $EvalPolyType : \mathbb{L}_1(Id) \times Type \to PDomEval$

33.1 $EvalPolyType(id_l,\ t) \triangleq$

.2 $\lambda env\ .\ \lambda d_l\ .$ **if** $(\underline{\text{len}}(d_l) \neq \underline{\text{len}}(id_l)) \vee (\underline{\text{len}}(id_l) \neq \underline{\text{card}}(\underline{\text{elems}}(id_l)))$

.3 **then** $(\emptyset_\perp,\ \{\ \})$

.4 **else let** $newenv = \underline{\text{overwrite}}(env,\ ExtDomEnv(id_l,\ d_l))$ **in**

.5 **let** $d = EvalType(t)(\underline{\text{overwrite}}(env,\ newenv))$ **in**

.6 **if** $d = \underline{\text{err}}$

.7 **then** $(\emptyset_\perp,\ \{\ \})$

.8 **else** d

Annotations to *EvalPolyType*:

33. This function takes a non-empty list of identifiers (the names of the type variables use the type expression of a polymorphic function) and a type expression (using these identifiers), and returns the polymorphic domain evaluator which correspond to the polymorphic domain (a mapping from the argument identifiers to the domain from the argument type expression).

.2–.3 If either the length of the domain list, *d_l*, is different from the length of the identifier list, *id_l*, or any of the identifiers are duplicated the bottom domain is returned.

.4 Otherwise it is possible to extend the given environment, *env*, with bindings of each identifier from *id_l* to the domain with the same index from *d_l*.

.5–.7 If the meaning of the type, *t*, in this extended environment returns an error, the bottom domain is returned.

.8 Otherwise the meaning of *t* is returned.

End of annotations

34. $PolyVals : POLYDOM \to \mathbb{P}(POLYVAL)$

34.1 $PolyVals(pd) \triangleq$

.2 $\{\ pv \mid pv \in POLYVAL \cdot \forall d_l \in \delta_0(pd) \cdot pv(d_l) \in (|pd(d_l)|)\ \}$

Annotations to *PolyVals*:

34. This auxiliary function returns the set of all polymorphic values which belongs to the polymorphic domain which is given as an argument.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2.5 Operation Definitions

This clause presents the verification predicates for the two different kind of operation definitions. In addition the corresponding evaluation functions are given.

8.2.5.1 Verification Predicates for Operation Definitions

35. $VerifyExplOps$: **Pred**($\mathbb{E}(ExplOpDef) \times ENV$)

35.1 $VerifyExplOps(op_m,\ env) \triangleq$
.2 **let** $id_s = \underline{\text{dom}}(op_m)$ **in**
.3 **if** $id_s \subseteq \underline{\text{dom}}(env) \wedge \forall id \in id_s \cdot env(id) \in OPVAL$
.4 **then** $\forall id \in id_s \cdot$
.5 **let** $rel = env(id)$,
.6 $st_d = GetStateDom(env)$ **in**
.7 $st_d \in DOM \wedge$
.8 $rel \subseteq \{ (\|fd_1\| \times \|st_d\| \times \|st_d\| \times (\|fd_2\| \cup \{\underline{\text{nil}}\}) \times MODE)$
.9 $\mid fd_1, fd_2 \in DOM \} \wedge$
.10 $rel = EvalExplOp(op_m(id))(env)$
.11 **else** F

Annotations to *VerifyExplOps*:

35. This predicate checks whether the given environment contains proper denotations for the syntactic explicit operation definitions.
.3 All names of the explicit operation definitions must belong to the domain of the given environment, *env*, and all of them must be bound to operation values in the environment.
.4–.10 For all explicit operation names the denotation in the given environment must be identical to the result of evaluating the operation definition in the environment. In addition, it must be a relation which uses the state domain appropriately.

End of annotations

36. $VerifyImplOps$: **Pred**($\mathbb{E}(ImplOpDef) \times ENV$)

36.1 $VerifyImplOps(op_m,\ env) \triangleq$
.2 **let** $id_s = \underline{\text{dom}}(op_m)$ **in**
.3 **if** $id_s \subseteq \underline{\text{dom}}(env) \wedge \forall id \in id_s \cdot env(id) \in OPVAL$
.4 **then** $\forall id \in id_s \cdot$
.5 **let** $rel = env(id)$,
.6 $st_d = GetStateDom(env)$ **in**
.7 $st_d \in DOM \wedge$
.8 $rel \subseteq \{ (\|fd_1\| \times \|st_d\| \times \|st_d\| \times (\|fd_2\| \cup \{\underline{\text{nil}}\}) \times MODE)$
.9 $\mid fd_1, fd_2 \in DOM \} \wedge$
.10 $rel = EvalImplOp(op_m(id))(env)$
.11 **else** F

Annotations to *VerifyImplOps*:

36. This predicate checks whether the given environment contains proper denotations for the syntactic implicit operation definitions.
.3 All names of the explicit operation definitions must belong to the domain of the given environment, *env*, and all of them must be bound to operation values in the environment.
.4–.10 For all explicit operation names the denotation in the given environment must be identical to the result of evaluating the operation definition in the environment In addition, it must be a relation which uses the state domain appropriately.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.2.5.2 Evaluation Functions for Operation Definitions

```
37.   EvalImplOp : ImplOpDef → ENV → OPVAL ∪ { err }
37.1  EvalImplOp(MkTag('ImplOpDef', (h, body)))(env) ≜
  .2   let MkTag('OpHeading', (MkTag('Par', (id, tp)), restp, pre, ext)) = h,
  .3       MkTag('OpPost', (rid, post)) = body in
  .4     if ¬CheckTagEnv(env, tp) ∨ ¬CheckTagEnv(env, restp)
  .5     then err
  .6     else let i_d = EvalType(tp)(env),
  .7              o_d = if restp = nil
  .8                    then (∅⊥, { })
  .9                    else EvalType(restp)(env),
  .10             st_d = GetStateDom(env) in
  .11           if i_d = err ∨ o_d = err
  .12           then err
  .13           else let st_s = if ||st_d|| = { }
  .14                           then { nil }
  .15                           else ||st_d|| ∪ { ⊥ },
  .16                    o_s = if ||o_d|| = { }
  .17                          then { nil }
  .18                          else ||o_d|| ∪ { ⊥ } in
  .19                let r = { (i_v, i_st, o_st, o_v, m)
  .20                        | i_v ∈ ||i_d|| ∪ { ⊥ }, i_st ∈ st_s, o_st ∈ VAL,
  .21                          o_v ∈ VAL, m ∈ MODE ·
  .22                          let (e1, e2) = ExtImplOpEnv(id, i_v, rid, o_v, i_st, o_st)(env) in
  .23                            e1 ≠ err ∧ e2 ≠ err
  .24                            CheckStateConstancy(e2)(ext)(i_st, o_st) = True() ∧
  .25                            rid = nil ⟺ (o_v = nil ∧ m = cont) ∧
  .26                            rid ≠ nil ⟺ (o_v ≠ nil ∧ m = ret) ∧
  .27                            let pre_v_s = { evpre(e1) | evpre ∈ EvalExpr(pre) },
  .28                                post_v_s = { evpost(e2) | evpost ∈ EvalExpr(post) } in
  .29                              True() ∈ pre_v_s ⇒ post_v_s = { True() } } in
  .30                  if r = { } ∨
  .31                     ∃e ∈ r · let (_, _, o_st, o_v, _) = e in o_st ∉ st_s ∨ o_v ∉ o_s
  .32                  then err
  .33                  else r
```

Annotations to *EvalImplOp*:

37. This function gives meaning to implicit operation definitions.

.4–.5 If either the input or output type contains any undefined composite types, an error is reported.

.6–.10 The input and output domains for the operation are determined, while the state domain is found by *GetStateDom*. Note that it is possible that the operation has no output value, in which case an empty domain is used.

.11–.12 If any error is reported from *EvalType* the operation definition does not denote a proper operation value and an error is reported.

.13–.18 Both the state domain and the output domain can be empty. In that case nil is used as a marker to indicate that no value is used.

.19–.29 A relation with input values (i_v), input state values (i_st), output state values (o_st), output values (o_v), and a mode is defined. The different input values are restricted to their corresponding domains, while the output state and output value are allowed to be any value from *VAL*. The environments in which the pre (e_1) and post-conditions (e_2) are evaluated must first be expanded with bindings of the input and output values and input and output states (done by *ExtImplOpEnv*). With such an extended environment it is first checked that the read components of the state (if any) are unchanged (*CheckStateConstancy*). It is also required that if there is no return identifier, *rid*, the output value must be nil and the mode must be cont (indicating that no value is returned). On the other hand if there is a return identifier, the output

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

must be different from nil and the mode must be ret (meaning return). Finally, if it is possible for the pre-condition to be true, the post-condition must be true.

.29 The principle behind this is that if a valid pre-condition exists, then the post-condition must also be true. Otherwise a designer could in a step of development choose the valid pre-condition and an invalid post-condition to match it.

.30–.33 If the relation, r, is an empty set, or if either a value is returned outside the range of the operation or the returned state is outside the state domain, an error is reported. Otherwise r is returned as the meaning of the syntactic operation definition.

End of annotations

```
38.   EvalExplOp : ExplOpDef → ENV → OPVAL ∪ { err }
38.1  EvalExplOp(MkTag('ExplOpDef', (h, body)))(env) ≜
  .2    let MkTag('OpHeading', (MkTag('Par', (id, tp)), restp, pre, ext)) = h in
  .3      if ¬CheckTagEnv(env, tp) ∨ ¬CheckTagEnv(env, restp)
  .4      then err
  .5      else let i_d = EvalType(tp)(env),
  .6               o_d = if restp = nil
  .7                     then (∅⊥, { })
  .8                     else EvalType(restp)(env),
  .9               st_d = GetStateDom(env) in
  .10            if i_d = err ∨ o_d = err
  .11            then err
  .12            else let st_s = if ||st_d|| = { }
  .13                            then { nil }
  .14                            else ||st_d|| ∪ { ⊥ },
  .15                     o_s = if ||o_d|| = { }
  .16                           then { nil }
  .17                           else ||o_d|| ∪ { ⊥ } in
  .18                 let r = { (i_v, i_st, o_st, o_v, m)
  .19                         | i_v ∈ ||i_d|| ∪ { ⊥ }, i_st ∈ st_s, o_st ∈ VAL,
  .20                           o_v ∈ VAL, m ∈ MODE ·
  .21                           let (env', env'') = ExtExplOpEnv(id, i_v, i_st, o_st, ext)(env) in
  .22                             CheckStateConstancy(env')(ext)(i_st, o_st) = True() ∧
  .23                             let pre_v_s = { evpre(env') | evpre ∈ EvalExpr(pre) },
  .24                                 lsev = EvalStmt(body) in
  .25                               True() ∈ pre_v_s ⇒
  .26                               ∃ sev ∈ lsev · sev(env') = (env'', m, o_v) } in
  .27                   if r = { } ∨
  .28                      ∃e ∈ r · let (_, _, o_st, o_v, _) = e in o_st ∉ st_s ∨ o_v ∉ o_s
  .29                   then err
  .30                   else r
```

Annotations to *EvalExplOp*:

38. This function gives meaning to explicit operation definitions.

.3–.4 If either the input or output type contains any undefined composite types, an error is reported.

.5–.9 The input and output domain for the operation is determined, while the state domain is found by *GetStateDom*. Note that it is possible that the operation has no output, in which case an empty domain is used.

.10–.11 If any error is reported from *EvalType*, the operation definition does not denote a proper operation value and an error is reported.

.12–.17 Both the state domain and the output domain can be empty. In that case nil is used as a marker to indicate that no value is used.

.18–.26 A relation with input values (i_v), input state values (i_st), output state values (o_st), output values (o_v), and a mode (exit indicates an error situation, ret indicates that a value is returned, while cont indicates that if the operation is used as a statement the execution can continue) is defined. The different

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

input values are restricted to their corresponding domains, while the output value and the output state are allowed to be any value from *VAL*. The environment in which the pre and body statement are evaluated must first be expanded with bindings of the input value and the input state value (done by *ExtExplOpEnv*). The environment which should be returned as the first component when a statement evaluator is applied with the expanded environment is also returned from *ExtExplOpEnv* (*env''*). With this extended environment it is first checked that the read components of the state (if any) are unchanged (*CheckStateConstancy*). Finally, it is checked whether the pre-condition can be true in which case a statement evaluator (for the body of the operation which returns a three tuple with an environment with the state after calling the operation, if any), a mode, and an output value must exist.

.27 – .30 If the relation, r, is an empty set, or if either a value is returned outside the range of the operation or the returned state is outside the state domain, an error is reported. Otherwise r is returned as the meaning of the syntactic operation definition.

End of annotations

8.3 Expressions

This clause presents the evaluation functions for all different kinds of expressions.

In the following LITERAL is used as an abbreviation for the set of literal tags.

LITERAL = { *'BoolLit'*, *'NumLit'*, *'CharLit'*, *'TextLit'*, *'QuoteLit'*, *'NilLit'*}

and ID as an abbreviation for the set of identifier tags.

ID = { *'ValueId'*, *'PreId'*, *'PostId'*, *'InvId'*, *'InitId'*, *'MkId'*, *'IsId'*,

'StateConstId', *'RecSelId'*, *'RecModId'*, *'RecConsId'*, *'TypeEnvId'*, *'TagEnvId'*}

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

39. *EvalExpr* : *Expr* → *LEEval*

39.1 *EvalExpr*(*expr*) ≜
.2 **cases** *ShowTag*(*expr*) :
.3 *'LetExpr'* → *EvalLetExpr*(*expr*),
.4 *'LetBeSTExpr'* → *EvalLetBeSTExpr*(*expr*),
.5 *'DefExpr'* → *EvalDefExpr*(*expr*),
.6 *'IfExpr'* → *EvalIfExpr*(*expr*),
.7 *'CasesExpr'* → *EvalCasesExpr*(*expr*),
.8 *'UnaryExpr'* → *EvalUnaryExpr*(*expr*),
.9 *'BinaryExpr'* → *EvalBinaryExpr*(*expr*),
.10 *'AllOrExistsExpr'* → *EvalAllOrExistsExpr*(*expr*),
.11 *'ExistsUniqueExpr'* → *EvalExistsUniqueExpr*(*expr*),
.12 *'IotaExpr'* → *EvalIotaExpr*(*expr*),
.13 *'SetEnumeration'* → *EvalSetEnumeration*(*expr*),
.14 *'SetComprehension'* → *EvalSetComprehension*(*expr*),
.15 *'SetRange'* → *EvalSetRange*(*expr*),
.16 *'SeqEnumeration'* → *EvalSeqEnumeration*(*expr*),
.17 *'SeqComprehension'* → *EvalSeqComprehension*(*expr*),
.18 *'SubSequence'* → *EvalSubSequence*(*expr*),
.19 *'MapEnumeration'* → *EvalMapEnumeration*(*expr*),
.20 *'MapComprehension'* → *EvalMapComprehension*(*expr*),
.21 *'TupleConstructor'* → *EvalTupleConstructor*(*expr*),
.22 *'RecordConstructor'* → *EvalRecordConstructor*(*expr*),
.23 *'RecordModifier'* → *EvalRecordModifier*(*expr*),
.24 *'Apply'* → *EvalApplyExpr*(*expr*),
.25 *'FieldSelect'* → *EvalFieldSelectExpr*(*expr*),
.26 *'FctTypeInst'* → *EvalFctTypeInstExpr*(*expr*),
.27 *'Lambda'* → *EvalLambda*(*expr*),
.28 *'IsExpr'* → *EvalIsExpr*(*expr*),
.29 LITERAL → *EvalLiteralExpr*(*expr*),
.30 ID → *EvalId*(*expr*),
.31 *'OldId'* → *EvalOldId*(*expr*)

Annotations to *EvalExpr*:
39. This function gives meaning to expressions.
.3–.31 The meaning of the different kinds of expressions is split into sub-functions.
End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.1 Local Binding Expressions

```
40.   EvalLetExpr : LetExpr → LEEval
40.1  EvalLetExpr(MkTag('LetExpr', (vals, explfns, implfns, in))) ≜
 .2    let ids = ⋃{ ColIdSet(p) | p ∈ dom(vd_m) } in
 .3      let lev = { λenv.let fnenv = restrict(env', dom(explfns) ∪ dom(implfns) ∪ ids) in
 .4                        let env'' = overwrite(env, fnenv) in
 .5                          if def(env'') = err ∨ BotEnv(def(env''))
 .6                          then ⊥
 .7                          else ev(def(env''))
 .8                | def ∈ EvalLocalDef(vals, explfns, implfns), ev ∈ EvalExpr(in), env' ∈ ENV ·
 .9                  VerifyExplFns(explfns, env') ∧ VerifyImplFns(implfns, env') ∧
.10                  VerifyValues(vals, env') } in
.11       if lev = { }
.12       then { λenv . ⊥ }
.13       else lev
```

Annotations to *EvalLetExpr*:

40. This function gives meaning to *Let* expressions[1)].

.3 The denotations of the locally defined functions are extracted from the indexed environment.

.5–.6 If either the meaning of the local definitions is an error, or a non-strict binding has occurred, an error is reported by returning the bottom value.

.7 The evaluator for the 'in-expression' is applied with a context where the local bindings are in context.

.8–.10 This is done for all possible definers and all possible expression evaluators for the 'in-expression' and for all possible environments which provide appropriate semantics for the locally defined functions and values. In this way locally defined functions and values are given the same semantics as globally defined ones.

.11–.13 If no statement evaluators were collected the let statement denotes the singleton set with the expression evaluator that returns bottom.

End of annotations

[1)] Notice that the difference between a *Def* expression and a *Let* expression is that the *Def* expression is allowed to refer to components of the state, while a *Let* expression is purely applicative. In addition a *Let*-expression is interpreted as the least fixed point of a number of mutually recursive definitions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
41.   EvalLocalDef : 𝕄(Pattern, ValDef) × 𝔼(ExplFnDef) × 𝔼(ImplFnDef) → LDef
41.1  EvalLocalDef(vd_m, efd_m, ifd_m) ≜
  .2    let ids = ⋃{ ColIdSet(p) | p ∈ dom(vd_m) } in
  .3      if ids ∩ dom(efd_m) ≠ { } ∨
  .4         ids ∩ dom(ifd_m) ≠ { } ∨
  .5         dom(efd_m) ∩ dom(ifd_m) ≠ { }
  .6      then { λenv.err }
  .7      else let fd_m = { MkTag('PatternId', id) ↦ efd_m(id)
  .8                      | id ∈ dom(efd_m) } in
  .9           let mul_ldef =
  .10               { λenv.let fun_m_s = MakeFns(ifd_m)(env) in
  .11                      if fun_m_s = err
  .12                      then { }
  .13                      else { env'
  .14                           | env' ∈ ENV ·
  .15                             ∃ fun_m ∈ fun_m_s·
  .16                               env' = tf(overwrite(env, fun_m)) ∧
  .17                             VerifyImplFns(ifd_m)(env') }
  .18               | tf ∈ EvalExplDef(merge(vd_m, fd_m)) } in
  .19         PropErr(mul_ldef)
```

Annotations to *EvalLocalDef*:

41. The function *EvalLocalDef* gives meaning to a set of possibly mutually recursive function and value definitions.

.3–.6 If there is a clash between the identifiers used in the definitions then the one-element loose definer, mapping every environment to err is returned.

.7–.18 Otherwise the set of definers is constructed.

.10–.17 A set of multi-functions is constructed. Each of them will for a given environment return a (possibly empty) set of new environments. This is done by applying a transformation, *tf*, corresponding to the evaluation of explicit definitions to an environment which has been extended by all possible interpretations of implicit function definitions. In all possible resulting environments the pre and post-conditions for the implicitly defined functions must be fulfilled, which is checked by *VerifyImplFns*.

.19 A loose definer is constructed by propagating the looseness from the multi-functions.

End of annotations

```
42.   EvalLetBeSTExpr : LetBeSTExpr → LEEval
42.1  EvalLetBeSTExpr(MkTag('LetBeSTExpr', (bind, st, in))) ≜
  .2    PropE({ λenv.let venv_s = ⋃{ bev(env) | bev ∈ EvalBind(bind) } in
  .3                  let venv_s' = { venv | venv ∈ venv_s · venv ≠ err ∧ ¬BotEnv(venv) ∧
  .4                                  stev(overwrite(env, venv)) = True() } in
  .5                  { inev(overwrite(env, venv)) | venv ∈ venv_s' }
  .6          | stev ∈ EvalExpr(st), inev ∈ EvalExpr(in) })
```

Annotations to *EvalLetBeSTExpr*:

42. This function gives meaning to let be (such that) expressions.

.2 All possible bindings are gathered.

.3–.4 All possible value environments which fulfil the 'such that' predicate, *st*, are extracted from all possible binding environments. Notice that only strict binding environments are considered. If an error is reported from the pattern matching these two lines yields an empty set, which makes the semantics of the entire let be such that expression yield bottom.

.5 The set of values returned from applying the 'in-expression' evaluator with the given environment extended with all possible value environments.

.6 This is done for all possible 'such that' predicate evaluators and all possible expression evaluators for the

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

'in-expression'.

End of annotations

```
43.   EvalDefExpr : DefExpr → LEEval
43.1  EvalDefExpr(MkTag('DefExpr', (lhs, rhs, in))) ≙
  .2   { λenv.let venv = pev(ev(env))(env) in
  .3            if venv ∈ { err, unmatch } ∨ BotEnv(venv)
  .4            then ⊥
  .5            else inev(overwrite(env, venv))
  .6    | pev ∈ EvalPattern(lhs), ev ∈ EvalExpr(rhs), inev ∈ EvalExpr(in) }
```

Annotations to *EvalDefExpr*:

43. This function gives meaning to definition expressions[1)].

.3–.4 If either an error is returned (or the matching failed) from matching the pattern evaluator against the value of the right hand side expression (using the given environment, *env*), or the matching results in a non-strict binding, an error is reported.

.5 The evaluator for the 'in-expression' is applied with a context where these local bindings are in context.

.6 This is done for all possible evaluators for the left hand side pattern, *lhs*, the right hand side expression, *rhs*, and the in expression, *in*.

End of annotations

8.3.2 Conditional Expressions

```
44.   EvalIfExpr : IfExpr → LEEval
44.1  EvalIfExpr(MkTag('IfExpr', (test, cons, altn))) ≙
  .2   { λenv.if testev(env) ∉ BOOL_VAL
  .3            then ⊥
  .4            else if testev(env) = True()
  .5                 then consev(env)
  .6                 else altnev(env)
  .7    | testev ∈ EvalExpr(test), consev ∈ EvalExpr(cons), altnev ∈ EvalExpr(altn) }
```

Annotations to *EvalIfExpr*:

44. This function gives meaning to an if-then-else expression.

.2–.3 If the expression evaluator for the *test* expression applied with the given environment does not yield a Boolean value, bottom is returned.

.4–.5 If the expression evaluator, *testev*, yields *True*() when it is applied with the given environment, the consequence expression, *cons*, is applied with the environment.

.6 Otherwise the alternative expression, *altn*, is applied with the environment.

.7 This is done for all possible evaluators for the test expression, the consequence and the alternative expressions.

End of annotations

[1)] Notice that the difference between a *Def* expression and a *Let* expression is that the right hand side of the binding part of a *Def* expression is allowed to refer to parts of the state, while the right hand side of the binding part of a *Let* expression is purely applicative.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
45.   EvalCasesExpr : CasesExpr → LEEval
45.1  EvalCasesExpr(MkTag('CasesExpr', (sel, altns))) ≜
  .2    PropE({ λenv.if selev(env) = ⊥
  .3                 then { ⊥ }
  .4                 else { ev(env) | ev ∈ ApplyCasesExpr(selev(env), altns) }
  .5          | selev ∈ EvalExpr(sel) })
```

Annotations to *EvalCasesExpr*:

45. This function gives meaning to a cases expression.

.2 – .3 If an evaluator for the selector expression applied with the given environment yields bottom, bottom is returned.

.4 The meaning of the cases expression is found by applying *ApplyCasesExpr* with the selector value and the list of cases alternatives.

End of annotations

```
46.   ApplyCasesExpr : VAL × 𝕃(CaseExprAltn) → LEEval
46.1  ApplyCasesExpr(val, csa_l) ≜
  .2    if len(csa_l) = 0
  .3    then { λenv. ⊥ }
  .4    else { λenv.let res = caev(env) in
  .5                 if res = unmatch
  .6                 then c(env)
  .7                 else res
  .8         | caev ∈ ApplyCaseExprAltn(val, hd(csa_l)),
  .9           c ∈ ApplyCasesExpr(val, tl(csa_l)) }
```

Annotations to *ApplyCasesExpr*:

46. This function applies a selector value to a list of case expression alternatives. The first alternative which contains a pattern which can be successfully matched against the selector value is chosen, and the corresponding loose expression evaluator is returned.

.2 – .3 If the list of case expression alternatives is empty, no pattern matching has been successful, and bottom will be returned to indicate an error situation.

.4 – .9 Otherwise it is checked whether the first case expression alternative matches the selector value, *val*, in the given environment. If this is the case, the loose expression evaluator from the corresponding expression is returned. Alternatively the semantics is defined by means of recursion with the tail of the list of expression alternatives.

End of annotations

```
47.   ApplyCaseExprAltn : VAL × CaseExprAltn → ℙ(ENV → (VAL ∪ { unmatch }))
47.1  ApplyCaseExprAltn(val, MkTag('CaseAltn', (match, body))) ≜
  .2    PropE({ λenv.let s = { boev(env')
  .3                           | boev ∈ EvalExpr(body),
  .4                             env' ∈ NonErrPat(match, val, env) } in
  .5                 if s = { }
  .6                 then { unmatch }
  .7                 else s
  .8          })
```

Annotations to *ApplyCaseExprAltn*:

47. This function applies a selector value to an expression case alternative. If the alternative contains a pattern which can be successfully matched against the selector value, *val*, the expression evaluators for the expression, from the alternative, is returned. Otherwise the token unmatch is returned.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

End of annotations

8.3.3 Unary Expressions

```
48.    EvalUnaryExpr : UnaryExpr → LEEval
48.1   EvalUnaryExpr(MkTag('UnaryExpr', (opr, arg))) ≙
  .2    { λenv.let val = ev(env) in
  .3            cases opr :
  .4              NUMPLUS,NUMMINUS,NUMABS,FLOOR          → UNum(opr, val),
  .5              NOT                                    → Not(val),
  .6              SETCARD,SETPOWER                       → SetXx(opr, val),
  .7              SETDISTRUNION,SETDISTRINTERSECT        → SetDi(opr, val),
  .8              SEQHEAD,SEQTAIL,SEQLEN,SEQELEMS,SEQINDICES → SeqXx(opr, val),
  .9              SEQDISTRCONC                           → SeqDi(val),
  .10             MAPDOM,MAPRNG,MAPINVERSE               → MapXx(opr, val),
  .11             MAPDISTRMERGE                          → MapDi(val)
  .12    | ev ∈ EvalExpr(arg) }
```

Annotations to *EvalUnaryExpr*:

48. This function gives meaning to unary expressions.
.2–.3 If the expression evaluator applied with the given environment denotes a proper value, the unary operator will be applied.
.4 Apply the unary operator to a proper value.
.5 This is done for all possible expression evaluators for the argument expression, *arg*.

End of annotations

8.3.3.1 Numeric Operations

```
49.    UNum : { NUMPLUS,NUMMINUS,NUMABS,FLOOR } × VAL → VAL
49.1   UNum(un_op, val) ≙
  .2    if val ∈ NUM_VAL
  .3    then let n = StripNumTagVal(val) in
  .4            cases un_op :
  .5              NUMPLUS  → MkTag('num', n),
  .6              NUMMINUS → MkTag('num', -n),
  .7              NUMABS   → if n ≥ 0
  .8                         then MkTag('num', n)
  .9                         else MkTag('num', -n),
  .10             FLOOR    → Iota({ t ∈ ℤ | t ≤ n ∧ t + 1 > n })
  .11   else ⊥
```

Annotations to *UNum*:

49. This function applies one of the 'built-in' unary numeric operators to a value (if it is a numeric value).
.5–.10 Unary plus, minus and absolute value and the 'floor operator' are applied in the obvious way. The floor operator works on real values like, FLOOR(5.7) = 5.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.3.2 Logical Operation

```
50.    Not : VAL → VAL
50.1   Not(val) ≜
  .2    if val ∈ BOOL_VAL
  .3    then if val = True()
  .4           then False()
  .5           else True()
  .6    else ⊥
```

Annotations to *Not*:

50. This function perform logical negation of the argument value (if it is a Boolean value).

End of annotations

8.3.3.3 Set Operations

```
51.    SetXx : { SETCARD,SETPOWER } × VAL → VAL
51.1   SetXx(un_op, val) ≜
  .2    if val ∈ SET_VAL
  .3    then let s = StripSetTagVal(val) in
  .4           cases un_op :
  .5             SETCARD   → MkTag('num', card(s)),
  .6             SETPOWER → MkTag('set', { MkTag('set', sub) | sub ∈ IF(s) })
  .7    else ⊥
```

Annotations to *SetXx*:

51. This function applies one of the 'built-in' unary set operators to an argument value (if it is a set value).

.5 – .6 The finite set cardinality operator and the finite powerset operator are applied.

End of annotations

```
52.    SetDi : { SETDISTRUNION,SETDISTRINTERSECT } × VAL → VAL
52.1   SetDi(un_op, val) ≜
  .2    if val ∈ SET_VAL
  .3    then let set_set_val = StripSetTagVal(val) in
  .4           if set_set_val ⊆ SET_VAL
  .5           then let ss = { StripSetTagVal(s) | s ∈ set_set_val} in
  .6                  cases un_op :
  .7                    SETDISTRUNION      → MkTag('set', ⋃(ss))
  .8                    SETDISTRINTERSECT → if ss = { }
  .9                                        then ⊥
  .10                                       else MkTag('set', ⋂(ss))
  .11          else ⊥
  .12   else ⊥
```

Annotations to *SetDi*:

52. This function applies one of the 'built-in' unary distributed set operators to an argument value (if it is a set of set values).

.6 – .10 If the value fulfils the requirements for application of the set distributed union operator and the set intersection operator, it is done. Note that the distributed intersection of an empty set is an error.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

End of annotations

8.3.3.4 Sequence Operations

```
53.   SeqXx : { SEQHEAD,SEQTAIL,SEQLEN,SEQELEMS,SEQINDICES } × VAL → VAL
53.1  SeqXx(un_op, val) ≙
  .2   if val ∈ LIST_VAL
  .3   then let seq_val = StripSeqTagVal(val) in
  .4          cases un_op :
  .5            SEQHEAD    → if seq_val = [ ]
  .6                         then ⊥
  .7                         else hd(seq_val),
  .8            SEQTAIL    → if seq_val = [ ]
  .9                         then ⊥
  .10                        else MkTag('seq', tl(seq_val)),
  .11           SEQLEN     → MkTag('num', len(seq_val)),
  .12           SEQELEMS   → MkTag('set', elems(seq_val)),
  .13           SEQINDICES → MkTag('set', inds(seq_val))
  .14  else ⊥
```

Annotations to *SeqXx*:

53. This function applies one of the 'built-in' unary sequence operators to an argument value (if it is a list).

.5 – .13 The operators for sequence length, head, tail, elements, and indices are dealt with.

End of annotations

```
54.   SeqDi : VAL → VAL
54.1  SeqDi(val) ≙
  .2   if val ∈ LIST_VAL
  .3   then let seq_seq_val = StripSeqTagVal(val) in
  .4          if elems(seq_seq_val) ⊆ LIST_VAL
  .5          then let ll = ApplySeq(StripSeqTagVal)(seq_seq_val) in
  .6                 MkTag('seq', Join(ll))
  .7          else ⊥
  .8   else ⊥
```

Annotations to *SeqDi*:

54. This function applies the 'built-in' distributed sequence concatenation to the argument value (if it is a list of lists).

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.3.5 Map Operations

```
55.   MapXx : { MAPDOM,MAPRNG,MAPINVERSE } × VAL → VAL
55.1  MapXx(un_op, val) ≜
  .2   if val ∈ MAP_VAL
  .3   then let map_val = StripMapTagVal(val) in
  .4          cases un_op :
  .5            MAPDOM     → MkTag('set', dom(map_val)),
  .6            MAPRNG     → if ∀ a ∈ dom(map_val) · map_val(a) ∈ FLAT_VAL
  .7                         then MkTag('set', rng(map_val))
  .8                         else ⊥,
  .9            MAPINVERSE → if Injective(m) ∧
  .10                           ∀ a ∈ dom(m) · m(a) ∈ FLAT_VAL
  .11                        then MkTag('map', { m(d) ↦ d | d ∈ dom(m) })
  .12                        else ⊥
  .13  else ⊥
```

Annotations to *MapXx*:

55. This function applies the 'built-in' map inversion, map domain and range operators to the argument value (if it is a map value, and all values in the range of the map are flat unless the domain operator is used).

.9–.12 If the mapping is not injective, or any of the values in the range are not flat, then the map cannot be inverted.

End of annotations

```
56.   MapDi : VAL → VAL
56.1  MapDi(val) ≜
  .2   if val ∈ SET_VAL
  .3   then let set_map_val = StripSetTagVal(val) in
  .4          if ∀m ∈ set_map_val · m ∈ MAP_VAL
  .5          then let mm = { StripMapTagVal(m) | m ∈ set_map_val } in
  .6                 if Compatible(mm)
  .7                 then MkTag('map', Merge(mm))
  .8                 else ⊥
  .9          else ⊥
  .10  else ⊥
```

Annotations to *MapDi*:

56. This function applies the 'built-in' distributed map merge operator to the argument value (if it is a set of maps which are compatible).

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.4 Binary Expressions

```
57.    EvalBinaryExpr : BinaryExpr → LEEval
57.1   EvalBinaryExpr(MkTag('BinaryExpr', (left, opr, right))) ≜
  .2    { λenv.let l_v = l_ev(env),
  .3                 r_v = r_ev(env) in
  .4               cases opr :
  .5                 NUMPLUS,NUMMINUS,NUMMULT,NUMDIV   → NumBin(opr, l_v, r_v),
  .6                 INTDIV                            → IntDiv(l_v, r_v),
  .7                 NUMREM                            → NumRem(l_v, r_v),
  .8                 NUMMOD                            → NumMod(l_v, r_v),
  .9                 NUMLT,NUMLE,NUMGT,NUMGE           → NumComp(opr, l_v, r_v)
  .10                EQ,NE                             → EqualOpr(opr, l_v, r_v),
  .11                OR,AND,IMPLY,EQUIV                → LogOpr(opr, l_v, r_v),
  .12                INSET,NOTINSET                    → CheckMembership(opr, l_v, r_v),
  .13                SUBSET,PROPERSUBSET               → SubSet(opr, l_v, r_v),
  .14                SETUNION,SETDIFFERENCE,SETINTERSECT → SetBin(opr, l_v, r_v),
  .15                SEQCONC                           → SeqConc(l_v, r_v),
  .16                MAPORSEQMOD                       → MapOrSeqMod(l_v, r_v),
  .17                MAPDOMRESTRTO, MAPDOMRESTRBY,
  .18                MAPRNGRESTRTO, MAPRNGRESTRBY      → MapToSetOpr(opr, l_v, r_v),
  .19                MAPMERGE                          → MapMerge(l_v, r_v),
  .20                COMPOSE                           → Compose(l_v, r_v),
  .21                ITERATE                           → Iterate(l_v, r_v)
  .22   | l_ev ∈ EvalExpr(left), r_ev ∈ EvalExpr(right) }
```

Annotations to *EvalBinaryExpr*:

57. This function gives meaning to binary expressions.
.5–.21 The binary application is done by separate sub-functions.
.22 This is done for each evaluator for the *left* and *right* expression.

End of annotations

8.3.4.1 Numeric Operations

```
58.    NumBin : { NUMPLUS, NUMMINUS, NUMMULT, NUMDIV } × VAL × VAL → VAL
58.1   NumBin(bin_op, l_val, r_val) ≜
  .2    if (l_val ∈ NUM_VAL) ∧ (r_val ∈ NUM_VAL)
  .3    then let l_v = StripNumTagVal(l_val),
  .4               r_v = StripNumTagVal(r_val) in
  .5             cases bin_op :
  .6               NUMPLUS  → MkTag('num', l_v + r_v),
  .7               NUMMINUS → MkTag('num', l_v - r_v),
  .8               NUMMULT  → MkTag('num', l_v × r_v),
  .9               NUMDIV   → if r_v = 0
  .10                         then ⊥
  .11                         else MkTag('num', l_v / r_v)
  .12   else ⊥
```

Annotations to *NumBin*:

58. This function applies one of the 'built-in' unary numeric operators to two values (if they are both numeric values).
.6–.11 Numerical plus, minus, multiplication and division functions are applied in the obvious way.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

End of annotations

Binary operators for integer values

There are three binary operators concerning division, that are defined for integer operands only; *integer division*, *remainder*, and *modulus*:

l_v div r_v $\quad$ l_v rem r_v $\quad$ l_v mod r_v

Before presenting the semantics of these operators an example (taken from Daw91) is shown to illustrating the signs used:

l_v	r_v	l_v rem r_v	l_v div r_v	l_v mod r_v	floor (l_v/r_v)
+14	+3	+2	+4	+2	+4
+14	−3	+2	−4	−1	−5
−14	+3	−2	−4	+1	−5
−14	−3	−2	+4	−2	+4

□

59. $IntDiv : VAL \times VAL \rightarrow VAL$

59.1 $IntDiv(l_val, r_val) \triangleq$
.2 **if** $(l_val \in NUM_VAL) \wedge (r_val \in NUM_VAL)$
.3 **then let** $l_v = StripNumTagVal(l_val)$,
.4 $r_v = StripNumTagVal(r_val)$ **in**
.5 **if** $r_v = 0$
.6 **then** $\perp$
.7 **else if** $l_v \in \mathbb{Z} \wedge r_v \in \mathbb{Z}$
.8 **then let** $q = Iota(\{\, p \in \mathbb{Z} \mid \exists\, r \in \mathbb{Z} \cdot \underline{abs}(r) < \underline{abs}(r_v) \wedge$
.9 $r \times l_v \geq 0 \wedge l_v = r_v \times p + r \,\})$ **in**
.10 $MkTag('num', q)$
.11 **else** $\perp$
.12 **else** $\perp$

Annotations to *IntDiv*:

59. This function applies the 'built-in' integer division operator to two values (if they are both integers). It can be explained by means of a relation:
$l_v = r_v \times (l_v \text{ div } r_v) + (l_v \text{ rem } r_v)$

.8–.10 Given that both values are integers, the unique (integer) divisor is created and returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
60.    NumRem : VAL × VAL → VAL
60.1   NumRem(l_val, r_val) ≙
  .2    if (l_val ∈ NUM_VAL) ∧ (r_val ∈ NUM_VAL)
  .3    then let l_v = StripNumTagVal(l_val),
  .4               r_v = StripNumTagVal(r_val) in
  .5           if r_v = 0
  .6           then ⊥
  .7           else if l_v ∈ ℤ ∧ r_v ∈ ℤ
  .8                then let r = Iota({ r ∈ ℤ | abs(r) < abs(r_v) ∧ r×l_v ≥0 ∧
  .9                                                ∃p ∈ ℤ · l_v = r_v×p + r}) in
  .10                      MkTag('num', r)
  .11               else ⊥
  .12   else ⊥
```

Annotations to *NumRem*:

60. This function applies the 'built-in' true remainder for integer division to two values (if they are both integers). It can be explained by means of a relation:
$l_v = r_v \times (l_v \text{ div } r_v) + (l_v \text{ rem } r_v)$

.8–.10 Given that both values are integers, the unique (integer) remainder is created and returned.

End of annotations

```
61.    NumMod : VAL × VAL → VAL
61.1   NumMod(l_val, r_val) ≙
  .2    if (l_val ∈ NUM_VAL) ∧ (r_val ∈ NUM_VAL)
  .3    then let l_v = StripNumTagVal(l_val),
  .4               r_v = StripNumTagVal(r_val) in
  .5           if r_v = 0
  .6           then ⊥
  .7           else if (l_v ∈ ℤ ∧ r_v ∈ ℤ)
  .8                then let floor = Iota({ t ∈ ℤ | t ≤ l_v / r_v ∧ t + 1 > l_v / r_v}) in
  .9                         MkTag('num', l_v - r_v × floor)
  .10               else ⊥
  .11   else ⊥
```

Annotations to *NumMod*:

61. This function applies the 'built-in' modulus operator to two values (if they are both natural numbers). The modulus operator is a generalization of the concept of residue from number theory. It can be explained by means of a relation:
$l_v = r_v \times \text{floor } (l_v / r_v) + (l_v \text{ mod } r_v)$.

.8–.9 Given that both values are natural numbers, the unique modulus is created and returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.4.2 Numeric Comparison Operators

```
62.   NumComp : { NUMLT,NUMLE,NUMGT,NUMGE } × VAL × VAL → VAL
62.1  NumComp(bin_op, l_val, r_val) ≙
  .2   if (l_val ∈ NUM_VAL) ∧ (r_val ∈ NUM_VAL)
  .3   then let l_v = StripNumTagVal(l_val),
  .4             r_v = StripNumTagVal(r_val) in
  .5          cases bin_op :
  .6            NUMLT → if l_v < r_v
  .7                    then True()
  .8                    else False(),
  .9            NUMLE → if l_v ≤ r_v
  .10                   then True()
  .11                   else False(),
  .12           NUMGT → if l_v > r_v
  .13                   then True()
  .14                   else False(),
  .15           NUMGE → if l_v ≥ r_v
  .16                   then True()
  .17                   else False()
  .18  else ⊥
```

Annotations to *NumComp*:

62. This function applies the 'built-in' binary numeric comparison operators to two values (if they are both numeric values).

.6 – .17 The numeric comparison operators: less than (or equal to), greater than (or equal to) are applied.

End of annotations

8.3.4.3 Equality Operations

```
63.   EqualOpr : { EQ,NE } × VAL × VAL → VAL
63.1  EqualOpr(opr, l_v, r_v) ≙
  .2   if (l_v = ⊥ ∨ r_v = ⊥) ∨ (l_v ∉ FLAT_VAL ∨ r_v ∉ FLAT_VAL)
  .3   then ⊥
  .4   else cases opr :
  .5          EQ → if l_v = r_v
  .6               then True()
  .7               else False()
  .8          NE → if l_v = r_v
  .9               then False()
  .10              else True()
```

Annotations to *EqualOpr*:

63. This function applies the 'built-in' binary equality operators to two non-flat values. Thus, equality cannot be used between functions.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.4.4 Logical Operations

64. $LogOpr : \{ \text{OR,AND,IMPLY,EQUIV} \} \times VAL \times VAL \rightarrow VAL$

64.1 $LogOpr(opr, l_v, r_v) \triangleq$

.2 **cases** *opr* :

.3 OR → **if** $l_v \in BOOL_VAL_{\perp} \wedge r_v \in BOOL_VAL_{\perp}$

.4 **then if** $(l_v = True()) \vee (r_v = True())$

.5 **then** *True*()

.6 **else if** $(l_v = False()) \wedge (r_v = False())$

.7 **then** *False*()

.8 **else** $\perp$

.9 **else** $\perp$,

.10 AND → **if** $l_v \in BOOL_VAL_{\perp} \wedge r_v \in BOOL_VAL_{\perp}$

.11 **then if** $(l_v = False()) \vee (r_v = False())$

.12 **then** *False*()

.13 **else if** $(l_v = True()) \wedge (r_v = True())$

.14 **then** *True*()

.15 **else** $\perp$

.16 **else** $\perp$,

.17 IMPLY → **if** $l_v \in BOOL_VAL_{\perp} \wedge r_v \in BOOL_VAL_{\perp}$

.18 **then if** $(l_v = False()) \vee (r_v = True())$

.19 **then** *True*()

.20 **else if** $(l_v = True()) \wedge (r_v = False())$

.21 **then** *False*()

.22 **else** $\perp$

.23 **else** $\perp$,

.24 EQUIV → **if** $(l_v = True() \wedge r_v = True()) \vee$

.25 $(l_v = False() \wedge r_v = False())$

.26 **then** *True*()

.27 **else if** $(l_v = True() \wedge r_v = False()) \vee$

.28 $(l_v = False() \wedge r_v = True())$

.29 **then** *False*()

.30 **else** $\perp$

Annotations to *LogOpr*:

64. This function applies the 'built-in' binary logical operators to the two Boolean arguments. Here the 3 valued logic with weak equality based on Kle52. In Jon86 and Jon90 is called LPF – Logic for Partial Functions.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.4.5 Logical Set Operations

```
65.   CheckMembership : { INSET,NOTINSET } × VAL × VAL → VAL
65.1  CheckMembership(mem, val, set) ≜
  .2   if set ∈ SET_VAL ∧ val ≠ ⊥
  .3   then let s = StripSetTagVal(set) in
  .4          cases mem :
  .5            INSET    → if val ∈ s
  .6                       then True()
  .7                       else False(),
  .8            NOTINSET → if val ∈ s
  .9                       then False()
  .10                      else True()
  .11  else ⊥
```

Annotations to *CheckMembership*:

65. This function applies the 'built-in' binary set membership operators to two values (if the second value is a set value).

.5–.10 The 'in set' operator and the 'not in set' operator are applied.

End of annotations

```
66.   SubSet : { SUBSET,PROPERSUBSET } × VAL × VAL → VAL
66.1  SubSet(bin_op, l_val, r_val) ≜
  .2   if (l_val ∈ SET_VAL) ∧ (r_val ∈ SET_VAL)
  .3   then let l_v = StripSetTagVal(l_val),
  .4             r_v = StripSetTagVal(r_val) in
  .5          cases bin_op :
  .6            SUBSET       → if l_v ⊆ r_v
  .7                           then True()
  .8                           else False(),
  .9            PROPERSUBSET → if l_v ⊂ r_v
  .10                          then True()
  .11                          else False()
  .12  else ⊥
```

Annotations to *SubSet*:

66. This function deals with the subset and proper subset operations.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.4.6 Set Operations

67. $SetBin : \{ \text{SETUNION,SETDIFFERENCE,SETINTERSECT} \} \times VAL \times VAL \rightarrow VAL$

67.1 $SetBin(bin_op, l_val, r_val) \triangleq$
.2 **if** $(l_val \in SET_VAL) \wedge (r_val \in SET_VAL)$
.3 **then let** $l_v = StripSetTagVal(l_val)$,
.4 $r_v = StripSetTagVal(r_val)$ **in**
.5 **cases** bin_op :
.6 SETUNION $\rightarrow MkTag(\text{'set'}, l_v \cup r_v)$,
.7 SETDIFFERENCE $\rightarrow MkTag(\text{'set'}, l_v \backslash r_v)$,
.8 SETINTERSECT $\rightarrow MkTag(\text{'set'}, l_v \cap r_v)$
.9 **else** $\perp$

Annotations to *SetBin*:

67. This function applies the 'built-in' binary set operators to two values (if they are both set values).
.6–.8 The operators for 'set union', 'set intersection', and 'set difference' are applied.

End of annotations

8.3.4.7 Sequence Operation

68. $SeqConc : VAL \times VAL \rightarrow VAL$

68.1 $SeqConc(l_val, r_val) \triangleq$
.2 **if** $(l_val \in LIST_VAL) \wedge (r_val \in LIST_VAL)$
.3 **then let** $l_v = StripSeqTagVal(l_val)$,
.4 $r_v = StripSeqTagVal(r_val)$ **in**
.5 $MkTag(\text{'seq'}, \underline{\text{join}}(l_v, r_v))$
.6 **else** $\perp$

Annotations to *SeqConc*:

68. This function applies the 'built-in' binary sequence concatenation operator of two values (if they are both sequence values).

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.4.8 Sequence and Map Modification Operation

```
69.    MapOrSeqMod : VAL × VAL → VAL
69.1   MapOrSeqMod(l_val, r_val) ≜
  .2     if (l_val ∈ LIST_VAL) ∧ (r_val ∈ MAP_VAL)
  .3     then let l_v = StripSeqTagVal(l_val),
  .4               r_v = StripMapTagVal(r_val) in
  .5            if dom(r_v) ⊆ inds(l_v)
  .6            then let den = [if i ∈ dom(l_v)
  .7                            then r_v(i)
  .8                            else l_v(i)
  .9                           | i ∈ inds(l_v)] in
  .10                  if ⊥ ∈ elems(den)
  .11                  then ⊥
  .12                  else MkTag('seq', den)
  .13           else ⊥
  .14    else if (l_val ∈ MAP_VAL) ∧ (r_val ∈ MAP_VAL)
  .15         then let l_v = StripMapTagVal(l_val),
  .16                  r_v = StripMapTagVal(r_val) in
  .17                MkTag('map', overwrite(l_v, r_v))
  .18         else ⊥
```

Annotations to *MapOrSeqMod*:

69. This function applies the 'built-in' binary map or sequence modifier operator to two values (if the first value is either a sequence or a map, and the second value is a map value).

.3–.15 Sequence modification is applied if the first operand is a sequence value, and the second operand is a map value.

.5 Notice that it is required that all modifications take place for existing indices in the sequence value. Thus, if an attempt is made to modify a sequence outside its set of indices, bottom is returned.

.17–.20 Map modification (or map overwrite) is applied if both operands are map values.

End of annotations

8.3.4.9 Map Operations

```
70.    MapMerge : VAL × VAL → VAL
70.1   MapMerge(l_val, r_val) ≜
  .2     if (l_val ∈ MAP_VAL) ∧ (r_val ∈ MAP_VAL)
  .3     then let l_v = StripMapTagVal(l_val),
  .4               r_v = StripMapTagVal(r_val) in
  .5            if Compatible({ l_v, r_v})
  .6            then MkTag('map', merge(l_v, r_v))
  .7            else ⊥
  .8     else ⊥
```

Annotations to *MapMerge*:

70. This function applies the 'built-in' binary map merge operator to two values (if both values are map values, and they are compatible).

.5–.6 If the two map values are compatible, the map merge operator is applied

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

71. $MapToSetOpr : \{ \text{MAPDOMRESTRTO, MAPDOMRESTRBY, MAPRNGRESTRTO, MAPRNGRESTRBY} \} \times VAL \times VAL \to VAL$

71.1 $MapToSetOpr(bin_op, l_val, r_val) \triangleq$
.2 **if** $(l_val \in MAP_VAL) \wedge (r_val \in SET_VAL)$
.3 **then let** $l_v = StripMapTagVal(l_val)$,
.4 $r_v = StripSetTagVal(r_val)$ **in**
.5 **cases** bin_op :
.6 MAPDOMRESTRTO $\to$ $MkTag(\text{'map'}, \underline{\text{restrict}}(l_v, r_v))$,
.7 MAPDOMRESTRBY $\to$ $MkTag(\text{'map'}, \underline{\text{subtract}}(l_v, r_v))$,
.8 MAPRNGRESTRTO $\to$ $MkTag(\text{'map'}, \{ a \mapsto l_v(a)$
.9 $\mid a \in \underline{\text{dom}}(l_v) \cdot l_v(a) \in r_v \})$,
.10 MAPRNGRESTRBY $\to$ $MkTag(\text{'map'}, \{ a \mapsto l_v(a)$
.11 $\mid a \in \underline{\text{dom}}(l_v) \cdot l_v(a) \notin r_v \})$
.12 **else** $\perp$

Annotations to *MapToSetOpr*:

71. This function applies the 'built-in' binary map operators to two values (the first value is a map value, and the second value is a set value).

.6–.11 The different operators are applied. They restrict the domain (and range) of the map, l_v to the set r_v, and subtract the elements in common with the set from the domain (and range) of the map, respectively.

End of annotations

8.3.4.10 Compose and Iterate

72. $Compose : VAL \times VAL \to VAL$

72.1 $Compose(l_v, r_v) \triangleq$
.2 **if** $l_v \in FUN_VAL \wedge r_v \in FUN_VAL$
.3 **then let** $MkTag(\text{'fun'}, f) = l_v$,
.4 $MkTag(\text{'fun'}, g) = r_v$ **in**
.5 **if** $\delta_1(g) \subseteq \delta_0(f)$
.6 **then** $MkTag(\text{'fun'}, f \circ g)$
.7 **else** $\perp$
.8 **else if** $l_v \in MAP_VAL \wedge r_v \in MAP_VAL$
.9 **then let** $m_1 = StripMapTagVal(l_v)$,
.10 $m_2 = StripMapTagVal(r_v)$ **in**
.11 **if** $\underline{\text{rng}}(m_2) \subseteq \underline{\text{dom}}(m_1)$
.12 **then** $MkTag(\text{'map'}, \{ i \mapsto m_1(m_2(i)) \mid i \in \underline{\text{dom}}(m_2) \})$
.13 **else** $\perp$
.14 **else** $\perp$

Annotations to *Compose*:

72. This function applies the 'built-in' compose operator to two values. This operator is overloaded in the sense that it can be used for composition of functions and for composition of maps. Thus the two values must either be function values, or map values.

.3–.7 Function composition can be explained by:
$f_1 \circ f_2 = f_1(f_2(x))$
$f_1 \in A \to B \wedge f_2 \in C \to D$

.5 Recall that δ_0 is the domain operator on general functions, and that δ_1 is the range operator on general functions (as described page 8).

.5 Condition: $D \subseteq A$

.6 Result: $f_1 \circ f_2 \in C \to B$

.9–.13 Map composition can be explained by:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$m_1 \circ m_2 = m_1(m_2(x))$
$m_1 \in \mathbb{IM}(A, B) \wedge m_2 \in \mathbb{IM}(C, D)$
.11 Condition: $D \subseteq A$
.12 Result: $m_1 \circ m_2 \in \mathbb{IM}(C, B)$

End of annotations

73. *Iterate* : $VAL \times VAL \rightarrow VAL$

73.1 $Iterate(l_v, r_v) \triangleq$
.2 **if** $l_v \in FUN_VAL \wedge r_v \in NUM_VAL$
.3 **then let** $MkTag(\text{'fun'}, f) = l_v,$
.4 $MkTag(\text{'num'}, n) = r_v$ **in**
.5 **if** $n \in \mathbb{N} \wedge (n \geq 2 \Rightarrow \delta_1(f) \subseteq \delta_0(f))$
.6 **then** $MkTag(\text{'fun'}, IterateFct(f, n))$
.7 **else** $\perp$
.8 **else if** $l_v \in MAP_VAL \wedge r_v \in NUM_VAL$
.9 **then let** $m = StripMapTagVal(l_v),$
.10 $n = StripNumTagVal(r_v)$ **in**
.11 **if** $n \in \mathbb{N} \wedge (n \geq 2 \Rightarrow \underline{\text{rng}}(m) \subseteq \underline{\text{dom}}(m))$
.12 **then** $MkTag(\text{'map'}, IterateMap(m, n))$
.13 **else** $\perp$
.14 **else if** $l_v \in NUM_VAL \wedge r_v \in NUM_VAL$
.15 **then let** $l_v = StripNumTagVal(l_val),$
.16 $r_v = StripNumTagVal(r_val)$ **in**
.17 **if** $(l_v > 0) \vee$
.18 $(l_v = 0 \wedge r_v > 0) \vee$
.19 $(l_v < 0 \wedge OddDenom(r_v))$
.20 **then** $MkTag(\text{'num'}, l_v^{r_v})$
.21 **else** $\perp$
.22 **else** $\perp$

Annotations to *Iterate*:

73. This function applies the 'built-in' iterate operator to two values. This operator is overloaded in the sense that it can be used for iteration of functions and maps, and also as a numeric exponentiation operator. Thus the second value must always be a numeric value, while the first value should either be a function value, a map value or a numeric value.

.3–.7 Function iteration can be explained by:
$f^n(x) = \underbrace{f(f(f \ldots f}_{\text{n times}}(x) \ldots)))$
where $f \in A \rightarrow B$

.5 Recall that δ_0 is the domain operator on general functions, and that δ_1 is the range operator on general functions (as described page 8).

.5 Condition: $B \subseteq A$
.6 Result: $f^n \in A \rightarrow B$

.9–.13 Map iteration can be explained by:
$m^n(x) = \underbrace{m(m(m \ldots m}_{\text{n times}}(x) \ldots)))$
where $m \in \mathbb{IM}(A, B)$

.11 Condition: $n \in \mathbb{N} \wedge (n \geq 2 \Rightarrow B \subseteq A)$
.12 Result: $m^n \in \mathbb{IM}(A, B)$

.15–.21 Numeric exponentiation is only defined in the cases:
— $l_v > 0$, all numeric values of r_v; in this case the result will always be greater than 0.
— $l_v = 0$, $r_v > 0$; in this case the result will always be 0.
— $l_v < 0$, $r_v \in \mathbb{Z}$

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

74. $IterateFct : (A \to B) \times \mathbb{N} \to (A \to B) \cup \{ \bot \}$

74.1 $IterateFct(f, n) \triangleq$
.2 **if** $n \geq 1 \wedge (n \geq 2 \Rightarrow \delta_1(f) \subseteq \delta_0(f))$
.3 **then** $f \circ (IterateFct(f, n - 1))$
.4 **else if** $n = 0$
.5 **then** $\lambda x \in \delta_0(f) \cdot x$
.6 **else** $\bot$

Annotations to *IterateFct*:
74. This auxiliary function performs iteration of the given function, f, the given number of times, n.
.3 $f^n = f(f^{n-1})$
.5 $f^0 = \lambda x \in \delta_0(f) \cdot x$, i.e. the identity function on the domain of f. Thus, in general $f^0 \neq g^0$, because f and g can have different domains.

End of annotations

75. $IterateMap : \mathbb{M}(A, B) \times \mathbb{N} \to \mathbb{M}(A, B) \cup \{ \bot \}$

75.1 $IterateMap(map, n) \triangleq$
.2 **if** $n \geq 1 \wedge (n \geq 2 \Rightarrow \underline{\text{rng}}(m) \subseteq \underline{\text{dom}}(m))$
.3 **then** $\{ i \mapsto map(IterateMap(map, n - 1)(i)) \mid i \in \underline{\text{dom}}(map) \}$
.4 **else if** $n = 0$
.5 **then** $\{ a \mapsto a \mid a \in \underline{\text{dom}}(map) \}$
.6 **else** $\bot$

Annotations to *IterateMap*:
75. This auxiliary function performs iteration of the given map, *map*, the given number of times, n.
.3 $map^n = map(map^{n-1})$
.4 $map^0 = \{ a \mapsto a \mid a \in \underline{\text{dom}}(map) \}$ i.e. the identity map on the domain of *map*.

End of annotations

76. $OddDenom$: **Pred**($\mathbb{R}$)

76.1 $OddDenom(r) \triangleq$
.2 $\exists m, n \in \mathbb{N} \cdot GCD(m, n) = 1 \wedge IsOdd(n) \wedge$
.3 **if** $r > 0$
.4 **then** $r = m / n$
.5 **else** $r = -m / n$

Annotations to *OddDenom*:
76. This predicate checks whether a real number can be described with an odd denominator.

End of annotations

77. $GCD : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$

77.1 $GCD(m, n) \triangleq$
.2 $Iota(\{ d \in \mathbb{N} \mid m / d \in \mathbb{N} \wedge n / d \in \mathbb{N} \wedge$
.3 $\forall r \in \mathbb{N} \cdot m / r \in \mathbb{N} \wedge n / r \in \mathbb{N} \Rightarrow r \leq d \})$

Annotations to *GCD*:
77. This function returns the greatest common divisor for the two arguments.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

78. *IsOdd* : **Pred**(𝕀N)

78.1 *IsOdd*(*n*) ≜
.2 $\exists m \in \mathbb{N} \cdot (n+1) = 2 \times m$

Annotations to *IsOdd*:
78. This predicate simply checks whether a natural number is odd.
End of annotations

8.3.5 Quantified Expressions

79. *EvalAllOrExistsExpr* : *AllOrExistsExpr* → *LEEval*

79.1 *EvalAllOrExistsExpr*(*MkTag*('*AllOrExistsExpr*', (*quant*, *bind*, *pred*))) ≜
.2 **cases** *quant* :
.3 ALL → *EvalForAll*(*bind*, *pred*),
.4 EXISTS → *EvalExists*(*bind*, *pred*)

Annotations to *EvalForAll*:
79. This function gives meaning to 'traditional' quantified expressions (i.e. universal quantified expressions and existential quantified expressions).
End of annotations

80. *EvalForAll* : 𝕀F(*Bind*) × *Expr* → *LEEval*

80.1 *EvalForAll*(*bind_s*, *pred*) ≜
.2 **let** *pbev_s* = *Partition*({ *EvalBind*(*b*) | *b* ∈ *bind_s* }) **in**
.3 { λ*env*.**let** *venv_ss* = { *bev*(*env*) | *bev* ∈ *pbev* } **in**
.4 **if** { } ∈ *venv_ss* ∨ ∃*venv_s* ∈ *venv_ss* · err ∈ *venv_s*
.5 **then if** { } ∈ *venv_ss*
.6 **then** *True*()
.7 **else** ⊥
.8 **else let** *venvs* = { Merge(*envs*) | *envs* ∈ *Partition*(*venv_ss*) ·
.9 *Compatible*(*envs*) } **in**
.10 **if** ∃*venv* ∈ *venvs* · *ev*(overwrite(*env*, *venv*)) ∉ $BOOL_VAL_{\perp}$
.11 **then** ⊥
.12 **else if** ∀*venv* ∈ *venvs* · *ev*(overwrite(*env*, *venv*)) = *True*()
.13 **then** *True*()
.14 **else if** ∃*venv* ∈ *venvs* · *ev*(overwrite(*env*, *venv*)) = *False*()
.15 **then** *False*()
.16 **else** ⊥
.17 | *ev* ∈ *EvalExpr*(*pred*), *pbev* ∈ *pbev_s* }

Annotations to *EvalForAll*:
80. This function takes a binding and a predicate, and tests whether all possible bindings make the evaluation of the predicate true. Thus, it is used to give meaning to universal quantified expressions.
.3–.16 For all predicate evaluations the corresponding "for all" evaluator is created.
.3 All possible sets of value environments are collected.
.5–.6 If the empty set belongs to *venv_ss* it corresponds to the situation where one of the bindings has completely failed. Thus, no bindings have succeeded and therefore the predicate is satisfied 'for all of them' and *True* is returned.
.7 A 'real' error occurred in the binding and bottom must be returned.
.8–.9 All possible binding environments are constructed by performing a distributed merge for each set of environments from *venv_ss*.

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.10–.11 If the predicate evaluator applied to the given environment extended by any of the possible bindings does not return either a Boolean value, or bottom, the whole quantification is ill-typed and bottom is returned.

.12–.13 If the predicate evaluator applied to the given environment extended by a possible binding always returns true, then true is returned.

.14–.16 If the predicate evaluator applied to the given environment extended by one of the possible bindings returns false, then false is returned. Otherwise bottom is returned.

.17 This is done for all possible predicate evaluators and for all possible sets of binding evaluators (each corresponding to a model).

End of annotations

```
81.   EvalExists : IF(Bind) × Expr → LEEval
81.1  EvalExists(bind_s, pred) ≙
  .2    let pbev_s = Partition({ EvalBind(b) | b ∈ bind_s }) in
  .3      { λenv.let venv_ss = { bev(env) | bev ∈ pbev } in
  .4               if { } ∈ venv_ss ∨ ∃venv_s ∈ venv_ss · err ∈ venv_s
  .5               then if { } ∈ venv_ss
  .6                    then False()
  .7                    else ⊥
  .8               else let venvs = { Merge(envs) | envs ∈ Partition(venv_ss) ·
  .9                                                 Compatible(envs) } in
  .10                   if ∃venv ∈ venvs · ev(overwrite(env, venv)) ∉ BOOL_VAL⊥
  .11                   then ⊥
  .12                   else if ∃venv ∈ venvs · ev(overwrite(env, venv)) = True()
  .13                        then True()
  .14                        else if ∀venv ∈ venvs · ev(overwrite(env, venv)) = False()
  .15                             then False()
  .16                             else ⊥
  .17     | ev ∈ EvalExpr(pred), pbev ∈ pbev_s }
```

Annotations to *EvalExists*:

81. This function takes a binding and a predicate, and tests whether any binding exists which makes the evaluation of the predicate true. Thus, it is used to give meaning to existential quantified expressions.

.3–.16 For all predicate evaluations the corresponding "exists" evaluator is created.

.3 All possible sets of value environments are collected.

.5–.6 If the empty set belongs to *venv_ss* it corresponds to the situation where one of the bindings have completely failed. Thus, no bindings have succeeded and therefore there does not exist any binding for which the predicate is satisfied, and *False* is returned.

.7 A 'real' error occurred in the binding and bottom must be returned.

.8–.9 All possible binding environments are constructed by performing a distributed merge for each set of environments from *venv_ss*.

.10–.11 If the predicate evaluator applied with the given environment extended by any of the possible bindings does not return either a Boolean value or bottom, the whole quantification is ill-typed and bottom is returned.

.12–.13 If the predicate evaluator applied with the given environment extended by one of the possible bindings returns true, then true is returned.

.14–.16 If the predicate evaluator applied with the given environment extended by a possible binding always returns false, then false will be returned. Otherwise bottom will be returned.

.17 This is done for all possible predicate evaluators and for all possible sets of binding evaluators (each corresponding to a model).

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

An example with a unique quantified expression

Before defining with the semantics of the unique quantified expression two small examples are given, illustrating the complexity which needs to be taken into account here. The first example is a simple one which illustrates how a deterministic unique quantified expression is handled. In contrast to this, the second example includes looseness in all possible places. Thus, this example is only useful for illustrating the semantics of such combinations of looseness.

The concrete syntax for the first example looks like:

$simple = \exists!\ a \in \{1,2,3\} \cdot a = 2$

The set expression $\{1,2,3\}$ simply has one expression evaluator:

$$EvalExpr(\{1,2,3\}) =$$
$$\{\, \lambda env\ .\ MkTag(\text{'set'}, \{\, MkTag(\text{'num'}, 1),\ MkTag(\text{'num'}, 2),\ MkTag(\text{'num'}, 3)\,\})\,\}$$

The bindings for which the predicate '$a = 2$' must be checked are:

$$EvalBind(a \in \{1,2,3\}) =$$
$$\{\, \lambda env\ .\ \{\, \{\, a \mapsto MkTag(\text{'num'}, 1)\,\},$$
$$\{\, a \mapsto MkTag(\text{'num'}, 2)\,\},$$
$$\{\, a \mapsto MkTag(\text{'num'}, 3)\,\}\,\}$$

The point now is that this binding evaluator contains exactly one binding which makes the predicate '$a = 2$' true. Thus, the semantics of *simple* is:

$$EvalExpr(simple) = \{\, \lambda env\ .\ MkTag(\text{'bool'}, \mathsf{T})\,\}$$

The concrete syntax for the second (and much more complicated) example looks like:

$$unique = \exists!\ l_1 \frown l_2 \in (\mathsf{let}\ s \in \{\{[4],[2,5],7\},\{[2],[3],[7,4]\}\}\ \mathsf{in}\ s) \cdot$$
$$\mathsf{let}\ x \in \{2,4\}\ \mathsf{in}\ x \in (\mathsf{elems}\ l_1 \cup \mathsf{elems}\ l_2)$$

This is a unique quantified expression, where the first line contains the exists unique quantifier and a set binding, while the second line contains a predicate. The set binding consists of a sequence concatenation pattern, and a let-be-such-that expression producing a set value. All three components are loosely specified, in the sense that they have multiple evaluators. The entire expression is identical to true, and therefore $EvalExpr(unique) = \{\, \lambda env\ .\ True()\,\}$. A presentation will be given below of what the semantics of the components (*bind* and *pred*) is, in order to explain why *unique* is identical to true.

The set expression in the binding has two evaluators:

$$EvalExpr(set) =$$
$$\{\, \lambda env\ .\ \{\,[4],\,[2,\,5],\,7\,\},$$
$$\lambda env\ .\ \{\,[2],\,[3],\,[7,\,4]\,\}\,\}$$

In principle all the set values, the sequence values, and the numeric values should be tagged by *'set'*, *'seq'*, and *'num'*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

as is done in the domain universe, but for simplicity all these tags are omitted here[1)] (this will also be done in the following formulae).

These two set evaluators can be combined with the sequence concatenation pattern $l_1 \frown l_2$ in 6 and 9 ways respectively. The semantics of the binding will thus be (see *EvalBind* for an explanation of binding denotations):

$$
\begin{aligned}
&\mathit{EvalBind}(\mathit{bind}) = \\
&\{\lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [4]\}, \{l_1 \mapsto [\,], l_2 \mapsto [2,5]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [4]\}, \{l_1 \mapsto [2], l_2 \mapsto [5]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [4]\}, \{l_1 \mapsto [2,5], l_2 \mapsto [\,]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [4], l_2 \mapsto [\,]\}, \{l_1 \mapsto [\,], l_2 \mapsto [2,5]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [4], l_2 \mapsto [\,]\}, \{l_1 \mapsto [2], l_2 \mapsto [5]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [4], l_2 \mapsto [\,]\}, \{l_1 \mapsto [2,5], l_2 \mapsto [\,]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [2]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}, \{l_1 \mapsto [\,], l_2 \mapsto [7,4]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [2]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}, \{l_1 \mapsto [7], l_2 \mapsto [4]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [2]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}, \{l_1 \mapsto [7,4], l_2 \mapsto [\,]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [2]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}, \{l_1 \mapsto [\,], l_2 \mapsto [7,4]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [2]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}, \{l_1 \mapsto [7], l_2 \mapsto [4]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [\,], l_2 \mapsto [2]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}, \{l_1 \mapsto [7,4], l_2 \mapsto [\,]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [2], l_2 \mapsto [\,]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}, \{l_1 \mapsto [\,], l_2 \mapsto [7,4]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [2], l_2 \mapsto [\,]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}, \{l_1 \mapsto [7], l_2 \mapsto [4]\}\}, \\
&\ \lambda \mathit{env} \,.\, \{\{l_1 \mapsto [2], l_2 \mapsto [\,]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}, \{l_1 \mapsto [7,4], l_2 \mapsto [\,]\}\}\}
\end{aligned}
$$

Note that it is impossible to match the value 7 against the sequence concatenation pattern, and that this value therefore is incapable of giving possible matchings. Notice that the set of bindings returned in each binding evaluator has the same cardinality as the set value the pattern is being matched against (where the pattern matching succeeds against the element values). Thus, these sets represents possible ways of matching the pattern with a particular possible set value.

The predicate is also loosely specified; it has two expression evaluators:

$$
\begin{aligned}
&\mathit{EvalExpr}(\mathit{pred}) = \\
&\{\lambda \mathit{env} \,.\, \textbf{if } l_1 \in \underline{\mathrm{dom}}(\mathit{env}) \wedge l_2 \in \underline{\mathrm{dom}}(\mathit{env}) \\
&\qquad \textbf{then } 2 \in (\underline{\mathrm{elems}}(\mathit{env}(l_1)) \cup \underline{\mathrm{elems}}(\mathit{env}(l_2))) \\
&\qquad \textbf{else } \bot, \\
&\ \lambda \mathit{env} \,.\, \textbf{if } l_1 \in \underline{\mathrm{dom}}(\mathit{env}) \wedge l_2 \in \underline{\mathrm{dom}}(\mathit{env}) \\
&\qquad \textbf{then } 4 \in (\underline{\mathrm{elems}}(\mathit{env}(l_1)) \cup \underline{\mathrm{elems}}(\mathit{env}(l_2))) \\
&\qquad \textbf{else } \bot\}
\end{aligned}
$$

The point now is that for each of these two predicate evaluators exactly one binding exists in each of the binding evaluators, which makes the predicate true. Thus, the entire expression is identical to true. It is worth noting that it is true even though some of the pattern matchings failed. Failed pattern matchings are simply ignored. This also means that it is possible to quantify over the subset of the type/set value which matches the pattern in the binding (this can also be done for universal and existential quantification).

When a unique quantified expression is true, it is possible to select the 'unique' binding value by means of an iota

[1)] As an example of how much more complicated the expressions would look the set from the first expression evaluator given above can be described as:

MkTag('*set*', { *MkTag*('*seq*', [*MkTag*('*num*', 4)]),
MkTag('*seq*', [*MkTag*('*num*', 2), *MkTag*('*num*', 5)]),
MkTag('*num*', 7) })

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

expression with the same binding and the same predicate as the "unique quantified"-expression. Consider:

$$iota = \iota\, l_1 \curvearrowright l_2 \in (\textsf{let } s \in \{\{[4],[2,5],7\},\{[2],[3],[7,4]\}\} \textsf{ in } s) \cdot$$
$$\textsf{let } x \in \{2,4\} \textsf{ in } x \in (\textsf{elems } l_1 \cup \textsf{elems } l_2)$$

where the exists unique quantifier has been replaced by iota from *unique*.

The interesting point here is that even though the unique quantified expression yields true, there are three different 'unique' values because of the looseness.

Inside the definition of the function called *EvalIotaExpr* the pattern is transformed into a corresponding expression and evaluated (see line 13). In this case the evaluation yields one evaluator:

$$EvalExpr(GetExpr(SelPattern(bind))) =$$
$$\{\lambda env\ .\ \textbf{if } l_1 \in \underline{\text{dom}}(env) \wedge l_2 \in \underline{\text{dom}}(env)$$
$$\textbf{then } \underline{\text{join}}(env(l_1),\ env(l_2))$$
$$\textbf{else } \bot\ \}$$

A test is then made that for each of the binding evaluators only one value is returned (which is then considered to be 'unique' to that binding evaluator). In this case the evaluation of the entire iota expression yields:

$$EvalExpr(iota) =$$
$$\{\lambda env.\ [2,\ 5],\ \lambda env.\ [2],\ \lambda env.\ [4],\ \lambda env.\ [7,\ 4]\ \}$$

This completes the example, which can be read with advantage in parallel to understanding the semantics of the unique quantified expression and the iota expression.

□

```
82.   EvalExistsUniqueExpr : ExistsUniqueExpr → LEEval
82.1  EvalExistsUniqueExpr(MkTag('ExistsUniqueExpr', (bind, pred))) ≙
  .2    let lbev = EvalBind(bind) in
  .3      PropE({ λenv.{ if b(env) = { } ∨ err ∈ b(env)
  .4                     then if { } = b(env)
  .5                          then False()
  .6                          else ⊥
  .7                     else if ∃venv ∈ b(env) · ev(overwrite(env, venv)) ∉BOOL_VAL
  .8                          then ⊥
  .9                          else if ∃!venv ∈ b(env) · ev(overwrite(env, venv)) = True()
  .10                              then True()
  .11                              else False()
  .12                | b ∈ lbev}
  .13           | ev ∈ EvalExpr(pred) })
```

Annotations to *EvalExistsUniqueExpr*:

82. This function gives meaning to unique (existential) quantified expressions.

.3 – .12 For all predicate evaluations the corresponding 'exists unique' evaluator is created.

.4 – .5 If the binding evaluator in the given environment yields an empty set it corresponds to the situation where the binding has completely failed. Thus, there does not exist a unique binding for which the predicate is satisfied, and *False* is returned.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.6 A 'real' error occurred in the binding and bottom must be returned.
.7–.8 If the predicate evaluator applied with the given environment extended by any of the possible bindings does not return a Boolean value the whole quantification is ill-typed and bottom is returned.
.9–.11 If exactly one of the possible bindings makes the predicate evaluator applied with the given environment extended with that binding return true, then true is returned. Otherwise false is returned.
.13 This is done for all possible predicate evaluators.

End of annotations

8.3.6 Iota Expression

```
83.   EvalIotaExpr : IotaExpr → LEEval
83.1  EvalIotaExpr(MkTag('IotaExpr', (bind, pred))) ≙
 .2    let lbev = EvalBind(bind) in
 .3      PropE({ λenv.{ if b(env) = { } ∨ err ∈ b(env)
 .4                     then ⊥
 .5                     else if ∃ venv ∈ b(env) · ev(overwrite(env, venv)) ∉ BOOL_VAL
 .6                          then ⊥
 .7                          else let s = { eev(overwrite(env, venv))
 .8                                       | venv ∈ b(env) ·
 .9                                         ev(overwrite(env, venv)) = True() } in
 .10                                   if card(s) = 1
 .11                                   then Iota(s)
 .12                                   else ⊥
 .13                  | b ∈ lbev, eev ∈ EvalExpr(GetExpr(SelPattern(bind))) }
 .14           | ev ∈ EvalExpr(pred) })
```

Annotations to *EvalIotaExpr*:

83. This function gives meaning to an iota expression (an expression which returns the unique element which fulfils an expression like $\exists !x \in X \,.\, pred$).
.3–.13 For all predicate evaluations the corresponding 'iota' evaluator is created.
.3–.4 If the binding evaluator in the given environment yields an empty set it corresponds to the situation where the binding has completely failed. Thus, there does not exist a unique binding for which the predicate is satisfied, and bottom is returned. This is also the case if a 'real' error occurred in the binding.
.5–.6 If the predicate evaluator applied with the given environment extended by any of the possible bindings does not return a Boolean value, the whole iota expression is ill-typed and bottom is returned.
.7–.9 A set is created where an evaluator for the expression, which corresponds to the pattern in the binding, is used in a binding environment where a predicate evaluator yields true. This is supposed to be a singleton set containing the 'unique' element fulfilling the predicate.
.10–.12 If *s* shows that it really is a unique element, the element is returned. Otherwise (if it is not unique), bottom is returned.
.13 This is done for all possible successful binding evaluators, and for all evaluators for the expression corresponding to the pattern from *bind*.
.14 This is done for all possible predicate evaluators.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.7 Set Expressions

84. $EvalSetEnumeration : SetEnumeration \rightarrow LEEval$

84.1 $EvalSetEnumeration(MkTag(\text{'SetEnumeration'}, els)) \triangleq$
.2 **let** $el_lev_l = [\, EvalExpr(els(i)) \mid i \in \underline{\text{inds}}(els)\,]$ **in**
.3 **let** $seq_lev_s = \{\, ev_l \mid ev_l \in \mathbb{L}(EEval) \cdot$
.4 $\underline{\text{len}}(ev_l) = \underline{\text{len}}(el_lev_l) \wedge$
.5 $\forall i \in \underline{\text{inds}}(ev_l) \cdot ev_l(i) \in el_lev_l(i)\,\}$ **in**
.6 $\{\, \lambda env$. **let** $setden = \{\, ev(env) \mid ev \in \underline{\text{elems}}(ev_l)\,\}$ **in**
.7 **if** $\forall el \in setden \cdot el \in FLATVAL \backslash \{\, \perp\,\}$
.8 **then** $MkTag(\text{'set'}, setden)$
.9 **else** $\perp$
.10 $\mid ev_l \in seq_lev_s\}$

Annotations to *EvalSetEnumeration*:

84. This function gives meaning to set enumeration expressions. A *SetEnumeration* expression looks like: $\{els\}$ where *els* is a list of expressions separated by commas.
.2 The list of all element loose evaluators is defined.
.3 – .5 The set of all lists of (set element) evaluators is defined implicitly by using all possible combinations of the element evaluators.
.6 – .10 For all sequences of (set element) evaluators, the evaluator for the set enumeration is created.
.7 – .9 If all elements are proper flat values from the domain universe, the set is tagged and returned. Otherwise bottom is returned.

End of annotations

An example with set comprehension

Before proceeding with the semantics of an expression using set comprehension, two small examples are shown illustrating the complexity which needs to be taken into account here. The first example is a simple one which illustrates how deterministic set comprehension is handled. In contrast to this, the second example includes looseness in all possible places. Thus, this example is only useful for illustrating the semantics of such combinations of looseness.

The concrete syntax for the first example looks like:

$simple = \{a \mid a \in \{1, \ldots, 5\} \cdot a \text{ rem } 2 = 0\}$

This is a set comprehension expression which has three components; an element expression, a binding, and a predicate. The element expression is simply a, the binding is $a \in \{1, \ldots, 5\}$, while the predicate is $a \text{ rem } 2 = 0$.

The set range expression, $\{1, \ldots, 5\}$, has one expression evaluator:

$$
\begin{aligned}
&EvalExpr(\{1, \ldots, 5\}) = \\
&\{\, \lambda env\ .\ MkTag(\text{'set'}, \{\, MkTag(\text{'num'}, 1),\\
&\qquad\qquad\qquad\qquad MkTag(\text{'num'}, 2),\\
&\qquad\qquad\qquad\qquad MkTag(\text{'num'}, 3),\\
&\qquad\qquad\qquad\qquad MkTag(\text{'num'}, 4),\\
&\qquad\qquad\qquad\qquad MkTag(\text{'num'}, 5)\,\})\,\}
\end{aligned}
$$

The predicate will ensure that only even numbers are considered. Thus, the semantics of *simple* is:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$EvalExpr(simple) =$$
$$\{\, \lambda env\ .\ MkTag(\text{'set'},\ \{\, MkTag(\text{'num'},\ 2),\ MkTag(\text{'num'},\ 4)\,\})\,\}$$

The concrete syntax for the second (and much more complicated) example looks like:

$$\begin{aligned} set = \{&\textsf{let } e \in \{1,2\} \textsf{ in } mk\text{-}(e, i, l_1) \\ &\mid i \in (\textsf{let } s \in \{\{4,8,3\},\{5\}\} \textsf{ in } s), l_1 \frown l_2 \in \{[7,4],[3],7\} \cdot \\ &\qquad \textsf{let } x \in \{4,6\} \textsf{ in } i \leq x\} \end{aligned}$$

This is also a set comprehension expression which has three components; an element expression, a collection of bindings, and a predicate. The first line here contains the element expression, and the second line contains two set bindings followed by a predicate in the third line. There is looseness involved in all these components, in the sense that they all have multiple evaluators. A presentation is given below of the semantics of the components of the set comprehension, leading to an understanding of the semantics of the entire expression, *set*.

The semantics of the first binding is:

$$EvalBind(i \in (\textsf{let } s \in \{\{4,8,3\},\{5\}\} \textsf{ in } s)) =$$
$$\{\, \lambda env\ .\ \{\{\, i \mapsto 4\,\},\ \{\, i \mapsto 8\,\},\ \{\, i \mapsto 3\,\}\},$$
$$\lambda env\ .\ \{\{\, i \mapsto 5\,\}\}\,\}$$

while the semantics of the second binding is:

$$EvalBind(l_1 \frown l_2 \in \{[7,4],[3],7\}) =$$
$$\{\, \lambda env\ .\ \{\{\, l_1 \mapsto [\,],\ l_2 \mapsto [7,\ 4]\,\},\ \{\, l_1 \mapsto [\,],\ l_2 \mapsto [3]\,\}\},$$
$$\lambda env\ .\ \{\{\, l_1 \mapsto [7],\ l_2 \mapsto [4]\,\},\ \{\, l_1 \mapsto [\,],\ l_2 \mapsto [3]\,\}\},$$
$$\lambda env\ .\ \{\{\, l_1 \mapsto [7,\ 4],\ l_2 \mapsto [\,]\,\},\ \{\, l_1 \mapsto [\,],\ l_2 \mapsto [3]\,\}\},$$
$$\lambda env\ .\ \{\{\, l_1 \mapsto [\,],\ l_2 \mapsto [7,\ 4]\,\},\ \{\, l_1 \mapsto [3],\ l_2 \mapsto [\,]\,\}\},$$
$$\lambda env\ .\ \{\{\, l_1 \mapsto [7],\ l_2 \mapsto [4]\,\},\ \{\, l_1 \mapsto [3],\ l_2 \mapsto [\,]\,\}\},$$
$$\lambda env\ .\ \{\{\, l_1 \mapsto [7,\ 4],\ l_2 \mapsto [\,]\,\},\ \{\, l_1 \mapsto [3],\ l_2 \mapsto [\,]\,\}\}\,\}$$

Note that even though it is impossible to match the value 7 against the sequence concatenation pattern, no errors are indicated. If the same pattern ($l_1 \frown l_2$) was matched against the value 7, *EvalPattern* would yield unmatch but such situations are ignored by the *EvalBind* function. In principle all values should be tagged as by *'set'*, *'seq'*, *'num'* etc. as is done in the domain universe, but for simplicity all these tags are omitted here (this will also be done in the following formulae).

In the semantics of a *SetComprehension* these binding evaluators are combined in all possible ways by means of *Partition*. This looks like:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$
\begin{array}{l}
\mathit{Partition}(\{\, \mathit{bind}_1,\ \mathit{bind}_2 \,\}) = \\
\{\{\lambda \mathit{env}\ .\ \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [\,], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7, 4], l_2 \mapsto [\,]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [\,], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7, 4], l_2 \mapsto [\,]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 5\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [\,], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 5\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 5\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7, 4], l_2 \mapsto [\,]\}, \{l_1 \mapsto [\,], l_2 \mapsto [3]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 5\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [\,], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 5\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}\}\}, \\
\ \{\lambda \mathit{env}\ .\ \{\{i \mapsto 5\}\}, \\
\quad \lambda \mathit{env}\ .\ \{\{l_1 \mapsto [7, 4], l_2 \mapsto [\,]\}, \{l_1 \mapsto [3], l_2 \mapsto [\,]\}\}\}\}
\end{array}
$$

This is a set of sets of binding evaluators which are used, set by set, for producing the semantics of the set comprehension expression, *set*. Before the semantics of *set* can be determined it is necessary to find the semantics of the element expression, and the predicate expression. Firstly, the semantics of the element expression:

$$
\begin{array}{l}
\mathit{EvalExpr}(\mathsf{let}\ e \in \{1,2\}\ \mathsf{in}\ \mathit{mk}\text{-}(e, i, l_1)) = \\
\{\lambda \mathit{env}\ .\mathbf{if}\ i \in \underline{\mathrm{dom}}(\mathit{env}) \wedge l_1 \in \underline{\mathrm{dom}}(\mathit{env}) \\
\qquad \mathbf{then}\ (1,\ \mathit{env}(i),\ \mathit{env}(l_1)) \\
\qquad \mathbf{else}\ \bot, \\
\ \lambda \mathit{env}\ .\mathbf{if}\ i \in \underline{\mathrm{dom}}(\mathit{env}) \wedge l_1 \in \underline{\mathrm{dom}}(\mathit{env}) \\
\qquad \mathbf{then}\ (2,\ \mathit{env}(i),\ \mathit{env}(l_1)) \\
\qquad \mathbf{else}\ \bot\,\}
\end{array}
$$

and then the semantics of the predicate looks like:

$$
\begin{array}{l}
\mathit{EvalExpr}(\mathsf{let}\ x \in \{4,6\}\ \mathsf{in}\ i \leq x) = \\
\{\lambda \mathit{env}\ .\mathbf{if}\ i \in \underline{\mathrm{dom}}(\mathit{env}) \\
\qquad \mathbf{then}\ \mathit{env}(i) \leq 4 \\
\qquad \mathbf{else}\ \bot, \\
\ \lambda \mathit{env}\ .\mathbf{if}\ i \in \underline{\mathrm{dom}}(\mathit{env}) \\
\qquad \mathbf{then}\ \mathit{env}(i) \leq 6 \\
\qquad \mathbf{else}\ \bot\,\}
\end{array}
$$

The semantics of the entire set comprehension expression called *set* can now be written. Quantification is done over all possible expression evaluators from the element expression (2 evaluators) and the predicate (also 2 evaluators), and all possible sets of binding evaluators from *Partition* (12 sets with 2 binding evaluators each). Thus, 48 (2 times 2

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

times 12) calculations must be carried out. However, in this case some of the combinations of evaluators yields the same result. This is the reason why the entire set comprehension 'only' yields 25 expression evaluators:

$$
\begin{array}{l}
\mathit{EvalExpr}(\mathit{set}) = \\
\{\, \lambda \mathit{env} \,.\, \{(1,\,3,\,[\,]),\,(1,\,4,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,3,\,[7]),\,(1,\,4,\,[7]),\,(1,\,3,\,[\,]),\,(1,\,4,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,3,\,[7,\,4]),\,(1,\,4,\,[7,\,4]),\,(1,\,3,\,[\,]),\,(1,\,4,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,3,\,[\,]),\,(1,\,4,\,[\,]),\,(1,\,3,\,[3]),\,(1,\,4,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,3,\,[7]),\,(1,\,4,\,[7]),\,(1,\,3,\,[3]),\,(1,\,4,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,3,\,[7,\,4]),\,(1,\,4,\,[7,\,4]),\,(1,\,3,\,[3]),\,(1,\,4,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{\,\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,5,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,5,\,[7]),\,(1,\,5,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,5,\,[7,\,4]),\,(1,\,5,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,5,\,[\,]),\,(1,\,5,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,5,\,[7]),\,(1,\,5,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(1,\,5,\,[7,\,4]),\,(1,\,5,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,3,\,[\,]),\,(2,\,4,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,3,\,[7]),\,(2,\,4,\,[7]),\,(2,\,3,\,[\,]),\,(2,\,4,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,3,\,[7,\,4]),\,(2,\,4,\,[7,\,4]),\,(2,\,3,\,[\,]),\,(2,\,4,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,3,\,[\,]),\,(2,\,4,\,[\,]),\,(2,\,3,\,[3]),\,(2,\,4,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,3,\,[7]),\,(2,\,4,\,[7]),\,(2,\,3,\,[3]),\,(2,\,4,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,3,\,[7,\,4]),\,(2,\,4,\,[7,\,4]),\,(2,\,3,\,[3]),\,(2,\,4,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,5,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,5,\,[7]),\,(2,\,5,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,5,\,[7,\,4]),\,(2,\,5,\,[\,])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,5,\,[\,]),\,(2,\,5,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,5,\,[7]),\,(2,\,5,\,[3])\}, \\
\quad \lambda \mathit{env} \,.\, \{(2,\,5,\,[7,\,4]),\,(2,\,5,\,[3])\}\,\}
\end{array}
$$

Notice that one of the expression evaluators yields an empty set. This happens when the second of the binding evaluators for i is chosen (i bound to 5), and the first expression evaluator for the predicate is chosen (requiring that i is less than or equal to 4). In all other cases the result of the two different expression evaluators for the predicate does not make a difference. By carrying out the 48 different calculations one can convince oneself that these 25 expression evaluators actually give the intended meaning to this complicated example.

Notice that if one of the binding evaluators fails entirely, the semantics is simply one expression evaluator returning the empty set. Thus, if the example be modified such that the concatenation pattern is changed to a set union pattern, this binding would fail entirely because it cannot match any of the values in the set $\{[7,4],[3],7\}$.

It is also worth noting that an expression evaluator for a set comprehension expression will yield bottom if the predicate evaluator yield bottom for successful bindings. Thus, if the predicate is modified from the example to: $\mathsf{let}\ x \in \{\mathsf{true}, 4, 6\}\ \mathsf{in}\ i \leq x$ an expression evaluator returning bottom is added.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
85.    EvalSetComprehension : SetComprehension → LEEval
85.1   EvalSetComprehension(MkTag('SetComprehension', (elem, binds, pred))) ≜
  .2    let pbev_s = Partition({ EvalBind(b) | b ∈ binds }),
  .3        plev = EvalExpr(pred) in
  .4     { λenv.let venv_ss = { bev(env) | bev ∈ pbev } in
  .5              if { } ∈ venv_ss ∨ ∃venv_s ∈ venv_ss · err ∈ venv_s
  .6              then if { } ∈ venv_ss
  .7                   then MkTag('set', { })
  .8                   else ⊥
  .9              else if ∃venvs ∈ Partition(venv_ss) · Compatible(venvs) ∧
  .10                                                  pev(overwrite(env, Merge(venvs))) ∉ BOOL_VAL
  .11                  then ⊥
  .12                  else let den = { ev(overwrite(env, Merge(venvs)))
  .13                                 | venvs ∈ Partition(venv_ss) · Compatible(venvs) ∧
  .14                                   pev(overwrite(env, Merge(venvs))) = True() } in
  .15                       if ⊥∈ den
  .16                       then ⊥
  .17                       else if card(den) < ω ∧ ∀e ∈ den · e ∈ FLATVAL
  .18                            then MkTag('set', den)
  .19                            else ⊥
  .20     | ev ∈ EvalExpr(elem), pev ∈ plev, pbev ∈ pbev_s }
```

Annotations to *EvalSetComprehension*:

85. This function gives meaning to set comprehension expressions. A *SetComprehension* expression looks like: {*elem* | *binds* · *pred*} where *elem* is a set element expression, which contains pattern identifiers which are bound to either a type or a set expression via the *binds* component. The predicate *pred* restricts this set to those values fulfilling it (thus, pattern identifiers from *elem* may be used as free variables in *pred*).

.4–.20 The set of all evaluators for the set comprehension expression is constructed. For each element evaluator a set evaluator is returned by using all possible combinations of the predicate evaluators and the binding evaluators.

.4 All possible sets of value environments are collected.

.6–.7 If the empty set belongs to *venv_ss* it corresponds to the situation where one of the bindings have completely failed. Thus, no bindings have succeeded and therefore the empty set is returned.

.8 A 'real' error occurred in a binding and bottom must be returned.

.9–.11 If the predicate evaluator yields a non-Boolean value for any of the successful bindings, bottom is returned.

.12–.14 The part which is expected to be the denotation of the set comprehension expression is created. Each element value must be evaluated in the given environment, extended by a compatible set of value environments resulting from the partitioning of the different binding evaluators (within the given environment). It is also tested that the predicate transformer yields true for each combination within such an extended environment.

.15–.16 If bottom belongs to the set, bottom is returned.

.17–.19 In this case all set elements are "proper" values from the domain universe, it is checked that *den* has finite cardinality and that all elements of *den* are flat values. In this case the tagged set value is returned. Otherwise the construction makes no sense because only finite sets of flat values are allowed in VDM-SL.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

86. $EvalSetRange : SetRange \to LEEval$

86.1 $EvalSetRange(MkTag(\text{'SetRange'}, (lb, ub))) \triangleq$
.2 $\{\lambda env$. **let** $l_val = lb_ev(env)$,
.3 $r_val = rb_ev(env)$ **in**
.4 **if** $(l_val \in NUM_VAL) \wedge (r_val \in NUM_VAL)$
.5 **then let** $l_v = StripNumTagVal(l_val)$,
.6 $r_v = StripNumTagVal(r_val)$ **in**
.7 **let** $num_set = \{ x \in \mathbb{Z} \mid l_v \leq x \leq r_v \}$ **in**
.8 **let** $num_val_s = \{ MkTag(\text{'num'}, x) \mid x \in num_set \}$ **in**
.9 $MkTag(\text{'set'}, num_val_s)$
.10 **else** $\perp$
.11 $\mid lb_ev \in EvalExpr(lb),\ rb_ev \in EvalExpr(ub)\ \}$

Annotations to *EvalSetRange*:

86. This function gives meaning to set range expressions. A typical usage is: $\{1, \ldots, 7\}$ which means: $\{1, 2, 3, 4, 5, 6, 7\}$.

.4–.9 If the lower and upper bounds are both general numeric values, it will denote the set of integers in the closed interval from the lower bound, l_v, to the upper bound, r_l. Note that if the lower bound is greater than the upper bound, an empty set is returned.

.11 This is done for the lower and upper bound expressions for all combinations of their evaluators.

End of annotations

8.3.8 Sequence Expressions

87. $EvalSeqEnumeration : SeqEnumeration \to LEEval$

87.1 $EvalSeqEnumeration(MkTag(\text{'SeqEnumeration'}, els)) \triangleq$
.2 **let** $el_lev_l = [\, EvalExpr(els(i)) \mid i \in \underline{\text{inds}}(els)\,]$ **in**
.3 **let** $seq_lev_s = \{\, ev_l \mid ev_l \in \mathbb{IL}(EEval) \cdot$
.4 $\underline{\text{len}}(ev_l) = \underline{\text{len}}(el_lev_l) \wedge$
.5 $\forall\, i \in \underline{\text{inds}}(ev_l) \cdot ev_l(i) \in el_lev_l(i)\,\}$ **in**
.6 $\{\lambda env$. **let** $seqden = [\, ev_l(i)(env) \mid i \in \underline{\text{inds}}(ev_l)\,]$ **in**
.7 **if** $\perp \notin \underline{\text{elems}}(seqden)$
.8 **then** $MkTag(\text{'seq'}, seqden)$
.9 **else** $\perp$
.10 $\mid ev_l \in seq_lev_s\}$

Annotations to *EvalSeqEnumeration*:

87. This function gives meaning to sequence enumeration expressions.

.2 The list of all element loose evaluators is defined.

.3–.5 The set of all lists of (sequence element) evaluators is defined implicitly by using all possible combinations of the element evaluators.

.6–.10 For all sequences of (sequence element) evaluators the evaluator for the sequence enumeration is created.

.7–.9 If all elements are proper values from the domain universe, the set is tagged and returned. Otherwise it denotes bottom.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
88.   EvalSeqComprehension : SeqComprehension → LEEval
88.1  EvalSeqComprehension(MkTag('SeqComprehension', (elem, bind, pred))) ≙
 .2    {λenv.if bev(env) = { } ∨ err ∈ bev(env)
 .3          then if bev(env) = { }
 .4               then MkTag('seq', [ ])
 .5               else ⊥
 .6          else if ∀venv1, venv2 ∈ bev(env) · card(dom(venv1)) = 1 ∧
 .7                                            dom(venv1) = dom(venv2)
 .8               then let id = Iota(⋃{ dom(venv) | venv ∈ bev(env) }),
 .9                         set = ⋃{ rng(venv) | venv ∈ bev(env) } in
 .10                   if card(set) < ω ∧ ∀e ∈ set · e ∈ NUM_VAL
 .11                   then if ∃e ∈ set · pev(overwrite(env, { id ↦ e })) ∉ BOOL_VAL
 .12                        then ⊥
 .13                        else let v_l = [ eev(overwrite(env, { id ↦ e }))
 .14                                       | e ∈ set ·
 .15                                         pev(overwrite(env, { id ↦ e })) = True() ] in
 .16                             if ⊥ ∈ elems(v_l)
 .17                             then ⊥
 .18                             else MkTag('seq', v_l)
 .19                   else ⊥
 .20              else ⊥
 .21    | eev ∈ EvalExpr(elem), pev ∈ EvalExpr(pred), bev ∈ EvalBind(bind) }
```

Annotations to *EvalSeqComprehension*:

88. This function gives meaning to sequence comprehension expressions. A *SeqComprehension* looks like: [*elem* | *bind* · *pred*], where *elem* is a sequence element expression which contains one pattern identifier which is bound to either a type or a set expression via the *bind* component (to numeric values which can be ordered). The predicate *pred* restricts this set to those values fulfilling it (thus the pattern identifier from *elem* may be used as a free variable in *pred*).

.2 – .21 The set of all evaluators for the sequence comprehension expression is constructed. For each element evaluator a sequence evaluator is returned by using all the predicate evaluators and all possible combinations of the binding evaluators.

.3 – .4 If the binding evaluator yields an empty set with the current environment it corresponds to the situation where the binding has completely failed, and therefore the empty sequence is returned.

.5 A 'real' error occurred in the binding and bottom must be returned.

.6 – .7 For a sequence comprehension it is required that only one pattern identifier is introduced. This is necessary in order to find an appropriate order for the elements in the resulting sequence.

.10 Here it is required that the set which is used for the comprehension is finite and that all elements are numeric values such that they can be ordered.

.11 – .12 If the predicate evaluator yields a non-Boolean value for any of the successful bindings, bottom is returned.

.13 – .15 The part which is expected to be the denotation of the sequence comprehension expression is created. Each element value must be evaluated in the given environment, extended by a value environment mapping the pattern identifier to one of the values. It is also tested that all the predicate transformers return true for each combination within such an extended environment.

.16 – .18 If bottom is a sequence element the construction makes no sense, and bottom is returned. Otherwise the tagged sequence value is returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
89.   EvalSubSequence : SubSequence → LEEval
89.1  EvalSubSequence(MkTag('SubSequence', (seq, from, to))) ≜
 .2    let seq_lev  = EvalExpr(seq),
 .3        from_lev = EvalExpr(from),
 .4        to_lev   = EvalExpr(to) in
 .5     {λenv . let seq_v  = seq_ev(env),
 .6                 from_v = from_ev(env),
 .7                 to_v   = to_ev(env) in
 .8              if seq_v ∈ LIST_VAL ∧ from_v ∈ NUM_VAL ∧ to_v ∈ NUM_VAL
 .9              then let s_v = StripSeqTagVal(seq_v),
 .10                      f_v = StripNumTagVal(from_v),
 .11                      t_v = StripNumTagVal(to_v) in
 .12                   [s_v(i) | i ∈ inds(s_v) · f_v ≤ i ≤ t_v]
 .13              else ⊥
 .14    | seq_ev ∈ seq_lev, from_ev ∈ from_lev, to_ev ∈ to_lev}
```

Annotations to *EvalSubSequence*:

89. This function gives meaning to sub-sequence expressions.

.5–.14 The set of all evaluators for the sub-sequence expression is constructed. For each combination of an evaluator for the sequence expression, an evaluator for the 'from expression' and an evaluator for the 'to expression', an evaluator for the 'sub-sequence expression' is returned.

.8–.12 If the sequence expression denotes a sequence, and the from and to expressions denote numeric values, the sub-sequence is created. Notice that it causes no problems if the bounds are outside the indices for the sequence expression.

End of annotations

8.3.9 Map Expressions

```
90.   EvalMapEnumeration : MapEnumeration → LEEval
90.1  EvalMapEnumeration(MkTag('MapEnumeration', els)) ≜
 .2    let lev_p_l = [let MkTag('Maplet', (d, r)) = els(i) in
 .3                   (EvalExpr(d), EvalExpr(r)) | i ∈ inds(els)] in
 .4     {λenv.let ml_s = {{dev(env) ↦ rev(env)} | (dev, rev) ∈ elems(ev_p_l)} in
 .5             if Compatible(ml_s)
 .6             then let den = Merge(ml_s) in
 .7                    if ⊥ ∉ (dom(den) ∪ rng(den))
 .8                    then if ∀e ∈ dom(den) · e ∈ FLATVAL
 .9                         then MkTag('map', den)
 .10                        else ⊥
 .11                   else ⊥
 .12            else ⊥
 .13    | ev_p_l ∈ AllPairComb(lev_p_l)}
```

Annotations to *EvalMapEnumeration*:

90. This function gives meaning to map enumeration expressions.

.2–.3 A list of pairs of loose evaluators is constructed.

.4–.13 For all possible combinations of pairs in this list, *lev_p_l*, an evaluator for the map enumeration is returned.

.4 A set of singleton maps is created (corresponding to evaluators of the different 'maplets').

.5–.9 If these singleton maps are compatible, they are merged together and it is checked whether all values in the domain and range denote 'proper' values (remember that due to the domain universe construction the elements of the domain of the map are restricted to being flat). If all these tests are passed successfully, the map is tagged and returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
91.   EvalMapComprehension : MapComprehension → LEEval
91.1  EvalMapComprehension(MkTag('MapComprehension', (elem, binds, pred))) ≙
 .2    let MkTag('Maplet', (dom, rng)) = elem in
 .3      let pbev_s = Partition({ EvalBind(b) | b ∈ binds }),
 .4          plev = EvalExpr(pred) in
 .5        { λenv.let venv_ss = { bev(env) | bev ∈ pbev } in
 .6                if { } ∈ venv_ss ∨ ∃venv_s ∈ venv_ss · err ∈ venv_s
 .7                then if { } ∈ venv_ss
 .8                     then MkTag('map', {↦})
 .9                     else ⊥
.10                else if ∃venvs ∈ Partition(venv_ss) · Compatible(venvs) ∧
.11                                                      pev(overwrite(env, Merge(venvs))) ∉ BOOL_VAL
.12                     then ⊥
.13                     else let ml_s = { let env' = overwrite(env, Merge(venvs)) in
.14                                       { dev(env') ↦ rev(env') }
.15                                     | venvs ∈ Partition(venv_ss) ·
.16                                       Compatible(venvs) ∧
.17                                       pev(overwrite(env, Merge(venvs))) = True() } in
.18                          if Compatible(ml_s) ∧ card(ml_s) < ω
.19                          then let den = Merge(ml_s) in
.20                                 if ⊥ ∉ (dom(den) ∪ rng(den))
.21                                 then if ∀e ∈ dom(den) · e ∈ FLATVAL
.22                                      then MkTag('map', den)
.23                                      else ⊥
.24                                 else ⊥
.25                          else ⊥
.26      | dev ∈ EvalExpr(dom), rev ∈ EvalExpr(rng), pev ∈ plev, pbev ∈ pbev_s }
```

Annotations to *EvalMapComprehension*:

91. This function gives meaning to map comprehension expressions. A *MapComprehension* expression looks like: {*elem* | *binds* · *pred*} where *elem* is a 'maplet' expression (i.e. a domain element expression, and a range element expression), which contains pattern identifiers which are bound to either a type or a set expression via the *binds* component. The predicate *pred* restricts this set to those values fulfilling it (thus, pattern identifiers from *elem* may be used as free variables in *pred*).

.5–.26 The set of all evaluators for the map comprehension expression is constructed. For each domain element and range element evaluator, a map evaluator is returned by using all of the predicate evaluators and all possible combinations of the binding evaluators.

.5 All possible sets of value environments are collected.

.7–.8 If the empty set belongs to *venv_ss* it corresponds to the situation where one of the bindings have completely failed. Thus, no bindings have succeeded and therefore the empty map is returned.

.9 A 'real' error occurred in a binding and bottom must be returned.

.10–.12 If the predicate evaluator yields a non-Boolean value for any of the successful bindings, bottom is returned.

.13–.17 A set of singleton maps, from (element) domain evaluators to (element) range evaluators, is created. Each element value must be evaluated in the given environment extended by a compatible set of value environments, resulting from the partitioning of the different binding evaluators (within the given environment). It is also tested that the predicate transformer yields true for each combination within such an extended environment.

.18–.22 If these singleton maps are compatible, they are merged together and it is checked whether all values in the domain and range denote proper values (remember that due to the domain universe construction the elements of the domain of the map are restricted to being flat). If all these tests are passed successfully, the map is tagged and returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.10 Tuple Constructor

```
92.    EvalTupleConstructor : TupleConstructor → LEEval
92.1   EvalTupleConstructor(MkTag('TupleConstructor', fields)) ≜
  .2     let f_lev_l = [ EvalExpr(fields(i)) | i ∈ inds(fields) ] in
  .3       let tup_lev = { ev_l | ev_l ∈ 𝕃(EEval) ·
  .4                             len(ev_l) = len(f_lev_l) ∧
  .5                             ∀i ∈ inds(ev_l) · ev_l(i) ∈ f_lev_l(i) } in
  .6         { λenv . let tupden = [ ev_l(i)(env) | i ∈ inds(ev_l) ] in
  .7                  if ⊥ ∉ elems(tupden)
  .8                  then MkTag('tuple', tupden)
  .9                  else ⊥
  .10        | ev_l ∈ tup_lev }
```

Annotations to *EvalTupleConstructor*:

92. This function gives meaning to tuple constructor expressions.
.2 The list of all element loose evaluators is defined.
.3–.5 The set of all lists of (tuple element) evaluators is defined by using all possible combinations of the element evaluators.
.6–.10 For all possible lists of (tuple element) evaluators, a tuple constructor is created.
.6 The list of tuple elements which are supposed to be used in the denotation of the expression, is created.
.7–.9 If all values in the list are 'proper' values, a tagged tuple is returned. Otherwise bottom is returned.

End of annotations

8.3.11 Record Expressions

```
93.    EvalRecordConstructor : RecordConstructor → LEEval
93.1   EvalRecordConstructor(MkTag('RecordConstructor', (tag, fields))) ≜
  .2     let f_lev_l = [ EvalExpr(fields(i)) | i ∈ inds(fields) ] in
  .3       let rec_lev = { ev_l | ev_l ∈ 𝕃(EEval) ·
  .4                             len(ev_l) = len(f_lev_l) ∧
  .5                             ∀i ∈ inds(ev_l) · ev_l(i) ∈ f_lev_l(i) } in
  .6         { λenv . let recden = [ ev_l(i)(env) | i ∈ inds(ev_l) ] in
  .7                  if ⊥ ∉ elems(recden)
  .8                  then if GetRecConsId(tag) ∈ dom(env) ∧
  .9                          env(GetRecConsId(tag)) ∈ 𝕃(VAL) → RECORD_VAL⊥
  .10                 then env(GetRecConsId(tag))(recden)
  .11                 else ⊥
  .12                 else ⊥
  .13        | ev_l ∈ rec_lev }
```

Annotations to *EvalRecordConstructor*:

93. This function gives meaning to record constructor expressions.
.2 The list of all element loose evaluators is defined.
.3–.5 The set of all lists of (record element) evaluators is defined by using all possible combinations of the element evaluators.
.6–.13 For all possible lists of (record element) evaluators, a record constructor is created.
.6 The list of record elements which are supposed to be used in the denotation of the expression, is created.
.7–.10 If all values in the list are 'proper' values and the record constructor function exists in the environment, a record value returned. Note that the type $\mathbb{L}(VAL) \rightarrow RECORD_VAL_{\perp}$ is also called *CONS_FUN* in the definition of the extended semantic domains.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
94.   EvalRecordModifier : RecordModifier → LEEval
94.1  EvalRecordModifier(MkTag('RecordModifier', (rec, modifiers))) ≜
  .2    let lev_m = { id ↦ EvalExpr(modifiers(id)) | id ∈ dom(modifiers) } in
  .3      { λenv.if rev(env) ∉ RECORD_VAL
  .4             then ⊥
  .5             else let tag = ShowRecTagVal(rev(env)) in
  .6                    let mfn = GetRecModId(tag) in
  .7                      if mfn ∈ dom(env) ∧
  .8                         env(mfn) ∈ (𝔼(VAL) → RECORD_VAL → VAL)
  .9                      then let m = { id ↦ ev_m(id)(env) | id ∈ dom(ev_m) } in
  .10                              env(mfn)(m)(rev(env))
  .11                     else ⊥
  .12     | rev ∈ EvalExpr(rec),
  .13       ev_m ∈ 𝕄(dom(modifiers), EEval) ·
  .14       ∀id ∈ dom(lev_m) · ev_m(id) ∈ lev_m(id) }
```

Annotations to *EvalRecordModifier*:

94. This function gives meaning to record modifier expressions.

.2 A map from record selector identifiers to loose evaluators for the corresponding modifier expression, is created.

.5 – .6 Given that the record expression denotes a record value, the tag of the value can be extracted and that tag can be used to find the modifier function to record values with that tag.

.7 – .8 This modifier function must be present in the given environment, and it must be a Curried function from a map of component values (to be changed) to a function transforming a record value to a value (i.e. the modified record value if the selector identifiers exists for that record). This domain is also called *MOD_FUN* in the definition of the extended semantic domains.

If an attempt is made to modify fields in a record whose tag is not present in the type definition, ⊥ will be returned because *mfn* will not be in dom(*env*). Bottom will also be returned if an attempt is made to modify fields which are not present in the record (this can be seen in the auxiliary function *GenRecordModFn*).

.9 – .10 The modifier function is looked up, and applied with a map from selector identifiers to their corresponding values and the record value.

.12 – .14 This is done for all possible combinations of evaluators for the record expression, *rec*, and the evaluator mappings corresponding to the different modifications of the given record value.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.3.12 Apply Expressions

```
95.   EvalApplyExpr : Apply → LEEval
95.1  EvalApplyExpr(MkTag('Apply', (fct, arg))) ≜
 .2    {λenv.let a = a_ev(env),
 .3             f = f_ev(env) in
 .4         if f ∉ (FUN_VAL ∪ MAP_VAL ∪ LIST_VAL) ∨ a = ⊥
 .5         then ⊥
 .6         else cases f :
 .7                MkTag('fun', fn)  → if a ∈ δ0(fn)
 .8                                    then fn(a)
 .9                                    else ⊥,
 .10               MkTag('map', m)   → if a ∈ dom(m)
 .11                                   then m(a)
 .12                                   else ⊥,
 .13               MkTag('seq', l)   → if a ∈ inds(l)
 .14                                   then l(a)
 .15                                   else ⊥
 .16   | f_ev ∈ EvalExpr(fct), a_ev ∈ EvalExpr(arg) }
```

Annotations to *EvalApplyExpr*:

95. This function gives meaning to apply expressions.

.4–.5 If the value being applied is not either a function, a map or a list, or the argument value is equal to bottom, bottom is returned.

.6–.15 Depending on the different kinds of values of f it is checked whether the argument value can be used; if so it is applied, otherwise bottom is returned.

.7 Recall that δ_0 is the domain operator on general functions (as described page 8).

.16 This is done for all possible evaluators for *fct* and *arg*.

End of annotations

```
96.   EvalFieldSelectExpr : FieldSelect → LEEval
96.1  EvalFieldSelectExpr(MkTag('FieldSelect', (record, field))) ≜
 .2    {λenv.if ev(env) ∈ RECORD_VAL
 .3           then let tag = ShowRecTagVal(ev(env)) in
 .4                let sel_fn = GetRecSelId(tag, field) in
 .5                  if sel_fn ∈ dom(env) ∧ env(sel_fn) ∈ RECORD_VAL → VAL ∧
 .6                     ev(env) ∈ δ0(env(sel_fn))
 .7                  then env(sel_fn)(ev(env))
 .8                  else ⊥
 .9           else ⊥
 .10   | ev ∈ EvalExpr(record) }
```

Annotations to *EvalFieldSelectExpr*:

96. This function gives meaning to field selection expressions.

.2–.4 If an evaluator for the record expression, *record*, denotes a record value in the given environment, its tag can be found, and that tag together with the field selection identifier can be used to obtain of the name of the record selector function for that field in this particular kind of record.

.5–.7 If this selector function is present in the given environment and the selector function belongs to the domain which contains all record selector functions (also called *SEL_FUN* in the definition of the extended semantic domains), and the record value belongs to the domain of the record selector function, the record value is applied to it.

.10 This is done for all evaluators for the record expression, *record*.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

97. $EvalFctTypeInstExpr : FctTypeInst \to LEEval$

97.1 $EvalFctTypeInstExpr(MkTag(\text{'FctTypeInst'}, (id, inst))) \triangleq$
.2 $\{ \lambda env.$ **let** $d_l = EvalTypeList(inst)(env)$ **in**
.3 **if** $d_l = \underline{\text{err}} \vee \exists i \in \underline{\text{inds}}(inst) \cdot \neg CheckTagEnv(inst(i))$
.4 **then** $\bot$
.5 **else if** $id \in \underline{\text{dom}}(env) \wedge env(id) \in POLYVAL$
.6 **then** $env(id(d_l))$
.7 **else** $\bot$ $\}$

Annotations to *EvalFctTypeInstExpr*:

97. This function gives meaning to expressions which instantiate polymorphic functions.
.2 The meaning of the list of type to be instantiated is found in the given environment.
.3–.4 If an error is reported as the meaning of the type list or an undefined composite type is used, bottom is returned.
.5–.7 Otherwise if the identifier, *id*, is bound to a polymorphic value in the given environment, *env*, the meaning of the type list is applied to the polymorphic value.

End of annotations

8.3.13 Lambda Expression

98. $EvalLambda : Lambda \to LEEval$

98.1 $EvalLambda(MkTag(\text{'Lambda'}, (MkTag(\text{'Par'}, (id, tp)), body))) \triangleq$
.2 **let** $tev = EvalType(tp)$ **in**
.3 **let** $quasi_eval = \lambda env$. **let** $d = tev(env)$ **in**
.4 **if** $d = \underline{\text{err}} \vee \neg CheckTagEnv(env, tp)$
.5 **then** $\bot$
.6 **else** $\lambda ob \in |d|$. $\{$ **if** $ob \in \|d\|$
.7 **then** $ev(\underline{\text{overwrite}}(env, \{ id \mapsto ob \}))$
.8 **else** $\bot$
.9 $\mid ev \in EvalExpr(body) \}$ **in**
.10 $\{ eeval \mid eeval \in EEval \cdot$
.11 $\forall env \in ENV \cdot$
.12 $(eeval(env) = \bot \wedge (tev(env) = \underline{\text{err}} \vee \neg CheckTagEnv(env, tp))) \vee$
.13 $(ShowTag(eeval(env)) = \text{'Fun'} \wedge tev(env) \neq \underline{\text{err}} \wedge CheckTagEnv(env, tp) \wedge$
.14 $\delta_0(StripTag(eeval(env))) = |tev(env)| \wedge$
.15 $\forall ob \in |tev(env)| \cdot StripTag(eeval(env))(ob) \in quasi_eval(env)(ob)) \}$

Annotations to *EvalLambda*:

98. This function gives meaning to monomorphic lambda expressions. A *Lambda* expression looks like: $\lambda id : tp \cdot body$ where $id : tp$ is a binding of an identifier to a type, and the *body* part is an expression which uses this identifier as a free variable.
.3–.9 A quasi-evaluator for the lambda expression is produced. It has the type: $ENV \to \|d\| \to \mathbb{P}(VAL)$ given that the argument type, *tp*, is not erroneous and *tp* denotes *d* in the given environment. This is not a "real" evaluator because the function values are not tagged by *Fun* and they are some kind of multi-functions where looseness of the body have not been propagated out.
.10–.15 This yield all possible expression evaluators which yields proper functions with the domain corresponding to the type evaluator and a body which yields one of the possible values returned by the functions in the quasi-evaluator.
.12 Given that there is something wrong with the given domain type the expression evaluator must yield bottom for all possible environments.
.13 If there is nothing wrong with the given domain type the expression evaluator must yield a function value.
.14 The domain of this function must be equal to the denotation of the domain type.
.15 For all values in the domain of the function the function value applied with this value must be one of

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

the values from the quasi-evaluator applied with the same value. This ensures that there is a functional dependency for all choices which are present in the body of the function such that models exist where different choices are taken for different arguments.

End of annotations

8.3.14 Is Expression

```
99.    EvalIsExpr : IsExpr → LEEval
99.1   EvalIsExpr(MkTag('IsExpr', (tag, arg))) ≙
  .2    { λenv.let val = ev(env) in
  .3            if ShowTag(tag) = 'Id'
  .4            then if val ∈ RECORD_VAL
  .5                 then if ShowRecTagVal(val) = StripTag(tag)
  .6                      then True()
  .7                      else False()
  .8                 else ⊥
  .9            else let d = EvalType(tag)(env) in
  .10                 if d = err ∨ val = ⊥
  .11                 then ⊥
  .12                 else if val ∈ ||d||
  .13                      then True()
  .14                      else False()
  .15    | ev ∈ EvalExpr(arg) }
```

Annotations to *EvalIsExpr*:

99. This function gives meaning to 'is-expressions'. An is-expression, which takes a tag or a basic type A and an expression e, asks whether or not the value of the expression e either has the tag A or belongs to the basic type A. The semantics of an is-expression requires that a type must be tagged or be basic, in order to be used for checking the type membership. The reason for this is purely pragmatic, in order to let the user understand that in the cases where an is-expression is used it is necessary to explicitly use some kind of tagging in the implementation later on.

.5–.8 If *tag* is a tag (and not a basic type), it is checked whether the value has that tag.

.9–.14 Otherwise *tag* is a basic type for which it is checked whether the value belongs to that basic type.

End of annotations

```
100.    EvalLiteralExpr : Literal → LEEval
100.1   EvalLiteralExpr(lit) ≙
  .2     { λenv.cases lit :
  .3             MkTag('NumLit', val)  → MkTag('num', val),
  .4             MkTag('BoolLit', val) → if val = TRUE
  .5                                     then True()
  .6                                     else False(),
  .7             MkTag('CharLit', val) → let MkTag('CharLit', c) = val in
  .8                                       MkTag('char', c),
  .9             MkTag('TextLit', val) → let l = [ MkTag('char', StripTag(val(i)))
  .10                                            | i ∈ inds(val) ] in
  .11                                      MkTag('seq', l),
  .12            MkTag('QuoteLit', val) → MkTag('quot', val),
  .13            MkTag('NilLit', ())    → MkTag('nil', nil) }
```

Annotations to *EvalLiteralExpr*:

100. This function gives meaning to literal expressions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.2–.13 The different kinds of literals are transformed into corresponding basic values.

End of annotations

101. $EvalId : Id \rightarrow LEEval$

101.1 $EvalId(id) \triangleq$
.2 $\{ \lambda env$. **if** $id \in \underline{\text{dom}}(env) \wedge env(id) \in VAL$
.3 **then** $env(id)$
.4 **else** $\perp \}$

Annotations to *EvalId*:

101. This function gives meaning to identifiers.

End of annotations

102. $EvalOldId : OldId \rightarrow LEEval$

102.1 $EvalOldId(id) \triangleq$
.2 $\{ \lambda env$.**if** $id \in \underline{\text{dom}}(env)$
.3 **then** $env(id)$
.4 **else** $\perp \}$

Annotations to *EvalOldId*:

102. This function gives meaning to expressions with a 'hooked' state component. The *OldId* expression returns the state value or a specific part of the state before evaluation of an implicit operation. It does so by looking up in the model by means of the auxiliary function *LookUp*. This expression can only be used inside a post condition of an implicitly defined operation.

End of annotations

8.4 State Designators

When reference to a part of the state is desired or to a part of a local variable inside an operation a state designator is used. This clause¡ contains functions which modify a state designator and look up the value of a state designator in the environment.

103. $Modify : StateDesignator \times VAL \rightarrow LDef$

103.1 $Modify(sd, val) \triangleq$
.2 **cases** $ShowTag(sd)$:
.3 $'ValueId' \rightarrow ModifyValueId(sd, val)$,
.4 $'FieldRef' \rightarrow ModifyFieldRef(sd, val)$,
.5 $'MapOrSeqRef' \rightarrow ModifyMapOrSeqRef(sd, val)$

Annotations to *Modify*:

103. This function modifies a state designator, *sd*, with a value, *val*. The state designator can either be a simple identifier or a reference to an entry in a map, an index in a sequence or a field in a record. The different state designators are dealt with by means of sub-functions.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
104.    ModifyValueId : ValueId × VAL → LDef
104.1   ModifyValueId(id, val) ≜
  .2     { λenv.let te_id = MkTag('TypeEnvId', ()),
  .3                 st_id = GetStateTypeId(env) in
  .4              if te_id ∈ dom(env) ∧ env(te_id) ∈ LOC_ENV
  .5              then if id ∈ dom(env(te_id))
  .6                   then if val ∈ ||env(te_id)(id)||
  .7                        then overwrite(env, { id ↦ val })
  .8                        else err
  .9                   else ModifyState(id, val)(env)
  .10             else ModifyState(id, val)(env) }
```

Annotations to *ModifyValueId*:

104. This function modifies a value identifier state designator, *id*, with a value, *val*.

.5–.8 If the identifier, *id*, is a local variable it is checked whether the value, *val*, satisfy the invariant of the type of that local variable.

.9–.10 If it is not a local variable it must be a state component and that is dealt with by *ModifyState*.

End of annotations

```
105.    ModifyState : ValueId × VAL → Def
105.1   ModifyState(id, val)(env) ≜
  .2     let st_id = GetStateTypeId(env),
  .3         mod = { id ↦ val } in
  .4       if st_id = nostate
  .5       then err
  .6       else let st_v_id = GetStateValId(CUR, st_id) in
  .7            if id = st_v_id
  .8            then if val ∈ ||env(st_id)||
  .9                 then overwrite(env, mod)
  .10                else err
  .11           else let MkTag('ValueId', id') = id,
  .12                    MkTag('StateTypeId', st_id') = st_id in
  .13                if st_v_id ∈ dom(env) ∧
  .14                   MkTag('RecModId', st_id') ∈ dom(env) ∧
  .15                   env(MkTag('RecModId', st_id')) ∈ (𝔼(VAL) → RECORD_VAL → VAL)
  .16                then let mod_id = MkTag('RecModId', st_id'),
  .17                         old_st_v = env(st_v_id) in
  .18                     let new_st = env(mod_id)(mod)(old_st_v) in
  .19                       if new_st ∈ ||env(st_id)||
  .20                       then let st_m = { st_v_id ↦ new_st } in
  .21                                overwrite(env, merge(mod, st_m))
  .22                       else err
  .23                else err
```

Annotations to *ModifyState*:

105. This function modifies a value identifier, *id*, with a value, *val*. Note that the identifier is not a local variable, and thus it is assumed to be either the name of the entire state or a state component.

.4–.5 If no state is present, an error is reported.

.7–.10 If *id* is the entire state, its value is modified if it satisfy the state invariant; otherwise an error is reported.

.11–.23 Otherwise *id* must be a state component and the modifier function for the state must exist in the environment. This modifier function is used to create the new state. If the new state satisfy the state invariant, the given environment is overwritten with the entire new state and thus new binding of the given state component, *id*.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
106.    ModifyFieldRef : FieldRef × VAL → LDef
106.1   ModifyFieldRef(MkTag('FieldRef', (sd, sel)), val) ≜
  .2     PropErr({ λenv.if rev(env) ∉ RECORD_VAL
  .3                   then {err}
  .4                   else let tag = ShowRecTagVal(rev(env)) in
  .5                          let mfn = GetRecModId(tag) in
  .6                            if mfn ∈ dom(env) ∧
  .7                               env(mfn) ∈ (𝔼(VAL) → RECORD_VAL → VAL)
  .8                            then Modify(sd, env(mfn)({ sel ↦ val })(rev(env)))
  .9                            else {err}
  .10             | rev ∈ LookUp(sd) })
```

Annotations to *ModifyFieldRef*:

106. This function modifies a field reference state designator with a value, *val*.

.2–.3 If the state designator which one wish to refer to a field from is not a record value, an error is reported.

.4–.5 If it is a record value, the tag can be extracted and can be used to obtain the name of the corresponding record modifier function.

.6–.8 If the record modifier function is present in the environment and it has the right type (i.e. is generated by *CompRecordModifiers*), the state designator, *sd*, is modified with a value where the record modifier function has been applied with a map from the selector identifier to the given value, *val*, and the previous value of the record (as a Curried argument). The domain of record modifiers is also called *MOD_FUN* in the definition of the extended domains.

End of annotations

```
107.    ModifyMapOrSeqRef : MapOrSeqRef × VAL → LDef
107.1   ModifyMapOrSeqRef(MkTag('MapOrSeqRef', (sd, sel)), val) ≜
  .2     PropErr({ λenv.if sdev(env) ∉ MAP_VAL
  .3                   then if sdev(env) ∉ LIST_VAL
  .4                        then {err}
  .5                        else let MkTag('seq', l) = sdev(env) in
  .6                               if sev(env) ∈ inds(l)
  .7                               then let l' = [if i = sev(env)
  .8                                              then val
  .9                                              else l(i)
  .10                                             | i ∈ inds(l)] in
  .11                                    Modify(sd, MkTag('seq', l'))
  .12                              else {err}
  .13                   else let MkTag('map', m) = sdev(env) in
  .14                          if sev(env) ∈ dom(m)
  .15                          then let v = overwrite(m, { sev(env) ↦ val }) in
  .16                                 Modify(sd, MkTag('map', v))
  .17                          else {err}
  .18             | sdev ∈ LookUp(sd), sev ∈ EvalExpr(sel) })
```

Annotations to *ModifyMapOrSeqRef*:

107. This function modifies a map or sequence reference state designator with a value, *val*.

.2–.4 If the state designator which one wish to refer to is neither a map value nor a sequence value, an error is reported.

.5–.12 If it is a sequence value, it is required that the selector part, *sel*, has an expression evaluator which provides a legal index in the given environment. In this case, the state designator, *sd*, is modified with a sequence value where the value, *val*, has been inserted at the required position.

.13–.17 If it is a map value it is required that the selector part, *sel*, has an expression evaluator which provides a value which belongs to the domain of the map. In this case, the state designator, *sd*, is modified with a map value where the value, *val*, has been inserted at the required place in the map.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

End of annotations

```
108.   LookUp : StateDesignator → LEEval
108.1  LookUp(st) ≜
  .2    cases ShowTag(st) :
  .3      'ValueId'       → { λenv . if st ∈ dom(env)
  .4                                 then env(st)
  .5                                 else ⊥ },
  .6      'FieldRef'      → LookUpFieldRef(st),
  .7      'MapOrSeqRef'   → LookUpMapOrSeqRef(st)
```

Annotations to *LookUp*:

108. This function looks up the value of a state designator, *st*. This (and the following) lookup functions are not strictly necessary to give semantics to VDM-SL without modules. However, if the definition later will be extended to include modules they are necessary and therefore have been included here, in order to enable such an extension.

End of annotations

```
109.   LookUpFieldRef : FieldRef → LEEval
109.1  LookUpFieldRef(MkTag('FieldRef', (sd, sel))) ≜
  .2    { λenv.if rev(env) ∉ RECORD_VAL
  .3           then ⊥
  .4           else let tag = ShowRecTagVal(rev(env)) in
  .5                  let sfn = GetRecSelId(tag, sel) in
  .6                    if sfn ∈ dom(env) ∧ env(sfn) ∈ RECORD_VAL → VAL
  .7                    then env(sfn)(rev(env))
  .8                    else ⊥
  .9    | rev ∈ LookUp(sd) }
```

Annotations to *LookUpFieldRef*:

109. This function looks up the value of a field reference state designator.

.2–.3 If the state designator, *sd*, is not a record value, bottom is returned, indicating an error.

.4–.5 If it is a record value the tag can be extracted, and can be used together with the selector identifier to obtain the name of the corresponding record selector function.

.6–.8 If the record selector function is present in the environment and it has the right type[1)] (is generated by *CompFieldSelectors*), the field selector is looked up in the environment and applied with the record value.

.9 This is done for all possible ways of looking up the state designator, *sd*.

End of annotations

[1)] The domain containing record selector functions is also called *SEL_FUN* in the definition of the extended semantic domains.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
110.    LookUpMapOrSeqRef : MapOrSeqRef → LEEval
110.1   LookUpMapOrSeqRef(MkTag('MapOrSeqRef', (sd, sel))) ≜
  .2     { λenv.if sdev(env) ∉ MAP_VAL
  .3           then if sdev(env) ∉ LIST_VAL
  .4                then ⊥
  .5                else let MkTag('seq', l) = sdev(env) in
  .6                     if sev(env) ∈ inds(l)
  .7                     then l(sev(env))
  .8                     else ⊥
  .9           else let MkTag('map', m) = sdev(env) in
  .10               if sev(env) ∈ dom(m)
  .11               then m(sev(env))
  .12               else ⊥
  .13    | sdev ∈ LookUp(sd), sev ∈ EvalExpr(sel) }
```

Annotations to *LookUpMapOrSeqRef*:

110. This function looks up the value of a map or sequence reference state designator.

.2–.4 If the state designator which one wish to refer to is neither a map value nor a sequence value, bottom is returned, indicating an error.

.5–.8 If it is a sequence value, it is required that the selector part, *sel*, has an expression evaluator which provides a legal index in the given environment. In this case, the sequence is indexed with the value of the expression evaluator for the selector part in *env*.

.9–.12 If it is a map value, it is required that the selector part, *sel*, has an expression evaluator which provides a value belonging to the domain of the map. In this case, the map is applied with the value of the expression evaluator for the selector part in *env*.

.13 This is done for all possible combinations of expression evaluators from the selector expression, *sel*, and from looking up the state designator, *sd*.

End of annotations

8.5 Statements

The body of explicitly defined operations consists of a statement similar to that of traditional imperative programming languages. However, because of the presence of looseness in VDM-SL, statements are in general non-deterministic, which normally is not the case in programming languages. A deterministic statement denotes a function changing an environment. A non-deterministic statement denotes a set of such functions.

8.5.1 Underlying Theory For Statements

This clause describes the theory used to give the semantics of the non-deterministic statements in VDM-SL.

In case of deterministic statements the semantics of (possibly infinite) loops are usually given as the least fixed point of a functional $F(f) = comp(b, f)$, where b is the denotation of the body of the loop and *comp* is an appropriate function composition. The least fixed point can be calculated as the least upper bound of a chain $\{ f_n \mid n \in \mathbb{N} \}$ where $f_0 = \bot$ and $f_{n+1} = comp(b, f_n)$, $n \in \mathbb{N}$.

Let X be a cpo ordered by $\sqsubseteq_X$ and with bottom $\bot$. Let B be any subset of $|X|$, possibly without $\bot$, it is intended to be the denotation of the body of a nondeterministic loop. Let $comp : B \times X \to X$ be a function continuous with respect to the second argument. Define a set function *NonDetComp*[1)] by the formula: $NonDetComp(A) = \{ comp(b, a) \mid b \in B \wedge a \in A \}$. The function takes subsets of X as arguments and returns subsets of X as values. The denotation of a nondeterministic loop is defined as the fixed point of *NonDetComp*. The calculation and the properties of the fixed point follows.

[1)] *NonDetComp* will be a functional similar to F for dealing with loops which contains non-determinism.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Recall that $\mathbb{L}(B) = \bigcup\{ B^n \mid n \in \mathbb{N}\}$ and denote by $\mathbb{L}_\omega(B)$ the set of all infinite sequences over B. The union of the two sets is ordered by the prefix order, and is a cpo with respect to this order and the empty sequence as bottom. Define iteration over finite sequences of elements of B by the formula $(H : \mathbb{L}(B) \to X)$:

$$\begin{cases} H([\,]) &= \bot \\ H(s) &= comp(\underline{\mathrm{hd}}(s), H(\underline{\mathrm{tl}}(s))), \qquad s \neq [\,] \end{cases} \tag{1}$$

Note that H is a monotonic function. This can be shown by means of induction over the length of the sequence:

Iterations over infinite sequences are defined as:

$$H(s) = \bigsqcup_X (H(s_n) \mid n \in \mathbb{N}), \tag{2}$$

where $\bigsqcup_X$ stands for the least upper bound in X and $(H(s_n) \mid n \in \mathbb{N})$ is any chain of elements of $\mathbb{L}(B)$ approximating $s \in \mathbb{L}_\omega(B)$. By monotonicity of H, $(s_n \mid n \in \mathbb{N})$ is a chain. Due to the continuity of *comp* the formula (1) is still valid for infinite sequences. For,

$$\begin{aligned} H(s) &= \bigsqcup_X \{ H(s_n) \mid n \geq N \} \\ &= \bigsqcup_X \{ comp(\underline{\mathrm{hd}}(s_n),\ H(\underline{\mathrm{tl}}(s_n))) \mid n \in \mathbb{N} \} \\ &= \bigsqcup_X \{ comp(\underline{\mathrm{hd}}(s),\ H(\underline{\mathrm{tl}}(s_n))) \mid n \in \mathbb{N} \} \\ &= comp(\underline{\mathrm{hd}}(s),\ \bigsqcup_X \{ H(\underline{\mathrm{tl}}(s_n)) \mid n \in \mathbb{N} \}) \\ &= comp(\underline{\mathrm{hd}}(s),\ H(\underline{\mathrm{tl}}(s))). \end{aligned}$$

These five steps can be explained:

a) By the definition in (2).

b) By (1) applied to finite sequences.

c) Because $\underline{\mathrm{hd}}(s) = \underline{\mathrm{hd}}(s_n)$.

d) By the continuity of *comp*.

e) By (2) applied to the tail of s, and the continuity of $\underline{\mathrm{tl}}$.

The set of all infinite iterations $\{ H(s) \mid s \in \mathbb{L}_\omega(B) \}$ will be denoted by an expression $NonDetIter(B,\ comp)$, where *NonDetIter* is formally defined as one of the semantic functions. This set is a fixed point of *NonDetComp*, formally:

$$NonDetIter(B,\ comp) = \{ comp(b,\ a) \mid b \in B \wedge a \in NonDetIter(B,\ comp) \},$$

as can be seen by:

$$\begin{aligned} &\{ comp(b,\ a) \mid b \in B \wedge a \in NonDetIter(B,\ comp) \} \\ &\qquad = \{ comp(b,\ H(s)) \mid b \in B \wedge s \in \mathbb{L}_\omega(B) \} \\ &\qquad = \{ H(\underline{\mathrm{join}}_\omega([b],\ s)) \mid b \in B \wedge s \in \mathbb{L}_\omega(B) \} \\ &\qquad = \{ H(s) \mid s \in \mathbb{L}_\omega(B) \} \\ &\qquad = NonDetIter(B,\ comp) \end{aligned}$$

This construction will be used to define the semantics of non-deterministic iterations (*While* and *RecTrapStmt*).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

However, there is no ordering on the powerset $\mathbb{P}(X)$ hence the fixed point of *NonDetComp* is not necessarily the least one.

The operator $\underline{\text{join}}_\omega$ is is a simple extension of the $\underline{\text{join}}$ operator and it is defined as:

$\underline{\text{join}}_\omega : \mathbb{L}(A) \times \mathbb{L}_\omega(A) \to \mathbb{L}_\omega(A)$
$\underline{\text{join}}_\omega([a_1, \ldots, a_n], [b_1, \ldots]) = [a_1, \ldots, a_n, b_1, \ldots]$

8.5.2 The Statement Evaluation Functions

```
111.    EvalStmt : Stmt → LSEval
111.1   EvalStmt(stmt) ≙
   .2    if stmt = IDENT
   .3    then { λenv . (env, cont, nil) }
   .4    else cases ShowTag(stmt) :
   .5          'LetStmt'        → EvalLetStmt(stmt),
   .6          'LetBeSTStmt'    → EvalLetBeSTStmt(stmt),
   .7          'DefStmt'        → EvalDefStmt(stmt),
   .8          'Block'          → EvalBlock(stmt),
   .9          'Sequence'       → EvalSequence(stmt),
   .10         'Assign'         → EvalAssign(stmt),
   .11         'IfStmt'         → EvalIfStmt(stmt),
   .12         'CasesStmt'      → EvalCasesStmt(stmt),
   .13         'SeqForLoop'     → EvalSeqForLoop(stmt),
   .14         'SetForLoop'     → EvalSetForLoop(stmt),
   .15         'IndexForLoop'   → EvalIndexForLoop(stmt),
   .16         'WhileLoop'      → EvalWhileLoop(stmt),
   .17         'NonDetStmt'     → EvalNonDetStmt(stmt),
   .18         'Call'           → EvalCall(stmt),
   .19         'ReturnStmt'     → EvalReturnStmt(stmt),
   .20         'Always'         → EvalAlways(stmt),
   .21         'TrapStmt'       → EvalTrapStmt(stmt),
   .22         'RecTrapStmt'    → EvalRecTrapStmt(stmt),
   .23         'Exit'           → EvalExit(stmt)
```

Annotations to *EvalStmt*:

111. This function gives meaning to statements.
.3 The identity statement has no effect. It terminates normally and returns no value.
.5 – .23 Meaning of the different kinds of statements is split into sub-functions.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.5.3 Local Binding Statements

```
112.    EvalLetStmt : LetStmt → LSEval
112.1   EvalLetStmt(MkTag('LetStmt', (vals, explfns, implfns, in))) ≜
  .2     let lsev = { λenv.let fnenv = restrict(env', dom(explfns) ∪ dom(implfns)),
  .3                              ids = ⋃{ ColIdSet(pat) | pat ∈ dom(vals) } ∪
  .4                                    dom(explfns) ∪ dom(implfns) in
  .5                          let env'' = overwrite(env, fnenv) in
  .6                            if def(env'') = err ∨ BotEnv(def(env''))
  .7                            then (env, ret, ⊥)
  .8                            else Restore(env, ids, ev(def(env'')))
  .9                 | def ∈ EvalLocalDef(vals, explfns, implfns), ev ∈ EvalStmt(in), env' ∈ ENV ·
  .10                  VerifyExplFns(explfns, env') ∧ VerifyImplFns(implfns, env') } in
  .11    if lsev = { }
  .12    then { λenv . (env, ret, ⊥) }
  .13    else lsev
```

Annotations to *EvalLetStmt*:

112. This function gives meaning to *Let* statements[1].

.2 Only the denotations of the locally defined functions from the indexed environments are needed.

.3–.4 The locally defined identifiers (i.e. the pattern identifiers from the value definitions and the names of the function definitions) are collected.

.6–.7 If either the meaning of the local definitions is an error or a non-strict binding has occurred, an error is reported by returning the three-tuple where the bottom value is returned.

.8 The bindings of the locally defined identifiers must be restored after evaluation of the 'in-statement' in this context.

.9–.10 This is done for all possible definers and all possible statement evaluators for the 'in-statement' and for all possible environments which provide appropriate semantics for the locally defined functions. In this way locally defined functions are given the same semantics as globally defined ones.

.11–.13 If no statement evaluators were collected the let statement denotes the singleton set with the statement evaluator which returns bottom.

End of annotations

```
113.    EvalLetBeSTStmt : LetBeSTStmt → LSEval
113.1   EvalLetBeSTStmt(MkTag('LetBeSTStmt', (bind, st, in))) ≜
  .2     let ids = ColIdSet(bind) in
  .3       PropS({ λenv. let venv_s = ⋃{ bev(env) | bev ∈ EvalBind(bind) } in
  .4                       let venv_s' = { venv | venv ∈ venv_s · venv ≠ err ∧ ¬BotEnv(venv) ∧
  .5                                                stev(overwrite(env, venv)) = True() } in
  .6                         { Restore(env, ids, inev(overwrite(env, venv))) | venv ∈ venv_s' }
  .7              | stev ∈ EvalExpr(st), inev ∈ EvalStmt(in) })
```

Annotations to *EvalLetBeSTStmt*:

113. This function gives meaning to let be (such that) statements.

.3 All possible bindings are gathered.

.4–.5 All possible value environments which fulfil the 'such that' predicate, *st*, are extracted from all possible binding environments. Notice that only strict binding environments are considered. If an error is reported from the pattern matching these two lines yield an empty set, which makes the semantics of the entire let be such that expression yield bottom.

[1] Notice that the difference between a *Def* statement and a *Let* statement is that the right hand side of the binding part of a *Def* statement is allowed to have side effects on the state, while the right hand side of the binding part of a *Let* statement is purely applicative. Notice also that the names in the left hand side pattern, *lhs*, cannot refer to state components (the assignment statement is used for changing state components). In addition a *Let*-statement is interpreted as the least fixed point of a number of mutually recursive definitions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.6 The old bindings of the used identifiers are restored after evaluation of the 'in-statement' in all possible value environments.

.7 This is done for all possible 'such that' predicate evaluators and all possible statement evaluators for the 'in-statement'.

End of annotations

```
114.   EvalDefStmt : DefStmt → LSEval
114.1  EvalDefStmt(MkTag('DefStmt', (lhs, rhs, in))) ≜
  .2    let ids = ColIdSet(lhs) in
  .3      if ShowTag(rhs) = 'Call'
  .4      then { λenv.let (env', mode, val) = cev(env) in
  .5                  if mode = exit
  .6                  then (env', exit, val)
  .7                  else if val = nil
  .8                       then (env, ret, ⊥)
  .9                       else let venv = pev(val)(env') in
  .10                            if venv ∈ { err, unmatch } ∨ BotEnv(venv)
  .11                            then (env, ret, ⊥)
  .12                            else let res = inev(overwrite(env', venv)) in
  .13                                 Restore(env, ids, res)
  .14           | pev ∈ EvalPattern(lhs), cev ∈ EvalCall(rhs), inev ∈ EvalStmt(in) }
  .15     else { λenv.let venv = pev(eev(env))(env) in
  .16                 if venv ∈ { err, unmatch } ∨ BotEnv(venv)
  .17                 then (env, ret, ⊥)
  .18                 else Restore(env, ids, inev(overwrite(env, venv)))
  .19          | pev ∈ EvalPattern(lhs), eev ∈ EvalExpr(rhs), inev ∈ EvalStmt(in) }
```

Annotations to *EvalDefStmt*:

114. This function gives meaning to definition statements[1].

.4–.14 This part is used if the right hand side, *rhs*, of the definition statement is a call of a value returning operation.

.5–.6 If the mode of the operation call is exit the entire define statement is left and it will denote the meaning of the operation call.

.7–.8 If the operation called does not return any value an error is reported by returning the bottom value.

.9–.11 If an error is returned from matching the pattern evaluator against the value returned from the operation (using the returned environment, *env'*), an error is reported by returning the bottom value.

.12–.13 The old bindings of the used identifiers, *ids*, are restored.

.15–.19 This part is used if the right hand side, *rhs*, of the define statement is an expression.

.15–.17 If either an error is returned from matching the pattern evaluator against the value of the right hand side expression (using the given environment, *env*), or the matching resulted in a non-strict binding, an error is reported by returning the bottom value.

.18 The old bindings of the used identifiers, *ids*, are restored after evaluation of the 'in statement' in this context.

End of annotations

[1] Notice that the difference between a *Def* statement and a *Let* statement is that the right hand side of the binding part of a *Def* statement is allowed to have side effects on the state, while the right hand side of the binding part of a *Let* statement is purely applicative. Notice also that the names in the left hand side pattern, *lhs*, cannot refer to state components (the assignment statement is used for changing state components).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.5.4 Block and Assignment Statements

```
115.    EvalBlock : Block → LSEval
115.1   EvalBlock(MkTag('Block', (id, init, tp, body))) ≜
  .2     let d = EvalType(tp) in
  .3       if init = nil
  .4       then { λenv.if d(env) = err ∨ ¬CheckTagEnv(env, tp)
  .5                   then (env, ret, ⊥)
  .6                   else let env' = ExtLocDomEnv(id, d(env))(env) in
  .7                          ResLocId(env, id, bev(env'))
  .8            | bev ∈ EvalStmt(body) }
  .9       else if ShowTag(init) = 'Call'
  .10           then { λenv.if d(env) = err ∨ ¬CheckTagEnv(env, tp)
  .11                       then (env, ret, ⊥)
  .12                       else let (e', mode, val) = inev(env) in
  .13                              if mode = exit
  .14                              then (e', exit, val)
  .15                              else if val = nil
  .16                                   then (env, ret, ⊥)
  .17                                   else let e'' = ExtLocDomEnv(id, d(env))(e') in
  .18                                          let e''' = Iota(Modify(id, val))(e'') in
  .19                                            if e''' = err
  .20                                            then (env, ret, ⊥)
  .21                                            else let v = bev(overwrite(e'', e''')) in
  .22                                                   ResLocId(env, id, v)
  .23                | inev ∈ EvalCall(init), bev ∈ EvalStmt(body) }
  .24           else {λenv.if inev(env) = ⊥ ∨ d(env) = err ∨ ¬CheckTagEnv(env, tp)
  .25                       then (env, ret, ⊥)
  .26                       else let env' = ExtLocDomEnv(id, d(env))(env) in
  .27                              let env'' = Iota(Modify(id, inev(env)))(env') in
  .28                                if env'' = err
  .29                                then (env, ret, ⊥)
  .30                                else ResLocId(env, id, bev(overwrite(env', env'')))
  .31               | inev ∈ EvalExpr(init), bev ∈ EvalStmt(body) }
```

Annotations to *EvalBlock*:

115. This function gives meaning to block statements.

.3–.8 If no initial value of the local variable is provided, the 'special' type environment must be extended with the type information about it, and the body of the block statement must be evaluated in this context. Finally, after this evaluation the old state must be restored (i.e. if the local identifier, *id*, was in scope outside this block statement its old value must be restored). This is done for all statement evaluators for the body of the block statement.

.10–.23 If the local variable in the block statement is initialized by a call of a value returning operation it is slightly more complicated.

.10–.11 If either the initialization part or its type returns an error, or undefined composite type is used, an error is reported by returning the bottom value.

.13–.14 If the mode of the initialization part is exit the entire block statement is left and it will denote the meaning of the initialization.

.15–.16 If the operation called did not return any value, an error is reported by returning the bottom value.

.17–.18 The environment returned from the evaluation of the initialization part must be extended with both the 'special' type environment and the local variable, *id*, which must be bound to the returned value, *val*.

.19–.20 If an error occurred in extending the environment a three-tuple which returns bottom is used.

.22 The old state is restored.

.23 This is done for all possible combinations of evaluators for the initialization part and the body part.

.24–.31 Alternatively the local variable in the block statement is initialized by an expression.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.24–.25 If either the initialization expression yields bottom or the type returns an error or an undefined composite type is used, an error is reported by returning the bottom value.
.26–.27 The environment returned from the evaluation of the initialization part must be extended with both the 'special' type environment and the local variable, *id*, which must be bound to the value from the expression in, *ev(env)*.
.28–.29 If an error occurred in extending the environment it is reported.
.30 The old state is restored.
.31 This is done for all possible combinations of evaluators for the initialization part and the body part.

End of annotations

```
116.   EvalSequence : Sequence → LSEval
116.1  EvalSequence(MkTag('Sequence', stmt_l)) ≙
  .2    if len(stmt_l) = 0
  .3    then { λenv . (env, cont, nil) }
  .4    else { λenv.let (env', mode, val) = sev1(env) in
  .5                 if mode ≠ cont
  .6                 then (env', mode, val)
  .7                 else sev2(env')
  .8         | sev1 ∈ EvalStmt(hd(stmt_l)),
  .9           sev2 ∈ EvalSequence(MkTag('Sequence', tl(stmt_l))) }
```

Annotations to *EvalSequence*:

116. This function gives meaning to sequences of statements. It is defined by induction over the length of the sequence.
.2–.3 If the sequence of statements is empty, its denotation essentially consists of identity.
.4–.6 If the execution of the first statement ends with either an exit flag or returns a value, then the execution of the whole sequence is finished.
.7 Otherwise the rest of the sequence is executed, in the new environment.
.8–.9 This is done for all possible combinations of evaluators for the first statement and the rest of the statements.

End of annotations

```
117.   EvalAssign : Assign → LSEval
117.1  EvalAssign(MkTag('Assign', (lhs, rhs))) ≙
  .2    if ShowTag(rhs) = 'Call'
  .3    then PropS({ λenv.let (env', mode, val) = cev(env) in
  .4                        if mode = exit
  .5                        then { (env', exit, val) }
  .6                        else if val = nil
  .7                             then (env', ret, ⊥)
  .8                             else let env_s = { m(env)
  .9                                              | m ∈ Modify(lhs, val) } in
  .10                                 { if env'' = err ∨ BotEnv(env'')
  .11                                   then (env', ret, ⊥)
  .12                                   else (env'', cont, nil)
  .13                                 | env'' ∈ env_s }
  .14               | cev ∈ EvalCall(rhs) })
  .15   else PropS({ λenv.let env_s = { m(env) | m ∈ Modify(lhs, ev(env)) } in
  .16                        { if env'' = err ∨ BotEnv(env'')
  .17                          then (env, ret, ⊥)
  .18                          else (env'', cont, nil)
  .19                        | env'' ∈ env_s }
  .20               | ev ∈ EvalExpr(rhs) })
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annotations to *EvalAssign*:

117. This function gives meaning to assignment statements[1].

.3–.14 This part is used if the right hand side, *rhs*, of the assign statement is a call of a value returning operation.

.4–.5 If the mode of the operation call is exit the entire define statement is left and it will denote the meaning of the operation call.

.6–.7 If the operation called does not return any value, an error is reported by returning the bottom value.

.8–.13 The left hand side state designator, *lhs*, is modified with the returned value, *val*. If an error occurs in this modification or if a non-strict binding occurs, an error is reported by returning the bottom value. Otherwise a tuple with the resulting environment, cont (for continue), and nil (indicating that no value is present) is returned.

.15–.20 This part is used if the right hand side, *rhs*, of the assign statement is an expression.

.15 All possible environments resulting from modifying an expression evaluator, *ev*, applied with the given environment, *env*, are gathered.

.16–.17 If an error is returned from such a modification or it resulted in a non-strict binding, an error is reported by returning the bottom value.

.18 Otherwise a tuple with the resulting environment, cont (for continue), and nil (indicating that no value is present) is returned.

End of annotations

8.5.5 Conditional Statements

118. $EvalIfStmt : IfStmt \rightarrow LSEval$

118.1 $EvalIfStmt(MkTag(\text{'IfStmt'}, (test, cons, altn))) \triangleq$

.2 $\{ \lambda env.$ **if** $testev(env) \notin BOOL_VAL$

.3 **then** $(env, \underline{\text{ret}}, \bot)$

.4 **else if** $testev(env) = True()$

.5 **then** $consev(env)$

.6 **else** $altnev(env)$

.7 $\mid testev \in EvalExpr(test),\ consev \in EvalStmt(cons),\ altnev \in EvalStmt(altn) \}$

Annotations to *EvalIfStmt*:

118. This function gives meaning to an if-then-else statement.

.2–.3 If the expression evaluator for the *test* expression applied with the given environment does not yield a Boolean value, an error is reported by returning a three-tuple which yields a bottom value for the return component.

.4–.5 If the expression evaluator, *testev*, instead yields *True*() when it is applied with the given environment, the consequence statement, *cons*, is applied with the given environment.

.6 Otherwise the alternative statement, *altn*, is applied with the given environment.

.7 This is done for all possible evaluators for the test expression, the consequence and the alternative statements.

End of annotations

119. $EvalCasesStmt : CasesStmt \rightarrow LSEval$

119.1 $EvalCasesStmt(MkTag(\text{'CasesStmt'}, (sel, altns))) \triangleq$

.2 $PropS(\{ \lambda env.$ **if** $selev(env) = \bot$

.3 **then** $\{(env, \underline{\text{ret}}, \bot)\}$

.4 **else** $\{ sev(env) \mid sev \in ApplyCasesStmt(selev(env), altns) \}$

.5 $\mid selev \in EvalExpr(sel) \})$

Annotations to *EvalCasesStmt*:

[1] As in the *Def* statement an *Assign*ment statement may have side effects by means of calls to value returning operations.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

119. This function gives meaning to a cases statement.
.2–.3 If an evaluator for the selector expression applied with the given environment yields bottom, an error is reported by returning a three-tuple which uses bottom as a returned value.
.4 The meaning of the cases statement is found by applying *ApplyCasesStmt* with the selector value and the list of cases alternatives.

End of annotations

120. $ApplyCasesStmt : VAL \times \mathbb{L}(CaseStmtAltn) \to LSEval$

120.1 $ApplyCasesStmt(val, csa_l) \triangleq$
.2 **if** $\underline{\text{len}}(csa_l) = 0$
.3 **then** $\{ \lambda env.\ (env, \underline{\text{cont}}, \underline{\text{nil}}) \}$
.4 **else** $\{ \lambda env.$**let** $res = caev(env)$ **in**
.5 **if** $res = \underline{\text{unmatch}}$
.6 **then** $c(env)$
.7 **else** res
.8 $\mid caev \in ApplyCaseStmtAltn(val, \underline{\text{hd}}(csa_l)),$
.9 $c \in ApplyCasesStmt(val, \underline{\text{tl}}(csa_l)) \}$

Annotations to *ApplyCasesStmt*:

120. This function applies a selector value to a list of case statement alternatives. The first alternative which contains a pattern which can be successfully matched against the selector value is chosen, and the corresponding loose statement evaluator is returned.
.2–.3 If the list of case statement alternatives is empty no pattern matching has been successful, and the cases statement will terminate normally and return no value (as an IDENT statement).
.4–.9 Otherwise it is checked whether the first case statement alternative matches the selector value, *val*, in the given environment. If this is the case, the loose statement evaluator from the corresponding statement is returned. Alternatively the semantics is defined by means of recursion with the tail of the list of statement alternatives.

End of annotations

121. $ApplyCaseStmtAltn : VAL \times CaseStmtAltn \to LSEval \cup \{ \underline{\text{unmatch}} \}$

121.1 $ApplyCaseStmtAltn(val, MkTag(\text{'CaseStmtAltn'}, (pat, body))) \triangleq$
.2 $PropS(\{ \lambda env.$**let** $s = \{ Restore(env, ColIdSet(pat), boev(env'))$
.3 $\mid boev \in EvalStmt(body),$
.4 $env' \in NonErrPat(pat, val, env) \}$ **in**
.5 **if** $s = \{\ \}$
.6 **then** $\{ \underline{\text{unmatch}} \}$
.7 **else** s
.8 $\})$

Annotations to *ApplyCaseStmtAltn*:

121. This function applies a selector value to a statement case alternative. If the alternative contains a pattern which can be successfully matched against the selector value, *val*, the statement evaluators for the statement from the alternative is returned. Otherwise the token unmatch is returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.5.6 Loop Statements

```
122.    EvalSeqForLoop : SeqForLoop → LSEval
122.1   EvalSeqForLoop(MkTag('SeqForLoop', (cv, dirn, seq, body))) ≙
  .2     PropS({ λenv.if l_ev(env) ∉ LIST_VAL
  .3                  then {(env, ret, ⊥)}
  .4                  else let l_v = if dirn = FORWARDS
  .5                                 then StripSeqTagVal(l_ev(env))
  .6                                 else Reverse(StripSeqTagVal(l_ev(env))) in
  .7                         { sev(env) | sev ∈ DoSeqLoop(cv, l_v, body) }
  .8           | l_ev ∈ EvalExpr(seq) })
```

Annotations to *EvalSeqForLoop*:

122. This function gives meaning to *For Loop*s running over sequences. The semantics is affected by the fact that it is possible that the sequence expression may be loosely specified. However, the sequence is considered static (in the sense that it is **not** changed by the loop evaluation). It is also worth noting that the loop variables (the pattern identifiers in *cv*) are localized to the loop. Thus, the scope of the loop variables is restricted to the loop itself which is different from current practice in most programming languages like C or Pascal.

.2 – .3 If the expression which is supposed to denote a sequence does not denote a sequence, an error is reported by returning the bottom value.

.4 – .6 Otherwise a sequence of values is created; it takes into account that the order in which the sequence should be processed is indicated by the flag FORWARDS or BACKWARDS.

.7 An auxiliary function, *DoSeqLoop* is applied to process the loop.

.8 This is done for all possible evaluators of the syntactic sequence expression.

End of annotations

```
123.    EvalSetForLoop : SetForLoop → LSEval
123.1   EvalSetForLoop(MkTag('SetForLoop', (cv, set, body))) ≙
  .2     PropS({ λenv.if s_ev(env) ∉ SET_VAL
  .3                  then {(env, ret, ⊥)}
  .4                  else let s_v = StripSetTagVal(s_ev(env)) in
  .5                         ⋃{ { sev(env) | sev ∈ DoSeqLoop(cv, l_v, body) }
  .6                            | l_v ∈ SetToSeq(s_v) }
  .7           | s_ev ∈ EvalExpr(set) })
```

Annotations to *EvalSetForLoop*:

123. This function gives meaning to *For Loop*s running over sets. The semantics is affected by the fact that it is possible that the set expression is loosely specified. However, the set is considered static (in the sense that it is **not** changed by the loop evaluation). It is also worth noting that the loop variables (the pattern identifiers in *cv*) are localized to the loop. Thus, the scope of the loop variables is restricted to the loop itself which is different from current practice in most programming languages like C or Pascal.

.2 – .3 For all possible evaluators of the set expression it is checked whether the loop should be entered at all. It is only entered if the denotation is a set value.

.5 The loop is entered by using the function *DoSeqLoop* where the set value has been transformed into all possible sequences, indicating an order in which to match an element value against the pattern, *cv*.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
124.   EvalIndexForLoop : IndexForLoop → LSEval
124.1  EvalIndexForLoop(MkTag('IndexForLoop', (cv, lb, ub, by, body))) ≙
  .2    PropS({ λenv.if lbev(env) ∉ NUM_VAL ∨
  .3                    ubev(env) ∉ NUM_VAL ∨
  .4                    stev(env) ∉ NUM_VAL
  .5                 then { (env, ret, ⊥) }
  .6                 else let lb_v = StripNumTagVal(lbev(env)),
  .7                          ub_v = StripNumTagVal(ubev(env)),
  .8                          st_v = StripNumTagVal(stev(env)) in
  .9                      if lb_v ∉ ℤ ∨ ub_v ∉ ℤ ∨ st_v ∉ ℤ
  .10                     then { (env, ret, ⊥) }
  .11                     else let seq = StepsToSeq(lb_v, ub_v, st_v) in
  .12                          if seq = err
  .13                          then { (env, ret, ⊥) }
  .14                          else let pat = MkTag('PatternId', cv) in
  .15                               { sev(env)
  .16                               | sev ∈ DoSeqLoop(pat, seq, body) }
  .17          | lbev ∈ EvalExpr(lb), ubev ∈ EvalExpr(ub), stev ∈ EvalExpr(by) })
```

Annotations to *EvalIndexForLoop*:

124. This function gives meaning to *For Loops* with an *Index*. The semantics here is affected by the possibility that all the expressions for the lower bound, the upper bound and the step value may be loosely specified. However, these bounds are considered static (in the sense that they are **not** changed by the loop evaluation). It is also worth noting that the loop variable is localized to the loop. Thus, the scope of the loop variable is restricted to the loop itself which is different from current practice in most programming languages like C or Pascal.

.2–.10 It is checked whether an evaluator for the lower bound, the upper bound and the step size all are integers. If this is not the case, an error is reported by returning the bottom value.

.11 A sequence of values which the *cv* identifier must be bound to is created by applying *StepsToSeq* with the value of the lower bound, *lb_v*, the value of the upper bound, *ub_v*, and the step value, *st_v*.

.12–.13 If the step value is 0, an error is reported (from *StepsToSeq*) and then a three tuple returning the bottom value is used.

.16 The loop is entered by using the function *DoSeqLoop*.

.17 This is done for all possible expression evaluators for the lower bound, the upper bound, and the step size.

End of annotations

```
125.   DoSeqLoop : Pattern × 𝕃(VAL) × Stmt → LSEval
125.1  DoSeqLoop(pat, l_v, body) ≙
  .2    if len(l_v) = 0
  .3    then { λenv . (env, cont, nil) }
  .4    else let ids = ColIdSet(pat) in
  .5         PropS({ λenv.{ let (env'', mode, val) = sev(env') in
  .6                        if mode ≠ cont
  .7                        then Restore(env, ids, sev(env'))
  .8                        else Restore(env, ids, sev'(env''))
  .9                      | env' ∈ NonErrPat(pat, hd(l_v), env),
  .10                       sev' ∈ DoSeqLoop(pat, tl(l_v), body) }
  .11                | sev ∈ EvalStmt(body) })
```

Annotations to *DoSeqLoop*:

125. This is an auxiliary function used by *EvalSeqForLoop*, *EvalSetForLoop*, and *EvalIndexForLoop* to give meaning to a loop given the denotation of the sequence, the pattern which each element of the sequence is matched against, and the body statement which is executed within each turn of the loop. This function

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

is defined by induction over the length of the sequence.

.2–.3 If the sequence to iterate over is empty then the original environment is returned.

.9 Otherwise a successful matching of the pattern against the first value of the sequence is considered. Recall that if no matching exists, the constructed set will be empty and the function *PropS* will indicate an error by returning a three-tuple which returns with the bottom value.

.5–.7 If the execution either ends with an exit flag or returns a value, then the computation is finished.

.8–.10 Otherwise the function is defined recursively on the rest of the sequence.

.11 This is done for all possible evaluators of the *body* of the sequence loop statement.

End of annotations

```
126.   EvalWhileLoop : WhileLoop → LSEval
126.1  EvalWhileLoop(MkTag('WhileLoop', (test, body))) ≙
  .2    let LoopDen = ENV ≃→ ((ENV × MODE × (VAL ∪ {nil})) × VAL),
  .3        BodyDen = {λenv.let (env', mode, val) = boev(env) in
  .4                              ((env', mode, val), tev(env'))
  .5                    | boev ∈ EvalStmt(body), tev ∈ EvalExpr(test)} in
  .6    let comp = λ comp_one ∈ BodyDen, comp_rest ∈ LoopDen .λenv.
  .7                 let ((env', mode, val'), val) = comp_one(env) in
  .8                   if mode ≠ cont
  .9                   then ((env', mode, val'), val)
  .10                  else if val ∉ BOOL_VAL
  .11                            then ((env, ret, ⊥), ⊥)
  .12                            else if val = False()
  .13                                      then ((env', cont, nil), False())
  .14                                      else if env' ∈ δ0(comp_rest)
  .15                                               then comp_rest(env') in
  .16     {λenv.if tev(env) ∉ BOOL_VAL
  .17           then (env, ret, ⊥)
  .18           else if tev(env) = False()
  .19                then (env, cont, nil)
  .20                else if env ∉ δ0(comp_rest)
  .21                     then (env, ret, ⊥)
  .22                     else π1(comp_rest(env))
  .23      | tev ∈ EvalExpr(test), comp_rest ∈ NonDetIter(BodyDen, comp)}
```

Annotations to *EvalWhileLoop*:

126. This function gives meaning to *While* loop statements.

.2 The domain used to give semantics to a loop is defined. This is a domain of partial functions from an environment to a pair, where the first component of the pair is the result of applying a statement evaluator with an environment, and the second component is the value of the *test* expression after an evaluation of the body of the loop. It should be noted that instead of extending the range of this domain with another 'bottom' value, a locally defined partial function is used which is defined prior to application.

.3–.5 *BodyDen* is a subset of *LoopDen*. It describes all possible computations done on the body in one step of the while loop.

.3–.4 The *body* statement is evaluated and then the *test* expression is evaluated in a new environment (resulting from the evaluation of the body).

.6–.17 *comp* defines an appropriate composition of a total function from *BodyDen* with a partial function from *LoopDen* (the rest of the loop computation).

.7–.9 If an execution of the body (which is the first coordinate of the result) has ended with an exit or returned a value (by means of a return statement) then the execution of the loop is finished.

.10–.11 Alternatively, if the evaluation of the *test* expression does not denote a Boolean value, then an error is reported.

.12–.13 If *test* in the given environment evaluates to *False*(), then the execution of the loop is finished.

.14–.15 Otherwise, the second function, *comp_rest*, responsible for the execution of the rest of the loop is invoked.

.16–.23 The denotation of the *While* loop is defined.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.16–.17 If the evaluation of the *test* expression does not denote a Boolean value, then an error is returned.
.18–.19 If *test* evaluates to *False*() (before entering the loop), then the execution of the loop statement is finished.
.22 Otherwise a non-deterministic iteration over executions of *body* is invoked.
.20–.21 If the above iteration happens to be infinite then an error is reported instead by returning a three-tuple where the bottom value is returned. Note that this test ensures that the partial function *comp_rest* is defined for the given environment, *env*.

End of annotations

8.5.7 Non-Deterministic Sequences

127. $EvalNonDetStmt : NonDetStmt \rightarrow LSEval$

127.1 $EvalNonDetStmt(MkTag(\text{'NonDetStmt'}, stmt_l)) \triangleq$
.2 $\bigcup\{ EvalSequence(MkTag(\text{'Sequence'}, s_l)) \mid s_l \in Permutations(stmt_l) \}$

Annotations to *EvalNonDetStmt*:

127. This function gives meaning to a list of statements which are executed in a non-deterministic order. Please note that this is not parallelism. All statements inside the *NonDetStmt* are evaluated atomically in a non-deterministic order.
.2 All possible permutations of the list of statements are collected by means of *Permutations*, and for each of these a sequence statement is created and its meaning is found.

End of annotations

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.5.8 Call and Return Statements

```
128.    EvalCall : Call → LSEval
128.1   EvalCall(MkTag('Call', (oprt, arg, st))) ≙
  .2     PropS({ λenv.if oprt ∉ dom(env) ∨ env(oprt) ∉ OPVAL
  .3                  then { (env, ret, ⊥) }
  .4                  else ⋃{ if st = nil
  .5                            then { (env, mode, out_v) }
  .6                            else let m_s = { m(env) | m ∈ Modify(st, out_st) } in
  .7                                   { (e, mode, out_v) | e ∈ m_s · e ≠ err } ∪
  .8                                   { (env, ret, ⊥) | err ∈ m_s }
  .9                                 | (in_v, in_st, out_st, out_v, mode) ∈ env(oprt),
  .10                                  in_st ∈ if st = nil
  .11                                          then let st_id = GetStateTypeId(env) in
  .12                                                 if st_id = nostate
  .13                                                 then { nil }
  .14                                                 else { env(GetStateValId(CUR, st_id)) }
  .15                                          else { ev(env) | ev ∈ LookUp(st) },
  .16                                  in_v ∈ { ev(env) | ev ∈ EvalExpr(arg) } } ∪
  .17                              { (env, ret, ⊥)
  .18                              | in_st ∈ if st = nil
  .19                                        then let st_id = GetStateTypeId(env) in
  .20                                               if st_id = nostate
  .21                                               then { nil }
  .22                                               else { env(GetStateValId(CUR, st_id)) }
  .23                                        else { ev(env) | ev ∈ LookUp(st) },
  .24                                in_v ∈ { ev(env) | ev ∈ EvalExpr(arg) } ·
  .25                                ∀ out_st, out_v ∈ VAL, m ∈ MODE ·
  .26                                 (in_v, in_st, out_st, out_v, m) ∉ env(oprt) }
  .27       })
```

Annotations to *EvalCall*:

128. This function gives meaning to the *Call* of an operation.

.2–.3 If either the name of the called operation, *oprt*, is not in the domain of the given environment, or if it is in the domain and it is not bound to an operation value, an error is reported by returning a three-tuple which uses bottom as a return value.

.9 The operation value which can be looked up in the given environment is a relation. Each component of the relation is a tuple with five elements (input value, input state, output state, output value, mode).

.10–.16 The input value and the input state value restrict the interesting components of the relation (from line .9).

.4–.5 If there is no state a three-tuple with the environment (unchanged), the mode returned from the operation, and the output value from the operation, are returned.

.6–.7 Otherwise the state designator is present, the environment is modified according to the output state value from the operation. All such possible modifications are then used as first components of the returned three-tuples.

.17–.26 If the input value or state are not possible for the given operation (i.e. if those components do not exist together in the relation which the operation denotes), an error is reported by returning a three-tuple which uses bottom as a return value.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

129. *EvalReturnStmt* : *ReturnStmt* → *LSEval*

129.1 $EvalReturnStmt(MkTag(\text{'ReturnStmt'}, expr)) \triangleq$
.2 **if** $expr = \underline{\text{nil}}$
.3 **then** $\{\lambda env.(env, \underline{\text{ret}}, \underline{\text{nil}})\}$
.4 **else** $\{\lambda env.(env, \underline{\text{ret}}, ev(env)) \mid ev \in EvalExpr(expr)\}$

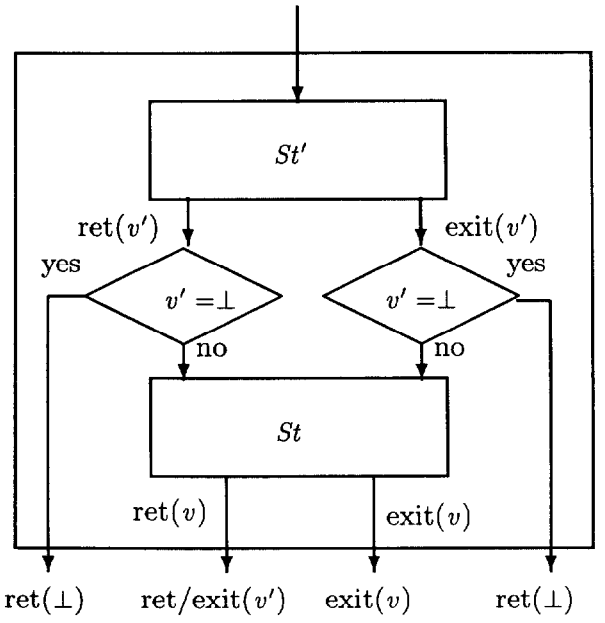
Annotations to *EvalReturnStmt*:
129. This function gives meaning to return statements.
.2–.3 If no value is returned the return statement denotes one statement evaluator with nil as a value component.
.4 Otherwise there will be one statement evaluator per expression evaluator.
End of annotations

8.5.9 Exception Handling Statements

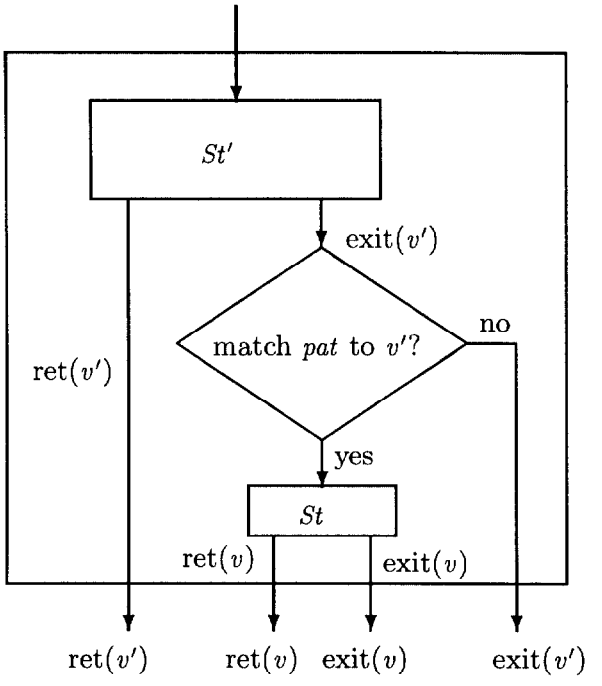
Flow Diagrams for Handler Statements

Before describing the semantics for handler statements the execution order is illustrated by means of three flow diagrams (inspired from Daw91), because experience shows that it is difficult to understand the strategy in the beginning. The outer box in each diagram represents the effect of the entire handler statement. Termination, with value v, is represented by arrows labelled with "exit(v)" or "ret(v)" for abnormal or normal termination respectively. In order to make diagrams understandable the values (v's) have been indexed in the same way as the statement which they are returned from.

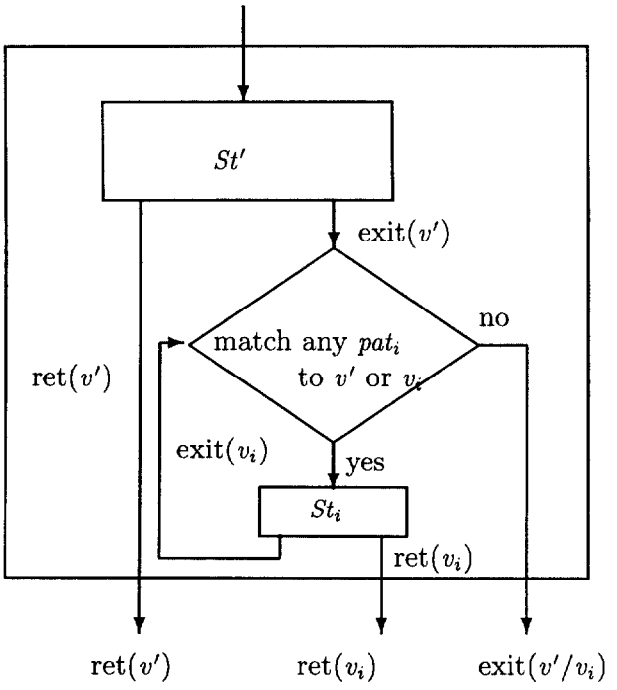
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale



always St in St'



trap pat with St in St'



tixe $\{pat_1 \mapsto St_1, \ldots, pat_n \mapsto St_n\}$ in St'

□

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
130.    EvalAlways : Always → LSEval
130.1   EvalAlways(MkTag('Always', (post, body))) ≙
  .2     { λenv.let (env', mode, val) = boev(env) in
  .3             if val = ⊥
  .4             then boev(env)
  .5             else let (env'', mode', val') = poev(env') in
  .6                     if mode' ≠ exit
  .7                     then (env'', mode, val)
  .8                     else poev(env')
  .9      | boev ∈ EvalStmt(body), poev ∈ EvalStmt(post) }
```

Annotations to *EvalAlways*:

130. This function gives meaning to *Always* statements.

.2–.3 In order to ensure that the *Always* statement is strict wrt. its body the post statement is not evaluated if bottom is returned from the body statement.

.6–.7 If the mode of the 'post' statement evaluator is not an exit, the mode and value from the body statement evaluator is used together with the environment from the 'post' statement evaluator.

.8 If an exception is raised inside the 'post' statement the 'post' statement evaluator is used directly.

.9 This is done for all possible statement evaluators: for both the 'post' statement and the body statement.

End of annotations

```
131.    EvalTrapStmt : TrapStmt → LSEval
131.1   EvalTrapStmt(MkTag('TrapStmt', (pat, post, body))) ≙
  .2     let ids = ColIdSet(pat) in
  .3       PropS({ λenv.let (env', mode, val) = boev(env) in
  .4                    if mode ≠ exit ∨ val = nil
  .5                    then { (env', mode, val) }
  .6                    else let s = { Restore(env', ids, poev(nenv))
  .7                                 | poev ∈ EvalStmt(post),
  .8                                   nenv ∈ NonErrPat(pat, val, env') } in
  .9                         if s = { }
  .10                        then { (env', exit, val) }
  .11                        else s
  .12           | boev ∈ EvalStmt(body) })
```

Annotations to *EvalTrapStmt*:

131. This function gives meaning to *Trap* statements.

.4–.5 If the body evaluator does not raise an exception, then the body evaluator is simply returned. This is also done if an exception is raised without a corresponding value. Such exceptions can only be trapped by an always statement.

.6–.8 A set of results, from applying a 'post' statement evaluator with the given environment and restoring the old bindings (if any) of the pattern identifiers from *pat*, is created.

.9–.11 If the set, *s*, is empty no successful pattern matching is found in this trap statement, and the exception is passed on to an exception handler at an outer level. Otherwise *s* can simply be used directly.

.12 This is done for all possible statement evaluators for the body statement.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
132.    EvalRecTrapStmt : RecTrapStmt → LSEval
132.1   EvalRecTrapStmt(MkTag('RecTrapStmt', (traps, body))) ≙
  .2     let ids = ⋃{ ColIdSet(p) | p ∈ dom(traps) } in
  .3       let RecDen = ENV × VAL ⇸ ((ENV × MODE × (VAL ∪ { nil }))
  .4                                 × { match, unmatch }),
  .5           TrapDen = PropS({ λ env ∈ ENV, val ∈ VAL .
  .6                               let s = ⋃{ { Restore(env, ids, poev(nenv))
  .7                                          | poev ∈ EvalStmt(traps(p)),
  .8                                            nenv ∈ NonErrPat(p, val, env) }
  .9                                        | p ∈ dom(traps) } in
  .10                                if s = { }
  .11                                then { ((env, exit, val), unmatch) }
  .12                                else { (e, match) | e ∈ s } }) in
  .13       let comp = λ comp_one ∈ TrapDen, comp_rest ∈ RecDen .
  .14                  λenv ∈ ENV, val ∈ VAL .
  .15                  let ((env', mode', val'), m) = comp_one(env, val) in
  .16                    if m = unmatch ∨ mode' ≠ exit ∨ val = nil
  .17                    then ((env', mode', val'), m)
  .18                    else if (env', val') ∈ δ0(comp_rest)
  .19                         then comp_rest(env', val') in
  .20         { λenv.let (env', mode, val) = boev(env) in
  .21                if mode ≠ exit ∨ val = nil
  .22                then (env', mode, val)
  .23                else if (env', val) ∉ δ0(comp_rest)
  .24                     then (env, ret, ⊥)
  .25                     else π1(comp_rest(env', val))
  .26          | boev ∈ EvalStmt(body), comp_rest ∈ NonDetIter(TrapDen, comp) }
```

Annotations to *EvalRecTrapStmt*:

132. This function gives meaning to *Rec*ursive *Trap* statements.

.3 The domain used to give semantics to a recursive usage of a trap statement is defined. This is a domain of partial functions from a pair of environments and a value (to be matched against the patterns in the domain of *traps*) to a pair, where the first component of the pair is the result of applying a statement evaluator with an environment, and the second component indicates whether a successful pattern matching was encountered. It should also be noted that instead of extending the range of this domain with another "bottom" value, a locally defined partial function is used, which is defined prior to application.

.5–.12 *TrapDen* is a subset of *RecDen*. It describes all possible computations done when matching an exit value to a pattern and then executing the corresponding trap handler statement.

.6–.9 For all patterns in the different traps it is tested whether they have successful matchings against the exit value in the given environment. If this is the case the new resulting environment *nenv* is applied to all evaluators of the trap statement corresponding to the matching pattern. So *s* is the set of all possible evaluations of the handler statements with a given environment and a given exit value (which must match a trap pattern).

.10–.11 If this set is empty, i.e. no pattern matches the exit value, then the original environment is returned with a flag indicating that the matching failed.

.12 Otherwise the successful matching is reported for all elements of *s*.

.13–.19 *comp* defines an appropriate composition of a total function from *TrapDen* (used for one 'computation' of the recursive trap part) with a partial function from *RecDen* (the recursive call of the trap statement 'computing' the rest of the recursive call).

.15–.17 If either no pattern has matched the exit value, or the execution of the statement corresponding to the matching pattern has ended with a flag different from exit, the execution of *RecTrapStmt* is finished. This also happens if an exception is raised without a corresponding value. Such exceptions can only be trapped by an always statement.

.18–.19 Otherwise the second function, responsible for the recursive call of the *RecTrapStmt* is invoked.

.20–.26 The denotation of *RecTrapStmt* is defined.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.20–.22 If the execution of the *body* statement ends with a flag different from exit this ends the execution of the whole statement. This is also done if an exception is raised without a corresponding value. Such exceptions can only be trapped by an always statement.
.25 Otherwise a non-deterministic iteration over executions of recursive calls of *RecTrapStmt* is invoked.
.23–.24 If the above iteration happens to be infinite then an error is reported instead. Note that this test ensures that the partial function *comp_rest* is defined for the given pair of environments and values, (*env'*, *val*).

End of annotations

```
133.    EvalExit : Exit → LSEval
133.1   EvalExit(MkTag('Exit', expr)) ≜
  .2      if expr = nil
  .3      then { λenv.(env, exit, nil) }
  .4      else let lev = EvalExpr(expr) in
  .5             { λenv. if ev(env) = ⊥
  .6                     then (env, ret, ev(env))
  .7                     else (env, exit, ev(env))
  .8             | ev ∈ lev}
```

Annotations to *EvalExit*:

133. This function gives meaning to exit statements.
.2–.3 If no value is returned the exit statement denotes one statement evaluator with nil as a value component.
.4–.8 Otherwise there will be one statement evaluator per expression evaluator. An exit exception is raised if the exit expression yields a proper value. Otherwise the exit statement denotes a statement evaluator which returns bottom.

End of annotations

8.6 Patterns and Bindings

In this clause functions for evaluating patterns and bindings are presented. Before going on and reading these functions it is important to understand the semantic difference between patterns and bindings.

A binding consists of an arbitrary pattern, and either a type or a set. The purpose of such a binding is to match the pattern against all elements in the set (or type). In contrast to this, the purpose of a pattern is to be matched against one value[1)]. So the result of applying a binding evaluator with an environment can be seen as a set of possible matchings which, for example, is used by quantified expressions. It should also be noted here that only successful matchings and cases where a 'real' error has occurred, are considered[2)]. Thus, in this way there can exist binding evaluators which for a given environment return an empty set of pattern matchings (in the case where the binding completely fails). This case is treated differently in the different occurrences of binding in VDM-SL. In the table below the different strategies used in case the binding completely fails in all the different places where bindings can occur are listed:

[1)] When it is possible in the concrete syntax of VDM-SL to supply extra type information about a pattern, it will look like a binding. However, if it is supposed to be matched against one value it should still be thought of as a pattern. In the CAS this kind of patterns 'constrained patterns' have been called. A type constrained pattern, *TypeConstrPattern*, and a set constrained pattern, *SetConstrPattern*, exist.

[2)] A 'real' error occurs if either the pattern contains a match value which yields bottom, or the set expression or the type from the binding yields a value which is not a set value or the type is erroneous, respectively.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Set comprehension	An empty set is returned.
Sequence comprehension	An empty sequence is returned.
Map comprehension	An empty map is returned.
Let be expressions	Bottom is returned.
Let be statements	(*env*, ret, ⊥) is returned.
Universal quantification	True is returned.
Existential quantification	False is returned.
Unique quantification	False is returned.
Iota Expression	Bottom is returned.

In the cases where a 'real' error has occurred strictness is ensured and bottom will be returned.

8.6.1 Patterns

134. $EvalPattern : Pattern \to LPatEval$

134.1 $EvalPattern(pat) \triangleq$
.2 **cases** $ShowTag(pat)$:
.3 $'PatternId' \to EvalPatternId(pat)$,
.4 $'MatchVal' \to EvalMatchVal(pat)$,
.5 $'SeqEnumPattern' \to EvalSeqEnumPattern(pat)$,
.6 $'SeqConcPattern' \to EvalSeqConcPattern(pat)$,
.7 $'TuplePattern' \to EvalTuplePattern(pat)$,
.8 $'RecordPattern' \to EvalRecordPattern(pat)$,
.9 $'SetEnumPattern' \to EvalSetEnumPattern(pat)$,
.10 $'SetUnionPattern' \to EvalSetUnionPattern(pat)$,
.11 $'SetConstrPattern' \to EvalSetConstrPattern(pat)$,
.12 $'TypeConstrPattern' \to EvalTypeConstrPattern(pat)$

Annotations to *EvalPattern*:
134. This function gives meaning to patterns.
.3–.12 The meaning of different kinds of patterns is split into sub-functions.
End of annotations

135. $EvalPatternId : PatternId \to LPatEval$

135.1 $EvalPatternId(MkTag('PatternId', id)) \triangleq$
.2 $\{\lambda val.\lambda env.$**if** $id = \underline{\text{nil}}$
.3 **then** $\{\mapsto\}$
.4 **else** $\{id \mapsto val\}\}$

Annotations to *EvalPatternId*:
135. This function gives meaning to pattern identifiers.
.2–.3 If it is a 'don't care' pattern, an empty map is returned.
End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
136.    EvalMatchVal : MatchVal → LPatEval
136.1   EvalMatchVal(MkTag('MatchVal', expr)) ≜
  .2     { λval.λenv.if val = ⊥ ∨ eval(env) = ⊥
  .3                 then err
  .4                 else if eval(env) = val
  .5                      then {↦}
  .6                      else unmatch
  .7      | eval ∈ EvalExpr(expr) }
```

Annotations to *EvalMatchVal*:

136. This function gives meaning to 'match value patterns'.
.2–.3 If either the value of the match value yields bottom a 'real' error is reported.
.4–.6 The matching succeeds if the value, *val*, is equal to an application of an expression evaluator with the given environment; otherwise the matching fails.
.7 This is done for all evaluators for the expression in the 'match value pattern', *expr*.

End of annotations

```
137.    EvalSeqEnumPattern : SeqEnumPattern → LPatEval
137.1   EvalSeqEnumPattern(MkTag('SeqEnumPattern', pat_l)) ≜
  .2     EvalPatternList(pat_l)
```

Annotations to *EvalSeqEnumPattern*:

137. This function gives meaning to sequence enumeration patterns.
.2 It is defined simply by means of *EvalPatternList*.

End of annotations

```
138.    EvalSeqConcPattern : SeqConcPattern → LPatEval
138.1   EvalSeqConcPattern(MkTag('SeqConcPattern', (p1, p2))) ≜
  .2     let mul_pev_s = { λval.λenv.if val ∈ LIST_VAL
  .3                                then let val_p_s = ChopSeqVal(val) in
  .4                                     { let venv_s = { pev1(v1)(env),
  .5                                                      pev2(v2)(env) } in
  .6                                       if {err, unmatch} ∩ venv_s ≠ { }
  .7                                       then if err ∈ venv_s
  .8                                            then err
  .9                                            else unmatch
  .10                                      else if Compatible(venv_s)
  .11                                           then Merge(venv_s)
  .12                                           else unmatch
  .13                                     | (v1, v2) ∈ val_p_s}\{unmatch}
  .14                               else if val = ⊥
  .15                                    then { EvalBotPat({ p1, p2 }) }
  .16                                    else {unmatch}
  .17                     | pev1 ∈ EvalPattern(p1), pev2 ∈ EvalPattern(p2) } in
  .18     PropToLPatEval(mul_pev_s)
```

Annotations to *EvalSeqConcPattern*:

138. This function gives meaning to sequence concatenation patterns.
.2–.15 A set of 'multi-pattern' evaluators is defined (which have the type: $\mathbb{P}(VAL \to ENV \to \mathbb{P}(VEnv \cup \{\underline{\text{err}}\}))$).
.2–.3 If the value is a sequence value, it is 'chopped' into all possible pairs of sequences which joined together is equal to the given sequence value.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.4–.11 A set of all possible binding environments for a particular pair of sequence values, is created. If both (component) pattern evaluators, pev_1 and pev_2, are matched successfully with their corresponding sequence value from the pair, and they return compatible value environments the entire sequence concatenation pattern has been successfully matched and the compatible value environments are merged together and returned.

.12–.14 If the value is not a sequence value, it can either be the bottom value (and it that case all pattern identifiers from the pattern will be matched to bottom by *EvalBotPat*) or it can be a normal value, and then the matching fails.

.16 The looseness inside the 'multi-pattern' evaluator set is propagated to the outermost level. Notice that if the set of bindings at the innermost level is empty, an unmatch is returned.

End of annotations

```
139.    EvalTuplePattern : TuplePattern → LPatEval
139.1   EvalTuplePattern(MkTag('TuplePattern', fields)) ≙
  .2     { λval.λenv.if val ∈ TUPLE_VAL
  .3                 then let MkTag('tuple', l) = val in
  .4                        pev(MkTag('seq', l))(env)
  .5                 else if val = ⊥
  .6                      then EvalBotPat(elems(fields))
  .7                      else unmatch
  .8       | pev ∈ EvalPatternList(fields) }
```

Annotations to *EvalTuplePattern*:

139. This function gives meaning to tuple patterns.

.2–.4 If the value, *val*, is a tuple value, its list of values are matched against the list of patterns corresponding to the fields.

.5–.7 If the value is not a tuple value, it can either be the bottom value (and in that case all pattern identifiers from the pattern will be matched to bottom by *EvalBotPat*) or it can be a normal value, and then the matching fails.

.8 This is done for all possible pattern evaluators for the list of patterns, *fields*.

End of annotations

```
140.    EvalRecordPattern : RecordPattern → LPatEval
140.1   EvalRecordPattern(MkTag('RecordPattern', (id, fields))) ≙
  .2     { λval.λenv.if val ∈ RECORD_VAL
  .3                 then let MkTag('record', MkTag(id', l)) = val in
  .4                        if id ≠ id'
  .5                        then unmatch
  .6                        else pev(MkTag('seq', l))(env)
  .7                 else if val = ⊥
  .8                      then EvalBotPat(elems(fields))
  .9                      else unmatch
  .10      | pev ∈ EvalPatternList(fields) }
```

Annotations to *EvalRecordPattern*:

140. This function gives meaning to record patterns.

.2–.6 If the value, *val*, is a record value, its list of values are matched against the list of patterns corresponding to the fields, given that the record pattern and the record value have the same tag. Otherwise the matching fails.

.7–.9 If the value is not a record value, it can either be the bottom value (and in that case all pattern identifiers from the pattern will be matched to bottom by *EvalBotPat*) or it can be a normal value, and then the matching fails.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.10 This is done for all possible pattern evaluators for the list of patterns, *fields*.

End of annotations

```
141.   EvalSetEnumPattern : SetEnumPattern → LPatEval
141.1  EvalSetEnumPattern(MkTag('SetEnumPattern', els)) ≙
  .2    let pev_s_l = [ EvalPattern(els(i)) | i ∈ inds(els) ] in
  .3      let pev_l_s = { pev_l | pev_l ∈ 𝕃(PatEval) ·
  .4                              len(pev_l) = len(pev_s_l) ∧
  .5                              ∀i, j ∈ inds(pev_l) · pev_l(i) ∈ pev_s_l(i) ∧
  .6                                                   i ≠ j ⇒ pev_l(i) ≠ pev_l(j) } in
  .7        let mul_pev_s =
  .8              { λval.λenv.if val ∈ SET_VAL
  .9                          then let val_s = StripSetTagVal(val) in
  .10                                { let venv_s = { el_pev_l(i)(match(i))(env)
  .11                                               | i ∈ inds(el_pev_l) } in
  .12                                  if { err, unmatch } ∩ venv_s ≠ { }
  .13                                  then if err ∈ venv_s
  .14                                       then err
  .15                                       else unmatch
  .16                                  else if Compatible(venv_s)
  .17                                       then Merge(venv_s)
  .18                                       else unmatch
  .19                                | match ∈ 𝕄(inds(el_pev_l), val_s) ·
  .20                                  Injective(match) ∧ dom(match) = inds(el_pev_l) }\{ unmatch }
  .21                          else if val = ⊥
  .22                               then { EvalBotPat(els) }
  .23                               else { unmatch }
  .24              | el_pev_l ∈ pev_l_s } in
  .25        PropToLPatEval(mul_pev_s)
```

Annotations to *EvalSetEnumPattern*:

141. This function gives meaning to set enumeration patterns.

.2 The meaning of each element in the set enumeration pattern is found and combined into a set of sets of pattern evaluators.

.3–.6 A set of lists of pattern evaluators is created. Each list pattern evaluators correspond to the given pattern list, *els*, where an evaluator is given for each element in the list. The list of patterns is supposed to be matched against a set value with the same size, where each pattern match the corresponding element in the set value. Therefore, are lists with duplicate elements also left out.

.7–.22 A set of 'multi-pattern' evaluators is defined (which have the type: $\mathbb{P}(VAL \to ENV \to \mathbb{P}(VEnv \cup \{ \underline{\text{err}} \}))$).

.8–.9 If the value is a set value, the elements of the set are extracted in *val_s*.

.10–.18 A set of all possible binding environments for a particular list of element evaluators, *el_pev_l* is created. The set is indexed with an injective matching from (element) pattern evaluators to (element) values from the set value. If all (element) pattern evaluators are matched successfully to their corresponding value from the injective mapping, and they return compatible value environments, the entire set enumeration pattern has been successfully matched and the compatible value environments are merged together and returned.

.19–.21 If the value is not a set value, it can either be the bottom value (and in that case all pattern identifiers from the pattern will be matched to bottom by *EvalBotPat*) or it can be a normal value, and then the matching fails.

.23 The looseness inside the 'multi-pattern' evaluator set is propagated to the outermost level. Notice that if the set of bindings at the innermost level is empty, an unmatch is returned.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
142.    EvalSetUnionPattern : SetUnionPattern → LPatEval
142.1   EvalSetUnionPattern(MkTag('SetUnionPattern', (p1, p2))) ≜
  .2     let mul_pev_s = { λval.λenv.if val ∈ SET_VAL
  .3                                then let pos_val_p_s = ChopSetVal(val) in
  .4                                       { let venv_s = { pev1(s1)(env), pev2(s2)(env) } in
  .5                                         if { err, unmatch } ∩ venv_s ≠ { }
  .6                                         then if err ∈ venv_s
  .7                                              then err
  .8                                              else unmatch
  .9                                         else if Compatible(venv_s)
  .10                                             then Merge(venv_s)
  .11                                             else unmatch
  .12                                       | (s1, s2) ∈ pos_val_p_s }\{ unmatch }
  .13                               else if val = ⊥
  .14                                    then { EvalBotPat({ p1, p2 }) }
  .15                                    else { unmatch }
  .16                     | pev1 ∈ EvalPattern(p1), pev2 ∈ EvalPattern(p2) } in
  .17     PropToLPatEval(mul_pev_s)
```

Annotations to *EvalSetUnionPattern*:

142. This function gives meaning to set union patterns.

.2–.15 A set of 'multi-pattern' evaluators is defined (which have the type: $\mathbb{P}(VAL \to ENV \to \mathbb{P}(VEnv \cup \{\underline{\text{err}}\}))$).

.2–.3 If the value is a set value, it is 'chopped' into all possible pairs of sets which union is equal to the set value (notice that it is not required that these set-values are non-empty or disjoint).

.4–.11 A set of all possible binding environments for a particular pair of set values is created. If both (component) pattern evaluators, pev_1 and pev_2, are matched successfully with their corresponding set value from the pair, and they return compatible value environments, the entire set union pattern has been successfully matched and the compatible value environments are merged together and returned.

.12–.14 If the value is not a set value, it can either be the bottom value (and in that case all pattern identifiers from the pattern will be matched to bottom by *EvalBotPat*) or it can be a normal value, and then the matching fails.

.16 The looseness inside the 'multi-pattern' evaluator set is propagated to the outermost level. Notice that if the set of bindings at the innermost level is empty, an unmatch is returned.

End of annotations

```
143.    EvalSetConstrPattern : SetConstrPattern → LPatEval
143.1   EvalSetConstrPattern(MkTag('SetConstrPattern', (pat, set))) ≜
  .2     { λval.λenv.let v = ev(env) in
  .3                 if v ∈ SET_VAL
  .4                 then let s = StripSetTagVal(v) in
  .5                        if val ∈ (s ∪ { ⊥ })
  .6                        then pev(val)(env)
  .7                        else unmatch
  .8                 else err
  .9      | ev ∈ EvalExpr(set), pev ∈ EvalPattern(pat) }
```

Annotations to *EvalSetConstrPattern*:

143. This function gives meaning to set constrained patterns.

.3–.4 If an evaluator for the expression, *set*, is a set value, the elements of the set are extracted in *s*.

.5–.7 If the given value, *val*, belongs to this set or is the bottom value, it is given as arguments to a pattern evaluator for *pat*. Otherwise the matching fails.

.9 This is done for all expression evaluators for the expression, *set*, and for all pattern evaluators for the pattern, *pat*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

End of annotations

144. $EvalTypeConstrPattern : TypeConstrPattern \rightarrow LPatEval$

144.1 $EvalTypeConstrPattern(MkTag(\text{'TypeConstrPattern'}, (pat, type))) \triangleq$
.2 $\{ \lambda val.\lambda env.$**let** $d = EvalType(type)(env)$ **in**
.3 **if** $d = \underline{\text{err}} \vee \neg CheckTagEnv(env, type)$
.4 **then** $\underline{\text{err}}$
.5 **else if** $val \in (\|d\| \cup \{\perp\})$
.6 **then** $pev(val)(env)$
.7 **else** $\underline{\text{unmatch}}$
.8 $\mid pev \in EvalPattern(pat) \}$

Annotations to *EvalTypeConstrPattern*:

144. This function gives meaning to type constrained patterns.
.2–.4 If the meaning of the type in the given environment indicates an error or if an undefined composite type was used, an error is reported.
.5–.7 If the given value, *val*, belongs to the invariant set for this domain or is the bottom value, it is given as arguments to a pattern evaluator for *pat*. Otherwise the matching fails.
.8 This is done for all pattern evaluators for the pattern, *pat*.

End of annotations

8.6.1.1 Auxiliary Functions

145. $ChopSetVal : SET_VAL \rightarrow \mathbb{F}(SET_VAL \times SET_VAL)$

145.1 $ChopSetVal(MkTag(\text{'set'}, elem_s)) \triangleq$
.2 $\{ (MkTag(\text{'set'}, el_s_1), MkTag(\text{'set'}, el_s_2)) \mid el_s_1, el_s_2 \in SET_VAL \cdot$
.3 $el_s_1 \cup el_s_2 = elem_s \}$

Annotations to *ChopSetVal*:

145. This function 'chops' a set value into a set of all possible pairs of sets, where each pair can be 'unioned' together to the given argument set.

End of annotations

146. $ChopSeqVal : LIST_VAL \rightarrow \mathbb{F}(LIST_VAL \times LIST_VAL)$

146.1 $ChopSeqVal(MkTag(\text{'seq'}, l)) \triangleq$
.2 $\{ (MkTag(\text{'seq'}, l_1), MkTag(\text{'seq'}, l_2)) \mid l_1, l_2 \in LIST_VAL \cdot \underline{\text{join}}(l_1, l_2) = l \}$

Annotations to *ChopSeqVal*:

146. This function 'chops' a sequence value into a set of all possible pairs of sequences where each pair can be concatenated together to the given argument sequences.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
147.    EvalPatternList : 𝕃(Pattern) → LPatEval
147.1   EvalPatternList(pat_l) ≜
  .2     let lpev_l = [ EvalPattern(pat_l(i)) | i ∈ inds(pat_l) ] in
  .3       let pev_l_s = { l | l ∈ 𝕃(PatEval) · len(l) = len(pat_l) ∧
  .4                                            ∀ i ∈ inds(pat_l) ·
  .5                                              l(i) ∈ lpev_l(i) } in
  .6         { λval.λenv.if val ∈ LIST_VAL
  .7                     then let l = StripSeqTagVal(val) in
  .8                             if len(l) ≠ len(pat_l)
  .9                             then unmatch
  .10                            else if len(pat_l) = 0
  .11                                 then { ↦ }
  .12                                 else let venv_s = { pev_l(i)(l(i))(env)
  .13                                                   | i ∈ inds(pat_l) } in
  .14                                        if { err, unmatch } ∩ venv_s ≠ { }
  .15                                        then if err ∈ venv_s
  .16                                             then err
  .17                                             else unmatch
  .18                                        else if Compatible(venv_s)
  .19                                             then Merge(venv_s)
  .20                                             else unmatch
  .21                     else if val = ⊥
  .22                          then EvalBotPat(elems(pat_l))
  .23                          else unmatch
  .24         | pev_l ∈ pev_l_s }
```

Annotations to *EvalPatternList*:

147. This function gives meaning to a list of patterns.

.3–.5 A set of lists of pattern evaluators is created. Each list of pattern evaluators correspond to the given pattern list, *pat_l*, where an evaluator is given for each element in the list. The list of patterns is supposed to be matched with a list value with the same length, where each pattern matches the corresponding element in the list value.

.6–.24 For each of such list of pattern evaluators a pattern evaluator for the entire list of patterns is created.

.6–.7 If the value, *val*, is a list value, its list of values is extracted and named *l*.

.8–.9 If the length of the list of patterns is different from the length of the list of values, an unmatch is returned.

.10–.11 If there is an empty list of patterns an empty binding is returned, and the matching succeed.

.12–.20 Each of the pattern evaluators are matched with their corresponding list element value, and if none of these matchings fail and the bindings are compatible, the bindings are merged together and returned.

.21–.23 If the given value, *val*, is not a list value, it can either be the bottom value (and in that case all pattern identifiers from the list of patterns will be matched to bottom by *EvalBotPat*) or it can be a normal value, and then the matching fails.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

148. $EvalBotPat : \mathbb{F}(Pattern) \to ENV$

148.1 $EvalBotPat(p_s) \triangleq$
.2 $\underline{\text{Merge}}(\{$ **cases** p :
.3 $MkTag(\text{'PatternId'}, id) \to \{ id \mapsto \bot \},$
.4 $MkTag(\text{'MatchVal'}, _) \to \underline{\text{err}},$
.5 $MkTag(\text{'SeqEnumPattern'}, l) \to EvalBotPat(\underline{\text{elems}}(l)),$
.6 $MkTag(\text{'SeqConcPattern'}, (p_1, p_2)) \to EvalBotPat(\{ p_1, p_2 \}),$
.7 $MkTag(\text{'RecordPattern'}, (_, l)) \to EvalBotPat(\underline{\text{elems}}(l)),$
.8 $MkTag(\text{'TuplePattern'}, l) \to EvalBotPat(\underline{\text{elems}}(l)),$
.9 $MkTag(\text{'SetEnumPattern'}, s) \to EvalBotPat(s),$
.10 $MkTag(\text{'SeqUnionPattern'}, (p_1, p_2)) \to EvalBotPat(\{ p_1, p_2 \})$
.11 $\mid p \in p_s \})$

Annotations to *EvalBotPat*:

148. This function binds all pattern identifiers from a set of patterns, p_s, to the bottom value. Such bindings are in most cases rejected later on, because strict bindings are used in the semantics. However, for mutually recursive definitions it is necessary to have such bindings until the least fixed point has been found.

End of annotations

149. $PropToLPatEval : \mathbb{P}(VAL \to ENV \to \mathbb{P}(ENV)) \to LPatEval$

149.1 $PropToLPatEval(mul_pev_s) \triangleq$
.2 $\{ pev \in PatEval \mid \exists mpev \in mul_pev_s \cdot$
.3 $\forall val \in VAL \cdot$
.4 $\forall env \in ENV \cdot$ **if** $mpev(val)(env) = \{ \}$
.5 **then** $pev(val)(env) = \underline{\text{unmatch}}$
.6 **else** $pev(val)(env) \in mpev(val)(env) \}$

Annotations to *PropToLPatEval*:

149. This function propagates looseness from a set of 'multi' pattern evaluators to the outermost level and achieves a loose pattern evaluator. Notice that if the set of bindings at the innermost level is empty, an unmatch is returned.

End of annotations

150. $NonErrPat : Pattern \times VAL \times ENV \to \mathbb{P}(ENV)$

150.1 $NonErrPat(pat, val, env) \triangleq$
.2 $\{ \underline{\text{overwrite}}(env, venv)$
.3 $\mid venv \in \{ pev(val)(env) \mid pev \in EvalPattern(pat) \} \cdot$
.4 $venv \neq \underline{\text{err}} \wedge venv \neq \underline{\text{unmatch}} \wedge \neg BotEnv(venv) \}$

Annotations to *NonErrPat*:

150. This function takes a syntactic pattern, a value and an environment, and yields the set of environments which extends the given environment with successful pattern matchings of the pattern with the value which contains no bindings to the bottom value.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.6.2 Bindings

```
151.    EvalBind : Bind → LBindEval
151.1   EvalBind(bind) ≙
   .2    cases ShowTag(bind) :
   .3      'SetBind'   → let MkTag('SetBind', (pat, set)) = bind in
   .4                    { λenv.let set_v = ev(env) in
   .5                            if set_v ∈ SET_VAL
   .6                            then let MkTag('set', s) = set_v in
   .7                                   { pev(e)(env) | e ∈ s }\{ unmatch }
   .8                            else { err }
   .9                     | ev ∈ EvalExpr(set), pev ∈ EvalPattern(pat) },
   .10     'TypeBind' → let MkTag('TypeBind', (pat, type)) = bind in
   .11                   { λenv.let d = EvalType(type)(env) in
   .12                           if d = err ∨ ¬CheckTagEnv(env, type)
   .13                           then { err }
   .14                           else { pev(ob)(env) | ob ∈ ||d|| }\{ unmatch }
   .15                    | pev ∈ EvalPattern(pat) }
```

Annotations to *EvalBind*:

151. This function gives meaning to bindings. Bindings are used in quantified expressions and implicit expression constructions (also called comprehensions).

.3–.9 The meaning of set bindings is found: It is a set of binding evaluators which is indexed by expression evaluators of the set expression, and pattern evaluators of the pattern.

.5–.7 If the expression evaluator applied with the given environment returns a set value, elements from this set are matched, one by one, against the pattern evaluator and possibly errors are removed.

.11–.15 The meaning of type bindings is found: It is a set of binding evaluators which is indexed by pattern evaluators of the pattern.

.12–.13 If the meaning of the type in the given environment is an error or an undefined composite type was used, an empty set is returned.

.14 Otherwise each object belonging to the domain is matched against the pattern evaluator and possible errors are removed.

End of annotations

8.7 Auxiliary Functions and Predicates

This clause presents all the auxiliary functions and predicates used in the definition of the dynamic semantics. The functions and predicates are grouped into clauses according to their kind.

8.7.1 Expansion Functions

The preliminary task of the *IsAModelOf* predicate is to expand the given model (environment) with constructs defined implicitly. This expansion is performed by the function *ExpandImplDefs*. *ExpandImplDefs* is defined in terms of two auxiliary functions: *ExpandImplRecTypeDefs* and *ExpandImplStateDef*. In *ExpandImplRecTypeDefs* constructs defined implicitly by record types are added[1)]: This applies to selector, constructor, and modifier-functions. The model must also be enlarged with a function to check the constancy[2)] of the state inside implicit operations. This is done in the function *ExpandImplStateDef*.

[1)] Here it is worth noting that record type definitions can be arbitrarily nested inside any type definition. This is also taken into account in the dynamic semantics. It is also worth noting that all record (or composite) types used in the entire specification must be defined in the type definition part. Therefore, a special 'tag environment' is also expanded, such that this can be checked for all usages of record types.

[2)] This function must ensure that all read state variables are unchanged.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

152. $ExpandImplDefs : \mathbb{E}(TypeDef) \times (StateDef \cup \{\underline{\text{nil}}\}) \to ENV \to ENV \cup \{\underline{\text{err}}\}$

152.1 $ExpandImplDefs(td_m,\ sd)(env) \triangleq$
.2 **let** $impl_ty_env = ExpandImplRecTypeDefs(td_m)(env)$ **in**
.3 **if** $impl_ty_env = \underline{\text{err}}$
.4 **then** $\underline{\text{err}}$
.5 **else** $ExpandImplStateDef(sd)(impl_ty_env)$

Annotations to *ExpandImplDefs*:

152. This function expands the given model, *env*, with constructs which are defined implicitly. Thus, these constructs are not visible outside the given model, but they can be used in the evaluation of the syntactic definition. It is the principle of 'information hiding' which is used here.

.2 Selector, modifier and constructor functions for all composite type definitions are gathered by means of *ExpandImplRecTypeDefs*.

.3–.5 Unless the expansion of implicit functions related to the record definitions fail, the implicit definition from the state definition must be expanded. The model is expanded further with a function which asserts the constancy of the state. This is done by *ExpandImplStateDef*.

End of annotations

153. $ExpandImplRecTypeDefs : \mathbb{E}(TypeDef) \to ENV \to ENV \cup \{\underline{\text{err}}\}$

153.1 $ExpandImplRecTypeDefs(td_m)(env) \triangleq$
.2 **let** $rec_m_s = \{\ ColRecs(SelShape(td_m(id))) \mid id \in \underline{\text{dom}}(td_m)\ \}$ **in**
.3 **if** $\underline{\text{err}} \in rec_m_s \vee \neg Compatible(rec_m_s)$
.4 **then** $\underline{\text{err}}$
.5 **else let** $rec_def_m = \underline{\text{Merge}}(rec_m_s)$ **in**
.6 **let** $rec_d_m = \{\ id \mapsto EvalCompositeType(rec_def_m(id))(env)$
.7 $\mid id \in \underline{\text{dom}}(rec_def_m)\ \}$ **in**
.8 **if** $\underline{\text{err}} \in \underline{\text{rng}}(rec_d_m)$
.9 **then** $\underline{\text{err}}$
.10 **else let** $rec_m = \{\ id \mapsto (rec_def_m(id),\ rec_d_m(id))$
.11 $\mid id \in \underline{\text{dom}}(rec_def_m)\ \}$ **in**
.12 **let** $ty_sel_m = CompFieldSelectors(rec_m)$,
.13 $mk_cons_m = CompRecordConstructors(rec_m)$,
.14 $rec_mod_m = CompRecordModifiers(rec_m)$,
.15 $tagenv = \{\ MkTag(\text{'TagEnvId'}, ()) \mapsto rec_def_m\ \}$ **in**
.16 **if** $ty_sel_m = \underline{\text{err}}$
.17 **then** $\underline{\text{err}}$
.18 **else** $\underline{\text{overwrite}}(env, \underline{\text{Merge}}(\{\ ty_sel_m,\ mk_cons_m,\ rec_mod_m,\ tagenv\ \}))$

Annotations to *ExpandImplRecTypeDefs*:

153. This function expands the given environment, *env*, with implicitly defined constructs in connection with record type definitions. Each record will get one record modifier, one record constructor, and n record selectors (where n is the number of fields where a selector identifier have been provided in the record definition). In addition a local environment is used to check that all composite types which are used outside the type definition part are defined (*tagenv*).

.2 A set of maps, *rec_m_s*, from tag identifier to corresponding record types (also called composite types) is created. Each map is collected by *ColRecs* for each type definition. Notice that this ensures that composite types which are used somewhere inside a type definition are also taken into account.

.3–.4 If either an error is reported from *ColRecs*[1)] or if the maps are not compatible[2)], an error is reported. This ensures that the tags from the different record definitions are uniquely identified.

.6–.7 All the composite types are evaluated and gathered in a map from tag identifier to the corresponding record domain.

[1)] An error will be reported if the same tag identifier is used for different composite types inside the type definition which is given as argument to *ColRecs*.

[2)] If the same tag identifier is mapped to different composite types in different maps.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.12 The field selectors are produced for all the records by means of the auxiliary function called *CompFieldSelectors*. An error is reported if two fields of a record have the same selector name.

.13 The record constructors are produced for all records by means of the auxiliary function called *CompRecordConstructors*.

.14 The record modifiers are produced for all of the records by means of the auxiliary function called *CompRecordModifiers*.

.15 A special 'tag environment' used to ensure that composite types used outside the type definition part are defined in the same (syntactic) way. This special environment is necessary to ensure that selector, constructor and modifier functions exist for all composite types which are used. Thus, all composite types used in an entire VDM-SL specification must be defined somewhere in the type definition part.

.16–.18 Unless an error is reported (due to multiple definitions of a selector name in one of the record definitions) the environment is overwritten with the four different maps.

End of annotations

154. $ExpandImplStateDef : (StateDef \cup \{ \underline{\text{nil}} \}) \to ENV \to ENV \cup \{ \underline{\text{err}} \}$

154.1 $ExpandImplStateDef(sd)(env) \triangleq$

.2 **if** $sd = \underline{\text{nil}}$

.3 **then** env

.4 **else** $\underline{\text{overwrite}}(env, \{ MkTag(\text{'StateConstId'}, ()) \mapsto AssertStateConstancy(env) \})$

Annotations to *ExpandImplStateDef*:

154. This function expands the given environment, *env*, with a function asserting the constancy of the state if a state definition is present.

.2–.3 If no state definition is present no expansion needs to be carried out.

.4 The environment is expanded with a function asserting the state constancy (done by *AssertStateConstancy*).

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.2 Functions for Extending the Environment

```
155.   ExtImplOpEnv : Id × VAL × (Id ∪ {nil}) × (VAL ∪ {nil}) × (VAL ∪ {nil}) ×
                      (VAL ∪ {nil}) × IF(ExtVarInf) → ENV →
                      (ENV ∪ {err} × ENV ∪ {err})
155.1  ExtImplOpEnv(inp_id, inp_v, out_id, out_v, inp_s, out_s, exts)(env) ≜
  .2    let st_id = GetStateTypeId(env),
  .3        inp_m = {inp_id ↦ inp_v},
  .4        out_m = if out_id = nil
  .5                then {↦}
  .6                else {out_id ↦ out_v} in
  .7    if ((inp_id = out_id) ∧ (inp_v ≠ out_v)) ∨
  .8       ((st_id = nostate) ∧ (inp_s ≠ nil ∨ out_s ≠ nil))
  .9    then err
  .10   else if st_id = nostate
  .11        then (overwrite(env, inp_m), overwrite(env, merge(inp_m, out_m)))
  .12        else let sel_m = {MkTag('ValueId', id) ↦ MkTag('RecSelId', (st_id, id))
  .13                        | MkTag('ValueId', id) ∈ elems(ColIdList(exts))},
  .14                 st_v_id = GetStateValId(CUR, st_id),
  .15                 old_st_v_id = GetStateValId(OLD, st_id) in
  .16             if ∃sel_id ∈ rng(sel_m) · sel_id ∉ dom(env)
  .17             then err
  .18             else let pre_st_m = {id ↦ env(sel_m(id))(inp_s) | id ∈ dom(sel_m)} ∪
  .19                                 {st_v_id ↦ inp_s},
  .20                      post_old_st_m = {MkTag('OldId', id) ↦ env(sel_m(id))(inp_s)
  .21                                      | MkTag('ValueId', id) ∈ dom(sel_m)} ∪
  .22                                      {old_st_v_id ↦ inp_s},
  .23                      post_new_st_m = {id ↦ env(sel_m(id))(out_s) | id ∈ dom(sel_m)} in
  .24                  let pre_env = merge(inp_m, pre_st_m),
  .25                      post_env = Merge({inp_m, post_old_m, post_new_m, out_m}) in
  .26                  (overwrite(env, pre_env), overwrite(env, post_env))
```

Annotations to *ExtImplOpEnv*:

155. This function extends the given environment, *env*, such that it can be used in evaluation of the pre-condition and the post-condition of an implicitly defined operation. The environment is extended with bindings of the argument identifier and the result identifier (if any) and the value of the state before and after evaluation of the operation (if any state is present). The first returned environment is to be used for evaluation of the pre-condition, while the second returned environment is to be used for evaluation of the post-condition.

.3–.6 The input and output identifiers are bound to their corresponding values.

.7–.9 If either the input and output identifiers are equal and their corresponding values are different, or there is no state and the input or the output state value is a real value, an error is reported.

.10–.11 If there is no state the two returned environments are simple extensions with the input and output bindings.

.12–.26 If there is a state, selector functions for all state components must be collected. Each of these selector functions must be in the environment and if so they can be used to select all state components. In this way, both the entire state and all its individual state components are made visible.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

156. $ExtExplOpEnv : Id \times VAL \times (VAL \cup \{\underline{\text{nil}}\}) \times (VAL \cup \{\underline{\text{nil}}\}) \times \mathbb{F}(ExtVarInf) \rightarrow ENV \rightarrow (ENV \cup \{\underline{\text{err}}\} \times ENV \cup \{\underline{\text{err}}\})$

156.1 $ExtExplOpEnv(inp_id, inp_v, inp_s, out_s, exts)(env) \triangleq$
.2 **let** $st_id = GetStateTypeId(env)$,
.3 $inp_m = \{ inp_id \mapsto inp_v \}$ **in**
.4 **if** $(st_id = \underline{\text{nostate}})$
.5 **then if** $inp_s \neq \underline{\text{nil}} \vee out_s \neq \underline{\text{nil}}$
.6 **then** $\underline{\text{err}}$
.7 **else** $(\underline{\text{overwrite}}(env, inp_m), \underline{\text{overwrite}}(env, inp_m))$
.8 **else let** $sel_m = \{ MkTag(\text{'ValueId'}, id) \mapsto MkTag(\text{'RecSelId'}, (st_id, id))$
.9 $\mid MkTag(\text{'ValueId'}, id) \in \underline{\text{elems}}(ColIdList(exts)) \}$,
.10 $st_v_id = GetStateValId(\text{CUR}, st_id)$ **in**
.11 **if** $\exists sel_id \in \underline{\text{rng}}(sel_m) \cdot sel_id \notin \underline{\text{dom}}(env)$
.12 **then** $\underline{\text{err}}$
.13 **else let** $st_m = \{ id \mapsto env(sel_m(id))(inp_s) \mid id \in \underline{\text{dom}}(sel_m) \} \cup$
.14 $\{ st_v_id \mapsto inp_s \}$,
.15 $new_st_m = \{ id \mapsto env(sel_m(id))(out_s) \mid id \in \underline{\text{dom}}(sel_m) \} \cup$
.16 $\{ st_v_id \mapsto out_s \}$ **in**
.17 **let** $cur_env = \underline{\text{merge}}(inp_m, st_m)$,
.18 $new_env = \underline{\text{merge}}(inp_m, new_st_m)$ **in**
.19 $(\underline{\text{overwrite}}(env, cur_env), \underline{\text{overwrite}}(env, new_env))$

Annotations to *ExtExplOpEnv*:

156. This function extends the given environment, *env*, such that it can be used in evaluation of the body of an explicitly defined operation. The environment is extended with bindings of the argument identifier and the value of the state before evaluation of the operation (if any state is present). The first returned environment is to be used for evaluation of the statement body, while the second returned environment is a possible returned environment from the evaluation of the body statement.

.4–.7 If there is no state either the input or output state values are 'real' values and an error is reported, or the two returned environments are identical (both extended with a binding of the input identifier).

.8–.19 If there is a state, selector functions for all state components must be collected. Each of these selector functions must be in the environment and if so they can be used to select all state components. In this way, both the entire state and all its individual state components are made visible. Note that each time the body statement modify a component of the state, the entire state is updated as well and it is checked that the state invariant still holds (see *ModifyState* for details about this).

End of annotations

157. $ExtImplFnsEnv : \mathbb{E}(ImplFnDef) \rightarrow ENV \rightarrow (\mathbb{P}(\mathbb{E}(VEnv)) \cup \{\underline{\text{err}}\})$

157.1 $ExtImplFnsEnv(ifd_m)(env) \triangleq$
.2 **let** $id_s = \underline{\text{dom}}(ifd_m)$ **in**
.3 **if** $\exists id \in id_s \cdot ExtImplFnEnv(ifd_m(id))(env) = \underline{\text{err}}$
.4 **then** $\underline{\text{err}}$
.5 **else let** $mul_venv_m = \{ id \mapsto ExtImplFnEnv(ifd_m(id))(env)$
.6 $\mid id \in id_s \}$ **in**
.7 $\{ m \mid m \in \mathbb{E}(VEnv) \cdot \underline{\text{dom}}(m) = id_s \wedge$
.8 $\forall id \in id_s \cdot m(id) \in mul_venv_m(id) \}$

Annotations to *ExtImplFnsEnv*:

157. This function extends the given environment for a collection of implicit function definitions. It returns a set of all possible combinations of maps of extended environments for each function. The environment is extended with a binding from the argument identifier, to a value from the domain of the function. This function is defined in terms of a function which extends the environment for one implicit function definition called *ExtImplFnEnv*.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

158. $ExtImplFnEnv : ImplFnDef \to ENV \to (\mathbb{P}(VEnv) \cup \{\underline{\text{err}}\})$

158.1 $ExtImplFnEnv(MkTag(\text{'}ImplFnDef\text{'}, (h, _, _)))(env) \triangleq$
.2 **let** $MkTag(\text{'}ImplFnHeading\text{'}, (MkTag(\text{'}Par\text{'}, (id, t)), _, _)) = h$ **in**
.3 **let** $d = EvalType(t)(env)$ **in**
.4 **if** $d = \underline{\text{err}} \vee \neg CheckTagEnv(env, t)$
.5 **then** $\underline{\text{err}}$
.6 **else** $\{\{id \mapsto v\} \mid v \in \|d\|\}$

Annotations to *ExtImplFnEnv*:

158. This function extends the given environment for an implicit function definition. It returns a set of maps of all possible combinations of extended environments for that function. The environment is extended with a binding from the argument identifier, to a value from the domain of the function.

End of annotations

159. $ExtImplPolyFnEnv : ImplPolyFnDef \to ENV \to \mathbb{L}(DOM) \to (\mathbb{P}(VEnv) \cup \{\underline{\text{err}}\})$

159.1 $ExtImplPolyFnEnv(MkTag(\text{'}ImplPolyFnDef\text{'}, (tp_l, h, _, _)))(env)(d_l) \triangleq$
.2 **let** $MkTag(\text{'}ImplFnHeading\text{'}, (MkTag(\text{'}par\text{'}, (id, t)), _, _)) = h,$
.3 $env' = ExtDomEnv(tp_l, d_l)$ **in**
.4 **if** $env' = \underline{\text{err}}$
.5 **then** $\underline{\text{err}}$
.6 **else let** $d = EvalType(t)(\underline{\text{overwrite}}(env, env'))$ **in**
.7 **if** $d = \underline{\text{err}} \vee CheckTagEnv(env, t)$
.8 **then** $\underline{\text{err}}$
.9 **else** $\{\{id \mapsto v\} \mid v \in \|d\|\}$

Annotations to *ExtImplPolyFnEnv*:

159. This function extends the given environment for an implicit polymorphic function definition. It returns a set of maps of all possible combinations of extended environments for that function. The environment is extended with a binding from the argument identifier, to a value from the domain of the function.

End of annotations

160. $ExtDomEnv : \mathbb{L}(Id) \times \mathbb{L}(DOM) \to (ENV \cup \{\underline{\text{err}}\})$

160.1 $ExtDomEnv(id_l, d_l) \triangleq$
.2 **if** $(\underline{\text{len}}(id_l) \neq \underline{\text{len}}(d_l)) \vee (\underline{\text{len}}(id_l) \neq \underline{\text{card}}(\underline{\text{elems}}(id_l)))$
.3 **then** $\underline{\text{err}}$
.4 **else if** $id_l = [\,]$
.5 **then** $\{\mapsto\}$
.6 **else** $\{id_l(i) \mapsto d_l(i) \mid i \in \underline{\text{inds}}(id_l)\}$

Annotations to *ExtDomEnv*:

160. This function creates an environment extension binding the identifiers from *id_l* to a domain from *d_l* with the same index.

.2–.3 An error is reported if either the list of identifiers and the list of domains have different lengths or if any of the identifiers are duplicated.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
161.    ExtValEnv : 𝕃(Id) × 𝕃(VAL) → (ENV ∪ { err })
161.1   ExtValEnv(id_l, v_l) ≜
  .2     if (len(id_l) ≠ len(v_l)) ∨ (len(id_l) ≠ card(elems(id_l)))
  .3     then err
  .4     else if id_l = [ ]
  .5          then { ↦ }
  .6          else { id_l(i) ↦ v_l(i) | i ∈ inds(id_l) }
```

Annotations to *ExtValEnv*:

161. This function creates an environment extension binding the identifiers from *id_l* to a value from *v_l* with the same index.

.2–.3 An error is reported if either the list of identifiers and the list of domains have different lengths or if any of the identifiers are duplicated.

End of annotations

```
162.    ExtLocDomEnv : Id × DOM → ENV → ENV
162.1   ExtLocDomEnv(id, dom)(env) ≜
  .2     let teid = MkTag('TypeEnvId', ()),
  .3         map = { id ↦ dom } in
  .4     if teid ∈ dom(env)
  .5     then let te = overwrite(env(teid), map) in
  .6               overwrite(env, { teid ↦ te })
  .7     else overwrite(env, map)
```

Annotations to *ExtLocDomEnv*:

162. This function extends the given environment, *env*, with a binding from *id* to *dom* inside the local domain environment (which is used for type checking localized assignable variables).

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.3 Functions and Predicates to deal with the State

163. $AssertStateConstancy : ENV \to (\mathbb{F}(ExtVarInf) \to (VAL \cup \{\underline{\text{nil}}\} \times VAL \cup \{\underline{\text{nil}}\})) \to \mathbb{B}$

```
163.1  AssertStateConstancy(env) ≙
  .2    let st_id = GetStateTypeId(env),
  .3        st_d = GetStateDom(env) in
  .4      if st_id = nostate
  .5      then λexts ∈ IF(ExtVarInf) .
  .6             λ(inp_st, out_st) ∈ (VAL ∪ {nil} × VAL ∪ {nil}).
  .7               if exts = { }
  .8               then True()
  .9               else False()
  .10     else λexts ∈ IF(ExtVarInf) .
  .11              let sel_fn_s = { MkTag('RecSelId', (tag, selid))
  .12                             | MkTag('RecSelId', (tag, selid)) ∈ dom(env) · tag = StripTag(st_id) },
  .13                  ext_wr_ids = { MkTag('RecSelId', (StripTag(st_id), SelExtVarInfId(ext)))
  .14                               | ext ∈ exts · READWRITE = SelExtVarInfMode(ext) } in
  .15                let const_ids = sel_fn_s\ext_wr_ids in
  .16                  λ(inp_st, out_st) ∈ (VAL ∪ {nil} × VAL ∪ {nil}).
  .17                    if (inp_st ∈ ||st_d||) ∧ (out_st ∈ ||st_d||) ∧
  .18                       ∀id ∈ sel_fn_s · id ∈ dom(env) ∧ env(id) ∈ RECORD_VAL → VAL ∧
  .19                             {inp_st, out_st} ⊆ δ0(env(id))
  .20                    then let b = ∀ cid ∈ const_ids · env(cid)(inp_st) = env(cid)(out_st) in
  .21                           MkTag('bool', b)
  .22                    else False()
```

Annotations to *AssertStateConstancy*:

163. This function creates a Curried function that checks whether or not all state components, which are not defined READWRITE in the external clause of an operation definition, are unchanged. This returned Curried function first takes a set of external variable information about some state components (only those which are marked with READWRITE are allowed to be changed) and then as a Curried parameter the value of the total state before and after the evaluation of the operation, and yields a Boolean result. The type $\mathbb{F}(ExtVarInf) \to (VAL \cup \{\underline{\text{nil}}\} \times VAL \cup \{\underline{\text{nil}}\}) \to \mathbb{B}$ is also called *STATE_FUN* in the definition of the extended semantic domains.

End of annotations

164. $CheckStateConstancy : ENV \to \mathbb{F}(ExtVarInf) \to (VAL \cup \{\underline{\text{nil}}\} \times VAL \cup \{\underline{\text{nil}}\}) \to \mathbb{B}$

```
164.1  CheckStateConstancy(env)(exts)(inp_st, out_st) ≙
  .2    let st_id = GetStateTypeId(env),
  .3        st_d = GetStateDom(env) in
  .4      let const_pred_id = MkTag('StateConstId', ()) in
  .5        if const_pred_id ∈ dom(env) ∧
  .6           env(const_pred_id) ∈ (IF(ExtVarInf) → (VAL ∪ {nil} × VAL ∪ {nil}) → IB)
  .7        then let const_pred = env(const_pred_id) in
  .8               if (inp_st ∈ ||st_d||) ∧ (out_st ∈ ||st_d||)
  .9               then const_pred(exts)(inp_st, out_st)
  .10              else False()
  .11       else False()
```

Annotations to *CheckStateConstancy*:

164. This function checks the constancy of the state, i.e. that only the READWRITE variables of an operation are changed. For this purpose it uses the Curried function created by *AssertStateConstancy*. This function is stored in the environment, *env*, with the name *MkTag('StateConstId', ())* and the call of this function

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

can be seen in line 9 of the definition above.
This Curried has the type $\mathbb{F}(ExtVarInf) \to (VAL \cup \{\underline{\text{nil}}\} \times VAL \cup \{\underline{\text{nil}}\}) \to \mathbb{B}$ which also is called *STATE_FUN* in the definition of the extended semantic domains.

End of annotations

165. $CheckTagEnv : \mathbf{Pred}(ENV \times (Type \cup \{\underline{\text{nil}}\}))$

165.1 $CheckTagEnv(env, type) \triangleq$
.2 **let** $tenvid = MkTag('TagEnvId', ())$ **in**
.3 **if** $tenvid \in \underline{\text{dom}}(ENV) \wedge env(tenvid) \in \mathbb{E}(CompositeType)$
.4 **then if** $type = \underline{\text{nil}}$
.5 **then** T
.6 **else let** $def_m = env(tenvid)$,
.7 $com_tp_m = ColRecs(type)$ **in**
.8 $\underline{\text{dom}}(com_tp_m) \subseteq \underline{\text{dom}}(def_m) \wedge Compatible(\{com_tp_m, def_m\})$
.9 **else** F

Annotations to *CheckTagEnv*:

165. Checks that an environment contains an entry for a localized environment, which contains information about the tags for all of the record definitions in a specification which must be 'compatible'. Each occurrence of any tag in a type definition must occur in identical type definitions. Otherwise, it would be possible to create different kinds of values with the same tag, and thereby it would be impossible to find out which selector function to use for such a value.

End of annotations

166. $Restore : ENV \times \mathbb{F}(Id) \times (ENV \times MODE \times (VAL \cup \{\underline{\text{nil}}\})) \to (ENV \times MODE \times (VAL \cup \{\underline{\text{nil}}\}))$

166.1 $Restore(oldenv, ids, (env, mode, val)) \triangleq$
.2 $(\underline{\text{overwrite}}(env, \underline{\text{restrict}}(oldenv, ids)), mode, val)$

Annotations to *Restore*:

166. This function restores the old values of variables on exit from a block-structure. This is necessary to cope with the scope rules of statements, which may introduce local identifiers which hide the identifiers outside with the same names. However, since the statement may have made other changes to the environment, it is necessary to restore the old values of the identifiers outside.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
167.   ResLocId : ENV × Id × (ENV × MODE × (VAL ∪ {nil}))
                  → (ENV × MODE × (VAL ∪ {nil}))
167.1  ResLocId(oldenv, id, (env, mode, val)) ≙
  .2    let teid = MkTag('TypeEnvId', ()) in
  .3      if teid ∉ dom(env) ∨ env(teid) ∉ LOC_ENV ∨ id ∉ dom(env(teid))
  .4      then (env, ret, ⊥)
  .5      else if teid ∉ dom(oldenv) ∨ oldenv(teid) ∉ LOC_ENV
  .6           then if id ∉ dom(oldenv)
  .7                then (restrict(env, {id, teid}), mode, val)
  .8                else let oldmap = {id ↦ oldenv(id)} in
  .9                     (overwrite(restrict(env, {teid}), oldmap), mode, val)
  .10          else if id ∉ dom(oldenv(teid))
  .11               then let te = subtract(env(teid), {id}) in
  .12                    if id ∉ dom(oldenv)
  .13                    then (overwrite(restrict(env, {id}), {teid ↦ te}), mode, val)
  .14                    else let oldmap = merge({id ↦ oldenv(id)}, {teid ↦ te}) in
  .15                         (overwrite(env, oldmap), mode, val)
  .16               else let te = overwrite(env(teid), {id ↦ oldenv(teid)(id)}) in
  .17                    if id ∉ dom(oldenv)
  .18                    then (overwrite(restrict(env, {id}), {teid ↦ te}), mode, val)
  .19                    else let oldmap = merge({teid ↦ te}, {id ↦ oldenv(id)}) in
  .20                         (overwrite(env, oldmap), mode, val)
```

Annotations to *ResLocId*:

167. This function restores the old value of an assignable variable on exit from a block statement. This is necessary to cope with the scope rules of block statements, which may introduce localized (assignable) identifiers which hide the identifiers outside with the same names. However, since the block statement may have made other changes to the environment, it is necessary to restore the old values of the identifier outside (if any).

.5–.9 If there does not exist a type environment in the old environment then *teid* must be removed from the new environment. If *id* existed in the old environment, the old value must also be restored.

.10–.15 If *id* was not in the old type environment is must be removed from the new type environment. If *id* existed in the old environment, the old value must also be restored.

.16–.20 If *id* was in the old type environment its old type definition must be restored. If *id* also existed in the old environment, the old value must also be restored.

End of annotations

8.7.4 Functions and Predicates to deal with Curried Functions

```
168.   EqCurDoms : Pred(ℕ × VAL × VAL)
168.1  EqCurDoms(n, v1, v2) ≙
  .2    if n = 0
  .3    then T
  .4    else if IsCurFunVal(n, v1) ∧ IsCurFunVal(n, v2)
  .5         then let MkTag('fun', f1) = v1,
  .6                  MkTag('fun', f2) = v2 in
  .7              if δ0(f1) = δ0(f2)
  .8              then ∀e ∈ δ0(f1) · EqCurDoms(n − 1, f1(e), f2(e))
  .9              else F
  .10        else F
```

Annotations to *EqCurDoms*:

168. This predicate checks whether the two values, v_1 and v_2, have equal domains in n Curried levels.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.2–.3 If n is zero, it is trivially fulfilled.
.4–.10 If both values are Curried in n levels, it is checked that they have equal domains.
.8 Here it would be possible to permit the criteria to be $\delta_0(f_1) \subseteq \delta_0(f_2)$. This would simply allow more functions to be acceptable as the denotation of a syntactic function definition. This would correspond to permitting functions with a larger domain to be permitted as the denotation.

End of annotations

169. $IsCurFunVal$: **Pred**($\mathbb{N} \times VAL$)

169.1 $IsCurFunVal(n, val) \triangleq$
.2 **if** $n = 0$
.3 **then** T
.4 **else if** $v \in FUN_VAL$
.5 **then let** $MkTag('fun', f) = val$ **in**
.6 $\forall e \in \delta_0(f) \cdot IsCurFunVal(n - 1, f(e))$
.7 **else** F

Annotations to $IsCurFunVal$:
169. This predicate checks that a value, val, is Curried in n levels.

End of annotations

170. $ApplyCurFn : VAL \times \mathbb{L}(VAL) \to VAL$

170.1 $ApplyCurFn(v, v_l) \triangleq$
.2 **if** $v_l = [\,]$
.3 **then** v
.4 **else if** $v \in FUN_VAL$
.5 **then let** $MkTag('fun', f) = v,$
.6 $arg = \underline{\text{hd}}(v_l)$ **in**
.7 **if** $arg \in \delta_0(f)$
.8 **then** $ApplyCurFn(f(arg), \underline{\text{tl}}(v_l))$
.9 **else** $\perp$
.10 **else** $\perp$

Annotations to $ApplyCurFn$:
170. This function applies a list of values to a value (which is supposed to be a (Curried) function) with a Currying level corresponding to the length of the value list.

End of annotations

8.7.5 Compute Functions

171. $CompFieldSelectors : \mathbb{E}(CompositeType \times RECORD_DOM) \to$
$\mathbb{E}(RECORD_VAL \to VAL) \cup \{\underline{\text{err}}\}$

171.1 $CompFieldSelectors(rec_m) \triangleq$
.2 **if** $\exists id \in \underline{\text{dom}}(rec_m) \cdot CompRecordSel(rec_m(id)) = \underline{\text{err}}$
.3 **then** $\underline{\text{err}}$
.4 **else** $\underline{\text{Merge}}(\{\, CompRecordSel(rec_m(id)) \mid id \in \underline{\text{dom}}(rec_m) \,\})$

Annotations to $CompFieldSelectors$:
171. This function takes a map of pairs consisting of a syntactic composite type and a corresponding record domain and yields a map of field selector functions for all of these records. Note that the type $RECORD_VAL \to VAL$ is also called SEL_FUN in the definition of the extended semantic domains.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

End of annotations

172. $CompRecordSel : CompositeType \times RECORD_DOM \to \mathbb{E}(RECORD_VAL \to VAL) \cup \{\underline{\text{err}}\}$

172.1 $CompRecordSel(MkTag(\text{'}CompositeType\text{'}, (id, f_l)), den) \triangleq$
.2 **let** $sel_m = ColSelMap(f_l)$ **in**
.3 **if** $sel_m = \underline{\text{err}}$
.4 **then** $\underline{\text{err}}$
.5 **else** $\{ GetRecSelId(id, sel) \mapsto GenSelectorFn(den, sel_m, sel) \mid sel \in \underline{\text{dom}}(sel_m) \}$

Annotations to *CompRecordSel*:
172. This function takes a syntactic composite type and a corresponding record domain, and yields a map of field selector functions for this record. Note that the type $RECORD_VAL \to VAL$ is also called *SEL_FUN* in the definition of the extended semantic domains.

End of annotations

173. $CompRecordConstructors : \mathbb{E}(CompositeType \times RECORD_DOM) \to \mathbb{E}(\mathbb{L}(VAL) \to RECORD_VAL)$

173.1 $CompRecordConstructors(rec_m) \triangleq$
.2 $\underline{\text{Merge}}(\{ GetRecConsId(id) \mapsto GenConstructorFn(rec_m(id)) \mid id \in \underline{\text{dom}}(rec_m) \})$

Annotations to *CompRecordConstructors*:
173. This function takes a map of pairs, consisting of a syntactic composite type and a corresponding record domain, and yields a map of record constructor functions for all of these records. Note that the type $\mathbb{L}(VAL) \to RECORD_VAL$ is also called *CONS_FUN* in the definition of the extended semantic domains.

End of annotations

174. $CompRecordModifiers : \mathbb{E}(CompositeType \times RECORD_DOM) \to \mathbb{E}(\mathbb{E}(VAL) \to RECORD_VAL \to VAL)$

174.1 $CompRecordModifiers(rec_m) \triangleq$
.2 $\{$ **let** $mid = GetRecModId(id),$
.3 $(t, d) = rec_m(id)$ **in**
.4 $mid \mapsto GenRecordModFn(id, CompRecordSel(t, d), d)$
.5 $\mid id \in \underline{\text{dom}}(rec_m) \}$

Annotations to *CompRecordModifiers*:
174. This function takes a map of pairs, consisting of a syntactic composite type and a corresponding record domain, and yields a map of modifier functions for all of these records. Note that the type $\mathbb{E}(VAL) \to RECORD_VAL \to VAL$ is also called *MOD_FUN* in the definition of the extended semantic domains.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.6 Generate Functions

175. $GenRecordModFn : Id \times \mathbb{E}(RECORD_VAL \to VAL) \times RECORD_DOM \to (\mathbb{E}(VAL) \to RECORD_VAL \to VAL)$

175.1 $GenRecordModFn(tag,\ sel_m,\ den) \triangleq$
.2 $\lambda\ mod_m \in \mathbb{E}(VAL)\ .$
.3 **let** $mod_m' = \{\ GetRecSelId(tag,\ mod_id) \mapsto mod_m(mod_id)$
.4 $\mid\ mod_id \in \underline{\mathrm{dom}}(mod_m)\ \}$ **in**
.5 **if** $\underline{\mathrm{dom}}(mod_m') \subseteq \underline{\mathrm{dom}}(sel_m)$
.6 **then let** $mod_ids = \underline{\mathrm{dom}}(mod_m'),$
.7 $const_ids = \underline{\mathrm{dom}}(sel_m) \backslash \underline{\mathrm{dom}}(mod_m')$ **in**
.8 $\lambda\ val \in RECORD_VAL\ .$
.9 **if** $val \in \|den\|$
.10 **then** $Iota(\{\ val' \mid val' \in \|den\| \cdot \forall id \in mod_ids \cdot$
.11 $sel_m(id)(val') = mod_m'(id) \wedge$
.12 $\forall id \in const_ids \cdot$
.13 $sel_m(id)(val) = sel_m(id)(val')\ \})$
.14 **else** $\bot$
.15 **else** $\lambda\ val \in RECORD_VAL\ .\ \bot$

Annotations to *GenRecordModFn*:

175. This function generates a record modifier function. It takes a *tag* of the record, a map of the record selectors for the record, and the record domain which the record value must belong to. It returns a record modifier function which takes a map from field selectors to modified values and yields a function from a record value to a modified value[1)].

.5 The selector functions used for modification must be included in the collection of all of the selector functions for the record.

.9–.13 If the argument record value belongs to the given record domain, *den*, the new record value is constructed defining which fields to modify and which fields keep the same value.

End of annotations

176. $GenSelectorFn : RECORD_DOM \times \mathbb{E}(\mathbb{N}) \times Id \to (RECORD_VAL \to VAL)$

176.1 $GenSelectorFn(den,\ sel_m,\ sel) \triangleq$
.2 $\lambda val \in RECORD_VAL.$**if** $val \in \|den\| \wedge sel \in \underline{\mathrm{dom}}(sel_m)$
.3 **then let** $val_l = StripRecTagVal(val)$ **in**
.4 **if** $sel_m(sel) \in \underline{\mathrm{inds}}(val_l)$
.5 **then** $val_l(sel_m(sel))$
.6 **else** $\bot$
.7 **else** $\bot$

Annotations to *GenSelectorFn*:

176. This function generates a record selector function. It takes a record domain, a map from selector identifiers to their number (index) in the record, and a record selector identifier for which a selector function must be generated. The returned domain of record selector functions is also called *SEL_FUN* in the definition of the extended semantic domains.

End of annotations

[1)] This domain is also called *MOD_FUN* in the definition of the extended semantic domains.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

177. $GenConstructorFn : CompositeType \times RECORD_DOM \rightarrow (\mathbb{L}(VAL) \rightarrow RECORD_VAL_{\perp})$

177.1 $GenConstructorFn(type,\ dom) \triangleq$
.2 **let** $v_l_s = \{\ StripRecTagVal(rec) \mid rec \in \|dom\|\ \},$
.3 $MkTag(\text{'CompositeType'},\ (tag,\ _)) = type$ **in**
.4 $\lambda v_l \in \mathbb{L}(VAL)$.**if** $v_l \in v_l_s$
.5 **then** $MkTag(\text{'record'},\ MkTag(tag,\ v_l))$
.6 **else** $\perp$

Annotations to *GenConstructorFn*:

177. This function generates a record constructor function. It takes a syntactic composite type and a record domain and yields a record constructor function. The returned domain of record constructor functions is also called *CONS_FUN* in the definition of the extended semantic domains.

End of annotations

178. $GenPolyApply : POLYVAL \times \mathbb{L}(DOM) \times VAL \rightarrow VAL$

178.1 $GenPolyApply(poly,\ d_l,\ val) \triangleq$
.2 **let** $f = poly(d_l)$ **in**
.3 **if** $f \in FUN_VAL$
.4 **then let** $MkTag(\text{'fun'},\ fn) = f$ **in**
.5 **if** $val \in \delta_0(fn)$
.6 **then** $fn(val)$
.7 **else** $\perp$
.8 **else** $\perp$

Annotations to *GenPolyApply*:

178. This function generates an instantiated version of a polymorphic function, by first instantiating the polymorphic function and then applying the resulting monomorphic function with the given argument value, *val*.

End of annotations

8.7.7 *Make* Functions

179. $MakePolyFnDef : \mathbb{L}_1(TypeVar) \times Lambda \rightarrow LPEval$

179.1 $MakePolyFnDef(t_l,\ body) \triangleq$
.2 $\{\ \lambda env\ .\ \lambda d_l\ .$ **if** $(\underline{\text{len}}(d_l) \neq \underline{\text{len}}(t_l)) \vee (\underline{\text{len}}(t_l) \neq \underline{\text{card}}(\underline{\text{elems}}(t_l)))$
.3 **then** $\perp$
.4 **else** $ev(\underline{\text{overwrite}}(env,\ ExtDomEnv(t_l,\ d_l)))$
.5 $\mid ev \in EvalExpr(body)\ \}$

Annotations to *MakePolyFnDef*:

179. This function makes a loose polymorphic evaluator from a list of type variables and a syntactic lambda expression.

.2–.3 If the length of the list of type variables is different from the length of the domain list, *d_l*, or if any type variable occur more than once, bottom is returned, indicating an error.

.4 A body evaluator is applied with an environment which has been extended with the actual type instantiation.

.5 This is done for all possible expression evaluators for the *body*.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
180.   MakeFuncAbs : Id × Type × Type × Expr → ENV → ℙ(VAL)
180.1  MakeFuncAbs(id, domtp, rngtp, expr)(env) ≜
  .2    let d1 = EvalType(domtp)(env),
  .3        d2 = EvalType(rngtp)(env),
  .4        lev = EvalExpr(expr) in
  .5     if d1 = err ∨ d2 = err
  .6     then { ⊥ }
  .7     else let fn_s = { λpar ∈ ‖d1‖. ev(overwrite(env, { id ↦ par }))
  .8                     | ev ∈ lev } in
  .9          { if ∀a ∈ ‖d1‖ · fn(a) ∈ ‖d2‖
  .10           then MkTag('fun', fn)
  .11           else ⊥
  .12         | fn ∈ fn_s }
```

Annotations to *MakeFuncAbs*:

180. This function makes function abstraction. It takes an argument identifier, a syntactic domain type, a syntactic range type, and a syntactic (body) expression. Furthermore it takes (as a Curried argument) an environment. It returns a set of (function) values corresponding to looseness of the body expression.

.5–.6 If the meaning of either the domain type or the range type is erroneous, an error is reported (using bottom).

.7–.12 Otherwise for all possible functional abstractions of the given body expression, it is checked that all domain elements return a proper range element when the function is applied. In this case, the function is tagged and returned; otherwise bottom is returned.

End of annotations

```
181.   MakeEvalMapSet : 𝔼(ImplFnDef) ∪ 𝔼(ImplPolyFnDef) →
                        (Id × (ImplFnDef ∪ ImplPolyDef) → Expr) →
                        ℙ(𝔼(EEval))
181.1  MakeEvalMapSet(ifd_m)(f) ≜
  .2    let id_s = dom(ifd_m) in
  .3     let lev_m = { id ↦ EvalExpr(f(id, ifd_m(id))) | id ∈ id_s } in
  .4      { m | m ∈ 𝔼(EEval) · dom(m) = id_s ∧ ∀id ∈ id_s · m(id) ∈ lev_m(id) }
```

Annotations to *MakeEvalMapSet*:

181. This auxiliary function makes a set of collections of expression evaluators. It takes a collection of implicit functions and a function as a Curried argument, which given an identifier and an implicit function definition selects an expression from the definition. The looseness of these selected expressions is then propagated such that a set of all possible mappings from the function identifiers to a corresponding expression evaluator is obtained.

End of annotations

```
182.   MakeFns : 𝔼(ImplFnDef) → ENV → (ℙ(𝔼(VAL)) ∪ { err })
182.1  MakeFns(implfndef_m)(env) ≜
  .2    let id_s = dom(implfndef_m) in
  .3     let tp_m = { id ↦ GetType(implfndef_m(id)) | id ∈ id_s } in
  .4      let d_m = EvalTypeMap(tp_m)(env) in
  .5       if d_m = err ∨ ∃tp ∈ rng(tp_m) · ¬CheckTagEnv(env, tp)
  .6       then err
  .7       else { m | m ∈ 𝔼(VAL) · dom(m) = id_s ∧
  .8                          ∀id ∈ id_s · m(id) ∈ ‖d_m(id)‖ ∧
  .9                                       IsFunCont(m(id), d_m(id)) }
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annotations to *MakeFns*:

182. This auxiliary function makes a set of collections of function values, or reports an error. It takes a collection of implicit function definitions and as a Curried parameter an environment. Given that the function types for all of the implicit definitions in the given environment (and all composite types which are used, are also defined in the type definition part of the specification), all collections of possible function values belonging to the respective function types are collected. Finally, it is guaranteed that the functions are continuous.

End of annotations

183. $MakePolyFns : \mathbb{IE}(polyImplFnDef) \to ENV \to (\mathbb{IP}(\mathbb{IE}(POLYVAL)) \cup \{\underline{\text{err}}\})$

183.1 $MakePolyFns(implfndef_m)(env) \triangleq$
.2 **let** $id_s = \underline{\text{dom}}(implfndef_m)$ **in**
.3 **let** $tp_m = \{ id \mapsto GetType(implfndef_m(id)) \mid id \in id_s \}$,
.4 $var_m = \{ id \mapsto ColTypeVars(implfndef_m(id)) \mid id \in id_s \}$ **in**
.5 **let** $d_m = \{ id \mapsto EvalPolyType(var_m(id), tp_m(id))(env) \mid id \in id_s \}$ **in**
.6 **if** $\exists tp \in \underline{\text{rng}}(tp_m) \cdot \neg CheckTagEnv(env, tp)$
.7 **then** $\underline{\text{err}}$
.8 **else** $\{ m \mid m \in \mathbb{IE}(POLYVAL) \cdot \underline{\text{dom}}(m) = id_s \wedge$
.9 $\forall id \in id_s \cdot \forall d_l \in \delta_0(d_m(id)) \cdot m(id)(d_l) \in \|d_m(id)(d_l)\| \wedge$
.10 $IsFunCont(m(id)(d_l), d_m(id)(d_l)) \}$

Annotations to *MakePolyFns*:

183. This auxiliary function makes a set of collections of polymorphic function values, or reports an error. It takes a collection of implicit polymorphic function definitions and as a Curried parameter an environment. Given that the function types for all of the implicit definitions have been given meaning in the given environment (and all composite types which are used are also defined in the type definition part of the specification), all collections of possible function values belonging to the respective polymorphic function types are collected. Finally, it is guaranteed that the functions are continuous.

End of annotations

8.7.8 *Get* Functions

184. $GetFnApp : Id \times ImplFnDef \to Expr$

184.1 $GetFnApp(fnid, MkTag(\text{'ImplFnDef'}, (heading, _, _))) \triangleq$
.2 **let** $MkTag(\text{'ImplFnHeading'}, (MkTag(\text{'Par'}, (argid, _)), _, _)) = heading$ **in**
.3 $MkTag(\text{'Apply'}, (fnid, argid))$

Annotations to *GetFnApp*:

184. This auxiliary function takes an identifier and a syntactic implicitly defined function and creates a syntactic apply expression where the given function is applied with the formal parameter.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
185.   GetType : ImplFnDef ∪ ImplPolyFnDef ∪ ExplFnDef ∪ ExplPolyFnDef → Type
185.1  GetType(def) ≜
  .2    cases ShowTag(def):
  .3      'ImplFnDef'      → let MkTag('ImplFnDef', (h, _, _)) = def in
  .4                             let MkTag('ImplFnHeading', (par, r_t, _)) = h in
  .5                               let MkTag('Par', (_, d_t)) in
  .6                                 MkTag('FnType', (d_t, r_t)),
  .7      'ImplPolyFnDef' → let MkTag('ImplPolyFnDef', (_, h, _, _)) = def in
  .8                             let MkTag('ImplFnHeading', (par, r_t, _)) = h in
  .9                               let MkTag('Par', (_, d_t)) in
  .10                                MkTag('FnType', (d_t, r_t)),
  .11     'ExplFnDef'      → let MkTag('ExplFnDef', (d_t, r_t, _, _, _)) = def in
  .12                            MkTag('FnType', (d_t, r_t)),
  .13     'ExplPolyFnDef' → let MkTag('ExplPolyFnDef', (_, tp, _, _, _)) = def in
  .14                            tp
```

Annotations to *GetType*:

185. This auxiliary function takes a syntactic function definition and yields the syntactic type of that function.

End of annotations

```
186.   GetExpr : Pattern → Expr
186.1  GetExpr(pat) ≜
  .2    cases pat :
  .3      MkTag('PatternId', id)                → id,
  .4      MkTag('MatchVal', expr)               → expr,
  .5      MkTag('SeqEnumPattern', p_l)          → let e_l = ApplySeq (GetExpr)(p_l) in
  .6                                                MkTag('SeqEnumeration', e_l),
  .7      MkTag('SeqConcPattern', (p1, p2))     → let e1 = GetExpr(p1),
  .8                                                e2 = GetExpr(p2) in
  .9                                              let e = (e1, SEQCONC, e2) in
  .10                                               MkTag('BinaryExpr', e),
  .11     MkTag('RecordPattern', (id, p_l))     → let e_l = ApplySeq (GetExpr)(p_l) in
  .12                                             let e = (id, e_l) in
  .13                                               MkTag('RecordConstructor', e),
  .14     MkTag('TuplePattern', p_l)            → let e_l = ApplySeq (GetExpr)(p_l) in
  .15                                               MkTag('TupleConstructor', e_l),
  .16     MkTag('SetUnionPattern', (p1, p2))    → let e1 = GetExpr(p1),
  .17                                                e2 = GetExpr(p2) in
  .18                                             let e = (e1, SETUNION, e2) in
  .19                                               MkTag('BinaryExpr', e),
  .20     MkTag('SetEnumPattern', p_s)          → let e_s = { GetExpr(p) | p ∈ p_s } in
  .21                                               MkTag('SetEnumeration', e_s),
  .22     MkTag('SetConstrPattern', (p, _))     → GetExpr(p),
  .23     MkTag('TypeConstrPattern', (p, _))    → GetExpr(p)
```

Annotations to *GetExpr*:

186. This auxiliary function takes a syntactic pattern and yields the corresponding syntactic expression.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

187. $GetStateDom : ENV \rightarrow DOM$

187.1 $GetStateDom(env) \triangleq$
.2 **let** $st_id = GetStateTypeId(env)$ **in**
.3 **if** $st_id = \underline{\text{nostate}}$
.4 **then** $(\emptyset_\perp, \{\ \})$
.5 **else if** $env(st_id) \in DOM$
.6 **then** $env(st_id)$
.7 **else** $(\emptyset_\perp, \{\ \})$

Annotations to *GetStateDom*:
187. This auxiliary function obtain the state domain if present. Otherwise an empty domain is returned.
End of annotations

188. $GetStateTypeId : ENV \rightarrow StateTypeId \cup \{ \underline{\text{nostate}} \}$

188.1 $GetStateTypeId(env) \triangleq$
.2 **if** $\exists! id \in \underline{\text{dom}}(env) \cdot ShowTag(id) = \text{'StateTypeId'} \wedge env(id) \in DOM$
.3 **then** $Iota(\{ id \mid id \in \underline{\text{dom}}(env) \cdot ShowTag(id) = \text{'StateTypeId'} \})$
.4 **else** $\underline{\text{nostate}}$

Annotations to *GetStateTypeId*:
188. This auxiliary function obtain the state type identifier if it is present. Otherwise nostate is returned to indicate that the specification does not contain a state definition.
End of annotations

189. $GetStateValId : \{ \text{CUR}, \text{OLD} \} \times StateTypeId \rightarrow ValueId \cup OldId$

189.1 $GetStateValId(mode, MkTag(\text{'StateTypeId'}, st_id)) \triangleq$
.2 **if** $mode = \text{CUR}$
.3 **then** $MkTag(\text{'ValueId'}, st_id)$
.4 **else** $MkTag(\text{'OldId'}, st_id)$

Annotations to *GetStateValId*:
189. This auxiliary function obtains state value identifier given the mode which is desired and the state type identifier.
End of annotations

190. $GetRecSelId : Id \times Id \rightarrow RecSelId$

190.1 $GetRecSelId(tag, field) \triangleq$
.2 $MkTag(\text{'RecSelId'}, (StripTag(tag), StripTag(field)))$

Annotations to *GetRecSelId*:
190. This auxiliary function obtains the identifier used to denote a record selector function. It takes a tag identifier and a field selector identifier and combines the two corresponding tokens into a new unique identifier.
End of annotations

191. $GetRecModId : ValueId \rightarrow RecModId$

191.1 $GetRecModId(MkTag(\text{'ValueId'}, tag)) \triangleq$
.2 $MkTag(\text{'RecModId'}, tag)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annotations to *GetRecModId*:

191. This auxiliary function obtains the identifier used to denote a record modifier function. It takes a tag identifier and use its token to produce a new unique identifier.

End of annotations

192. $GetRecConsId : ValueId \rightarrow RecConsId$

192.1 $GetRecConsId(MkTag(\text{'}ValueId\text{'}, tag)) \triangleq$

.2 $MkTag(\text{'}RecConsId\text{'}, tag)$

Annotations to *GetRecConsId*:

192. This auxiliary function obtains the identifier used to denote a record constructor function. It takes a tag identifier and use its token to produce a new unique identifier.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.9 Collector Functions

193. $ColRecs : Type \to (\mathbb{E}(CompositeType) \cup \{ \underline{\text{err}} \})$

193.1 $ColRecs(type) \triangleq$
.2 **cases** *type* :
.3 $MkTag(\text{'CompositeType'}, (id, f_l)) \to$ **let** $ty_s = \underline{\text{elems}}(ColTypeList(f_l))$ **in**
.4 **let** $r_s = \{ ColRecs(t) \mid t \in ty_s \}$ **in**
.5 **if** $\underline{\text{err}} \notin r_s \wedge$
.6 $Compatible(r_s \cup \{ \{ id \mapsto type \} \})$
.7 **then** $\underline{\text{Merge}}(r_s \cup \{ \{ id \mapsto type \} \})$
.8 **else** $\underline{\text{err}}$,
.9 $MkTag(\text{'ProductType'}, t_l) \to$ **let** $r_s = \{ ColRecs(t) \mid t \in \underline{\text{elems}}(t_l) \}$ **in**
.10 **if** $\underline{\text{err}} \notin r_s \wedge Compatible(r_s)$
.11 **then** $\underline{\text{Merge}}(r_s)$
.12 **else** $\underline{\text{err}}$
.13 $MkTag(\text{'UnionType'}, t_s) \to$ **let** $r_s = \{ ColRecs(t) \mid t \in t_s \}$ **in**
.14 **if** $\underline{\text{err}} \notin r_s \wedge Compatible(r_s)$
.15 **then** $\underline{\text{Merge}}(r_s)$
.16 **else** $\underline{\text{err}}$
.17 $MkTag(\text{'OptionalType'}, t) \to ColRecs(t)$,
.18 $MkTag(\text{'SetType'}, t) \to ColRecs(t)$,
.19 $MkTag(\text{'Seq0Type'}, t) \to ColRecs(t)$,
.20 $MkTag(\text{'Seq1Type'}, t) \to ColRecs(t)$,
.21 $MkTag(\text{'GeneralMapType'}, (t_1, t_2)) \to$ **let** $r_s = \{ ColRecs(t_1), ColRecs(t_2) \}$ **in**
.22 **if** $\underline{\text{err}} \notin r_s \wedge Compatible(r_s)$
.23 **then** $\underline{\text{Merge}}(r_s)$
.24 **else** $\underline{\text{err}}$,
.25 $MkTag(\text{'InjectiveMapType'}, (t_1, t_2)) \to$ **let** $r_s = \{ ColRecs(t_1), ColRecs(t_2) \}$ **in**
.26 **if** $\underline{\text{err}} \notin r_s \wedge Compatible(r_s)$
.27 **then** $\underline{\text{Merge}}(r_s)$
.28 **else** $\underline{\text{err}}$,
.29 $MkTag(\text{'FnType'}, (t_1, t_2)) \to$ **let** $r_s = \{ ColRecs(t_1), ColRecs(t_2) \}$ **in**
.30 **if** $\underline{\text{err}} \notin r_s \wedge Compatible(r_s)$
.31 **then** $\underline{\text{Merge}}(r_s)$
.32 **else** $\underline{\text{err}}$,
.33 $MkTag(\text{'BasicType'}, _) \to \{ \mapsto \}$,
.34 $MkTag(\text{'TypeId'}, _) \to \{ \mapsto \}$,
.35 $MkTag(\text{'TypeVar'}, _) \to \{ \mapsto \}$,
.36 $MkTag(\text{'QuoteType'}, _) \to \{ \mapsto \}$

Annotations to *ColRecs*:

193. This auxiliary function takes a syntactic type and collects all the composite types inside that type recursively. This is necessary in order to construct all field selectors and record modifiers for all of these composite types.

End of annotations

194. $ColTypeList : \mathbb{L}(Field) \cup \mathbb{L}(Par) \to \mathbb{L}(Type)$

194.1 $ColTypeList(f_l) \triangleq$
.2 **if** $f_l = [\,]$
.3 **then** $[\,]$
.4 **else cases** $\underline{\text{hd}}(f_l)$:
.5 $MkTag(\text{'Field'}, (_, type)) \to \underline{\text{join}}([\,type\,], ColTypeList(\underline{\text{tl}}(f_l)))$,
.6 $MkTag(\text{'Par'}, (_, type)) \to \underline{\text{join}}([\,type\,], ColTypeList(\underline{\text{tl}}(f_l)))$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annotations to *ColTypeList*:

194. This auxiliary function collects a list of syntactic types from either a list of fields or a list of parameters.

End of annotations

195. $ColIdSet : Pattern \cup Bind \to \mathbb{F}(Id)$

195.1 $ColIdSet(pb) \triangleq$

.2 **cases** pb :

.3 $MkTag(\text{'PatternId'}, \underline{\text{nil}}) \to \{\ \}$,

.4 $MkTag(\text{'PatternId'}, id) \to \{id\}$,

.5 $MkTag(\text{'MatchVal'}, _) \to \{\ \}$,

.6 $MkTag(\text{'SeqEnumPattern'}, p_l) \to \bigcup\{\ ColIdSet(p) \mid p \in \underline{\text{elems}}(p_l)\ \}$,

.7 $MkTag(\text{'SeqConcPattern'}, (p_1, p_2)) \to ColIdSet(p_1) \cup ColIdSet(p_2)$,

.8 $MkTag(\text{'RecordPattern'}, (_, p_l)) \to \bigcup\{\ ColIdSet(p) \mid p \in \underline{\text{elems}}(p_l)\ \}$,

.9 $MkTag(\text{'TuplePattern'}, p_l) \to \bigcup\{\ ColIdSet(p) \mid p \in \underline{\text{elems}}(p_l)\ \}$,

.10 $MkTag(\text{'SetUnionPattern'}, (p_1, p_2)) \to ColIdSet(p_1) \cup ColIdSet(p_2)$,

.11 $MkTag(\text{'SetEnumPattern'}, p_s) \to \bigcup\{\ ColIdSet(p) \mid p \in p_s\ \}$,

.12 $MkTag(\text{'SetConstrPattern'}, (p, _)) \to ColIdSet(p)$,

.13 $MkTag(\text{'TypeConstrPattern'}, (p, _)) \to ColIdSet(p)$,

.14 $MkTag(\text{'SetBind'}, (p, _)) \to ColIdSet(p)$,

.15 $MkTag(\text{'TypeBind'}, (p, _)) \to ColIdSet(p)$

Annotations to *ColIdSet*:

195. This auxiliary function collects a set of pattern identifiers used inside either a pattern or a binding.

End of annotations

196. $ColIdList : \mathbb{L}(Par) \cup \mathbb{L}(TypeVar) \to \mathbb{L}(Id)$

196.1 $ColIdList(params) \triangleq$

.2 **if** $params = [\ ]$

.3 **then** $[\ ]$

.4 **else cases** $\underline{\text{hd}}(params)$:

.5 $MkTag(\text{'Par'}, (id, _)) \to \underline{\text{join}}([id], ColIdList(\underline{\text{tl}}(params)))$,

.6 $MkTag(\text{'TypeVar'}, id) \to \underline{\text{join}}([id], ColIdList(\underline{\text{tl}}(params)))$

Annotations to *ColIdList*:

196. This auxiliary function collects a list of pattern identifiers from either a list of parameters or a list of type variables.

End of annotations

197. $ColParList : ExplPolyFnDef \cup ExplFnDef \to \mathbb{L}(Par)$

197.1 $ColParList(def) \triangleq$

.2 **cases** def :

.3 $MkTag(\text{'ExplFnDef'}, (_, _, _, body, _)) \to ColPars(body)$,

.4 $MkTag(\text{'ExplPolyFnDef'}, (_, _, _, body, _)) \to ColPars(body)$

Annotations to *ColParList*:

197. This auxiliary function collects a list of (Curried) parameters from a function definition. It is defined in terms of *ColPars*.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

198. $ColPars : Expr \to \mathbb{L}(Par)$

198.1 $ColPars(expr) \triangleq$
.2 **if** $ShowTag(expr) \neq$ *'Lambda'*
.3 **then** [ ]
.4 **else let** $MkTag(\text{'Lambda'}, (par, body)) = expr$ **in**
.5 $\underline{\text{join}}([par], ColPars(body))$

Annotations to *ColPars*:
198. This auxiliary function collects a list of (Curried) parameters from a syntactic expression (which is a body of a function definition).
End of annotations

199. $ColTypeVars : ExplPolyFnDef \to \mathbb{L}_1(Id)$

199.1 $ColTypeVars(MkTag(\text{'ExplPolyFnDef'}, (tv_l, _, _, _))) \triangleq$
.2 $[StripTag(tv_l(i)) \mid i \in \underline{\text{inds}}(tv_l)]$

Annotations to *ColTypeVars*:
199. This auxiliary function collects a list of identifiers from type variables from a polymorphic function definition.
End of annotations

200. $ColInvFns : \mathbb{E}(TypeDef) \to \mathbb{M}(Pattern, ExplFnDef)$

200.1 $ColInvFns(td_m) \triangleq$
.2 **let** $btp = MkTag(\text{'BasicType'}, \text{BOOLEAN})$,
.3 $pre = MkTag(\text{'BoolLit'}, \text{TRUE})$ **in**
.4 { **let** $invid = MkTag(\text{'PatternId'}, SelInvId(td_m(id)))$,
.5 $body = SelInvExpr(td_m(id))$ **in**
.6 $invid \mapsto MkTag(\text{'ExplFnDef'}, (SelShape(td_m(id)), btp, pre, body, \text{TOTAL}))$
.7 $\mid id \in \underline{\text{dom}}(td_m)$ }

Annotations to *ColInvFns*:
200. This auxiliary function collects all invariant predicate functions connected to the type definitions given as arguments. For each type definition an explicit function definition is created for the invariant.
End of annotations

201. $ColSelMap : \mathbb{L}(Field) \to \mathbb{E}(\mathbb{N}) \cup \{\underline{\text{err}}\}$

201.1 $ColSelMap(f_l) \triangleq$
.2 **if** $\underline{\text{card}}(\underline{\text{elems}}(f_l)) \neq \underline{\text{len}}(f_l)$
.3 **then** $\underline{\text{err}}$
.4 **else** $\{ SelSel(f_l(i)) \mapsto i \mid i \in \underline{\text{inds}}(f_l) \cdot SelSel(f_l(i)) \neq \underline{\text{nil}} \}$

Annotations to *ColSelMap*:
201. This auxiliary function takes a list of fields from a record type and yields a map from selector identifiers to their index in the list.
End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.10 Selector Functions

```
202.   SelPre : ImplFnDef ∪ ImplPolyFnDef ∪ ExplFnDef ∪ ExplPolyFnDef → Expr
202.1  SelPre(def) ≜
  .2     cases ShowTag(def):
  .3       'ImplFnDef'       → let MkTag('ImplFnDef', (_, pre, _)) = def in
  .4                             pre,
  .5       'ImplPolyFnDef'   → let MkTag('ImplPolyFnDef', (_, _, pre, _)) = def in
  .6                             pre,
  .7       'ExplFnDef'       → let MkTag('ExplFnDef', (_, _, pre, _, _)) = def in
  .8                             pre,
  .9       'ExplPolyFnDef'   → let MkTag('ExplPolyFnDef', (_, _, pre, _, _)) = def in
  .10                            pre
```

Annotations to *SelPre*:

202. This auxiliary function selects a pre-condition expression from any kind of function definition.

End of annotations

```
203.   SelPost : ImplFnDef ∪ ImplPolyFnDef → Expr
203.1  SelPost(def) ≜
  .2     cases ShowTag(def):
  .3       'ImplFnDef'       → let MkTag('ImplFnDef', (_, _, post)) = def in
  .4                             post,
  .5       'ImplPolyFnDef'   → let MkTag('ImplPolyFnDef', (_, _, _, post)) = def in
  .6                             post
```

Annotations to *SelPost*:

203. This auxiliary function selects a post-condition expression from an implicit function definition.

End of annotations

```
204.   SelRes : ImplFnDef ∪ ImplPolyFnDef → Id
204.1  SelRes(def) ≜
  .2     cases ShowTag(def):
  .3       'ImplFnDef'       → let MkTag('ImplFnDef', (h, _, _)) = def in
  .4                             let MkTag('ImplFnHeading', (_, _, resum)) = h in
  .5                               resum,
  .6       'ImplPolyFnDef'   → let MkTag('ImplPolyFnDef', (h, _, _, _)) = def in
  .7                             let MkTag('ImplFnHeading', (_, _, resum)) = h in
  .8                               resum
```

Annotations to *SelRes*:

204. This auxiliary function selects a result identifier from an implicit function definition.

End of annotations

```
205.   SelBody : ExplFnDef ∪ ValDef → Expr
205.1  SelBody(expldef) ≜
  .2     cases expldef :
  .3       MkTag('ExplFnDef', (_, _, _, body, _)) → body,
  .4       MkTag('ValDef', (_, expr))             → expr
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annotations to *SelBody*:

205. This auxiliary function selects the body of either an explicit function definition or a value definition.

End of annotations

206. $SelKind : ExplFnDef \cup ExplPolyFnDef \to \{ \text{TOTAL, PARTIAL} \}$

206.1 $SelKind(explfndef) \triangleq$

.2 **cases** *explfndef* :

.3 $MkTag(\text{'}ExplFnDef\text{'}, (_, _, _, _, kind)) \quad \to kind,$

.4 $MkTag(\text{'}ExplPolyFnDef\text{'}, (_, _, _, _, kind)) \to kind$

Annotations to *SelKind*:

206. This auxiliary function selects the function kind (either total or partial) from either a monomorphic or polymorphic explicit function definition.

End of annotations

207. $SelTypeVars : ExplPolyFnDef \to \mathbb{L}(TypeVar)$

207.1 $SelTypeVars(MkTag(\text{'}ExplPolyFnDef\text{'}, (tpvars, _, _, _, _))) \triangleq$

.2 $[\textbf{let } MkTag(\text{'}TypeVar\text{'}, id) = tpvars(i) \textbf{ in } id \mid i \in \underline{\text{inds}}(tpvars)]$

Annotations to *SelTypeVars*:

207. This auxiliary function selects the list of type variables from a polymorphic explicit function definition.

End of annotations

208. $SelPolyFn : ExplPolyFnDef \to \mathbb{L}_1(TypeVar) \times Lambda$

208.1 $SelPolyFn(MkTag(\text{'}ExplPolyFnDef\text{'}, (tpvars, _, _, lambda, _))) \triangleq$

.2 $(tpvars, lambda)$

Annotations to *SelPolyFn*:

208. This auxiliary function selects the list of type variables and the body from a polymorphic explicit function definition.

End of annotations

209. $SelShape : TypeDef \to Type$

209.1 $SelShape(MkTag(\text{'}TypeDef\text{'}, (shape, _))) \triangleq$

.2 $shape$

Annotations to *SelShape*:

209. This auxiliary function selects the shape (i.e. a type) of a type definition.

End of annotations

210. $SelInvId : TypeDef \to Id$

210.1 $SelInvId(MkTag(\text{'}TypeDef\text{'}, (_, (id, _)))) \triangleq$

.2 id

Annotations to *SelInvId*:

210. This auxiliary function selects the invariant identifier from a type definition.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

211. *SelInvExpr* : *TypeDef* → *Lambda*

211.1 *SelInvExpr*(*MkTag*('*TypeDef*', (_, (_, *lambda*)))) ≜
.2 *lambda*

Annotations to *SelInvExpr*:
211. This auxiliary function selects the invariant expression from a type definition.
End of annotations

212. *SelSel* : *Field* → *Id* ∪ { nil }

212.1 *SelSel*(*MkTag*('*Field*', (*sel*, _))) ≜
.2 **if** *sel* = nil
.3 **then** nil
.4 **else** *sel*

Annotations to *SelSel*:
212. This auxiliary function selects the selector identifier from *Field*, if present.
End of annotations

213. *SelExtVarInfMode* : *ExtVarInf* → { READ, READWRITE }

213.1 *SelExtVarInfMode*(*MkTag*('*ExtVarInf*', (_, _, *mode*))) ≜
.2 *mode*

Annotations to *SelExtVarInfMode*:
213. This auxiliary function selects the mode from a *ExtVarInf*.
End of annotations

214. *SelExtVarInfId* : *ExtVarInf* → token

214.1 *SelExtVarInfId*(*MkTag*('*ExtVarInf*', (_, *MkTag*(_, *token*), _))) ≜
.2 *token*

Annotations to *SelExtVarInfId*:
214. This auxiliary function selects the token of the identifier from *ExtVarInf*.
End of annotations

215. *SelPattern* : *Bind* → *Pattern*

215.1 *SelPattern*(*bind*) ≜
.2 **cases** *bind* :
.3 *MkTag*('*TypeBind*', (*p*, _)) → *p*,
.4 *MkTag*('*SetBind*', (*p*, _)) → *p*

Annotations to *SelPattern*:
215. This auxiliary function selects the pattern of a binding.
End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

8.7.11 Tag-processing Functions

216. $MkTag : Id \times VAL \to VAL$

216. $MkTag$ must satisfy :

(1) $\forall\ id \in Id,\ val \in VAL \backslash \{\perp\} \cdot \|T^{id}(\{val\}_{\perp}, \{val\})\| = \{MkTag(id, val)\}$

Annotations to *MkTag*:

216. *MkTag* is the corresponding tagging function on *values* associated with the tagging operator from the domain universe. Note that if a tagged value without any extra contents are needed, the value is modelled as an empty tuple written as () (this is done for some of the identifier values where the tag provide sufficient information). This function will for notational convenience be used for both syntactic and semantic entities. Alternatively the abstract syntax could be translated into the semantic domain (i.e. $CAS \subseteq VAL$) as it was done for the STC/VDM RL in Mon86. However, this trivial translation of the abstract syntax to its semantical counterpart is not necessary.

End of annotations

Given this function other useful tag-manipulation functions can be specified using the above:

217. $ShowTag : VAL \to Id$

217.1 $ShowTag(MkTag(id, _)) \triangleq$

.2 id

Annotations to *ShowTag*:

217. Get the tag.

End of annotations

218. $StripTag : VAL \to VAL$

218.1 $StripTag(MkTag(_, v)) \triangleq$

.2 v

Annotations to *StripTag*:

218. Strip the tag from the value.

End of annotations

219. $StripRecTagVal : VAL \to \mathbb{L}(VAL) \cup \{\underline{\text{err}}\}$

219.1 $StripRecTagVal(v) \triangleq$

.2 **if** $ShowTag(v) =$ *'record'*

.3 **then let** $MkTag(\text{'record'}, l) = StripTag(v)$ **in**

.4 l

.5 **else** $\underline{\text{err}}$

Annotations to *StripRecTagVal*:

219. Strip the tags from a record value. Notice that the value $StripTag(v)$ contains the tag which identifies the record identifier.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
220.    ShowRecTagVal : VAL → Id ∪ { err }
220.1   ShowRecTagVal(v) ≜
   .2     if ShowTag(v) = 'record'
   .3     then let MkTag(id, _) = StripTag(v) in
   .4             id
   .5     else err
```

Annotations to *ShowRecTagVal*:

220. Shows the tag from a record value.

End of annotations

```
221.    StripSeqTagVal : VAL → IL(VAL) ∪ { err }
221.1   StripSeqTagVal(v) ≜
   .2     if ShowTag(v) = 'seq'
   .3     then let MkTag('seq', l) = StripTag(v) in
   .4             l
   .5     else err
```

Annotations to *StripSeqTagVal*:

221. Strip the *'seq'* tag.

End of annotations

```
222.    StripSetTagVal : VAL → IF(VAL) ∪ { err }
222.1   StripSetTagVal(v) ≜
   .2     if ShowTag(v) = 'set'
   .3     then let MkTag('set', s) = StripTag(v) in
   .4             s
   .5     else err
```

Annotations to *StripSetTagVal*:

222. Strip the *'set'* tag.

End of annotations

```
223.    StripMapTagVal : VAL → IM(VAL, VAL) ∪ { err }
223.1   StripMapTagVal(v) ≜
   .2     if ShowTag(v) = 'map'
   .3     then let MkTag('map', m) = StripTag(v) in
   .4             m
   .5     else err
```

Annotations to *StripMapTagVal*:

223. Strip the *'map'* tag.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

224. $StripNumTagVal : VAL \to \mathbb{R} \cup \{\underline{\text{err}}\}$

224.1 $StripNumTagVal(v) \triangleq$
.2 **if** $ShowTag(v) = \text{'num'}$
.3 **then let** $MkTag(\text{'num'}, n) = StripTag(v)$ **in**
.4 n
.5 **else** err

Annotations to *StripNumTagVal*:
224. Strip the *'num'* tag.
End of annotations

8.7.12 General Functions and Predicates

Most of the functions and predicates in this clause are polymorphic. The convention is to use X and Y as type variables ranging over arbitrary sets, and C ranges over cpo domains (from the domain universe). Because of the general nature of these functions some of them need to use notation which have been introduced in the clause about the domain universe.

225. $Injective : \mathbf{Pred}(\mathbb{M}(X, Y))$

225.1 $Injective(m) \triangleq$
.2 $\forall x_1, x_2 \in \underline{\text{dom}}(m) \cdot x_1 \neq x_2 \Rightarrow m(x_1) \neq m(x_2)$

Annotations to *Injective*:
225. This predicates checks whether a map is injective, i.e. that all domain elements are mapped to different range elements.
End of annotations

226. $IsFunCont : \mathbf{Pred}(FUN_VAL \times FUN_DOM)$

226.1 $IsFunCont(MkTag(\text{'fun'}, f), d) \triangleq$
.2 $(\forall x_1, x_2 \in \delta_0(f) \cdot x_1 \sqsubseteq_{\delta_0(d)} x_2 \Rightarrow f(x_1) \sqsubseteq_{\delta_1(d)} f(x_2)) \wedge$
.3 $(\forall \{ x_i \mid i \in \omega \} \subseteq \delta_0(f) \cdot \forall i \in \omega \cdot x_i \sqsubseteq_{\delta_0(d)} x_{i+1} \Rightarrow$
.4 $f(\bigsqcup_{\delta_0(d)} \{ x_i \mid i \in \omega \}) = \bigsqcup_{\delta_1(d)} \{ f(x_i) \mid i \in \omega \})$

Annotations to *IsFunCont*:
226. The predicate *IsFunCont* verifies a given function f is continuous with respect to the ordering from the corresponding function domain d.
End of annotations

227. $IsCont_C : \mathbf{Pred}(C \to C)$

227.1 $IsCont_C(f) \triangleq$
.2 $(\forall x_1, x_2 \in |C| \cdot x_1 \sqsubseteq_C x_2 \Rightarrow f(x_1) \sqsubseteq_C f(x_2)) \wedge$
.3 $(\forall \{ x_i \mid i \in \omega \} \subseteq |C| \cdot \forall i \in \omega \cdot x_i \sqsubseteq_C x_{i+1} \Rightarrow$
.4 $f(\bigsqcup_C \{ x_i \mid i \in \omega \}) = \bigsqcup_C \{ f(x_i) \mid i \in \omega \})$

Annotations to $IsCont_C$:
227. The predicate $IsCont_C$ verifies a given cpo C if a function $f : C \to C$ is continuous with respect to the ordering $\sqsubseteq_C$.
End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

228. $IsCpo$: **Pred**($QUASI_DOM$)

228.1 $IsCpo(((|A|, \sqsubseteq_A), \|A\|)) \triangleq$
.2 $\forall x \in |A| \cdot x \sqsubseteq_A x \wedge$
.3 $\forall x, y \in |A| \cdot x \sqsubseteq_A y \wedge y \sqsubseteq_A x \Rightarrow x = y \wedge$
.4 $\forall x, y, z \in |A| \cdot x \sqsubseteq_A y \wedge y \sqsubseteq_A z \Rightarrow x \sqsubseteq_A z \wedge$
.5 $\exists bot \in |A| \cdot \forall x \in |A| \cdot bot \sqsubseteq_A x \wedge$
.6 $\forall f \in \mathbb{N} \to A \cdot (\forall i, j \in \mathbb{N} \cdot i \leq j \Rightarrow f(i) \sqsubseteq_A f(j)) \Rightarrow$
.7 $\exists lub \in |A| \cdot (\forall i \in \mathbb{N} \cdot f(i) \sqsubseteq_A lub) \wedge$
.8 $(\forall ub \in |A| \cdot (\forall i \in \mathbb{N} \cdot f(i) \sqsubseteq_A ub) \Rightarrow lub \sqsubseteq_A ub)$

Annotations to *IsCpo*:

228. This predicate checks whether a domain from $QUASI_DOM$ forms a CPO (see definition 7.7 in clause 7). If it forms a CPO the argument domain will belong to the 'real' domain universe, DOM.

.2–.4 Here it is checked whether $\sqsubseteq_A$ is a partial ordering. The ordering must be reflexive (.2), antisymmetric (.3), and transitive (.4) as defined in definition 7.1 in clause 7.

.5 Here it is checked that there exists a least element in the CPO (see also definition 7.5 in clause 7).

.6–.8 Here it is checked that there exists a least upper bound for each countable chain in the domain (see definition 7.6 in clause 7).

End of annotations

229. $Reverse : \mathbb{L}(X) \to \mathbb{L}(X)$

229.1 $Reverse(l) \triangleq$
.2 **if** $l = [\,]$
.3 **then** $[\,]$
.4 **else** $\underline{\text{join}}(Reverse(\underline{\text{tl}}(l)), [\underline{\text{hd}}(l)])$

Annotations to *Reverse*:

229. This is an auxiliary function that simply reverses a sequence.

End of annotations

230. $ApplySeq : (X \to Y) \to \mathbb{L}(X) \to \mathbb{L}(Y)$

230.1 $ApplySeq(func)(seq) \triangleq$
.2 **if** $seq = [\,]$
.3 **then** $[\,]$
.4 **else** $\underline{\text{join}}([func(\underline{\text{hd}}(seq))], ApplySeq(func)(\underline{\text{tl}}(seq)))$

Annotations to *ApplySeq*:

230. The result of applying $ApplySeq(func)$ to a sequence seq is equivalent to applying $func$ to each element of seq (in the functional programming "folklore" this function is often called Map).

End of annotations

231. $Iota : \mathbb{F}(X) \to X$

231.1 $Iota(x) \triangleq$
.2 **if** $\underline{\text{card}}(x) = 1$
.3 **then let** $\{v\} = x$ **in**
.4 v
.5 **else** $\perp$

Annotations to *Iota*:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

231. This auxiliary function yields the unique element from a singleton set. Note that *Iota* returns $\perp$ if the argument is not a singleton set.

End of annotations

232. $Choose : \mathbb{P}(X) \to X$

232. *Choose* must satisfy :

(1) $\forall\ s \in \mathbb{P}(X) \cdot\ s \neq \{\ \} \Rightarrow Choose(s) \in s$

Annotations to *Choose*:

232. This auxiliary function selects an arbitrary element from a non-empty arbitrary set.

End of annotations

233. $BotEnv$: **Pred**(ENV)

233.1 $BotEnv(env) \triangleq$

.2 $\perp \in \underline{\text{rng}}(env)$

Annotations to *BotEnv*:

233. This function is an auxiliary predicate which checks whether any bindings occur to the bottom value.

End of annotations

234. $SetToSeq : \mathbb{F}(X) \to \mathbb{F}(\mathbb{L}(X))$

234.1 $SetToSeq(set) \triangleq$

.2 **if** $set = \{\ \}$

.3 **then** $\{[\ ]\}$

.4 **else** $\{\underline{\text{join}}([\ e\ ],\ rest) \mid e \in set,\ rest \in SetToSeq(set \backslash \{\ e\ \})\}$

Annotations to *SetToSeq*:

234. This auxiliary function takes a finite set of elements and yields the finite set of all possible lists of the argument (i.e. sequences consisting precisely of the elements of the given set). This function is defined by induction over the cardinality of the set.

End of annotations

235. $StepsToSeq : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{L}(NUM_VAL) \cup \{\underline{\text{err}}\}$

235.1 $StepsToSeq(lb,\ ub,\ step) \triangleq$

.2 **if** $step = 0$

.3 **then** $\underline{\text{err}}$

.4 **else if** $((lb < ub) \wedge (step < 0)) \vee ((lb > ub) \wedge (step > 0))$

.5 **then** $[\ ]$

.6 **else** $\underline{\text{join}}([\ MkTag(\text{'num'},\ lb)\ ],\ StepsToSeq(lb + step,\ ub,\ step))$

Annotations to *StepsToSeq*:

235. This auxiliary function takes a lower bound, an upper bound and a step size from a for-loop, and creates a list of numeric values in the given interval with the given step size.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

236. $NonDetIter : \mathbb{P}(X) \times (X \times X \to X) \to \mathbb{P}(X)$

236.1 $NonDetIter(x_s,\ comp) \triangleq$
.2 **if** $\forall x \in x_s,\ y,\ z \in X \cdot y \sqsubseteq_X z \Rightarrow comp(x,\ y) \sqsubseteq_X comp(x,\ z)$
.3 **then let** $h = \lambda l \in \mathbb{IL}_\omega(x_s)$.**if** $l \in \mathbb{IL}(x_s)$
.4 **then if** $\underline{\text{len}}(l) = 0$
.5 **then** $\bot$
.6 **else** $comp(\underline{\text{hd}}(l),\ h(\underline{\text{tl}}(l)))$
.7 **else** $\bigsqcup_X \{ h(ApprList(l)(n)) \mid n \in \mathbb{N} \}$ **in**
.8 $\{ h(l) \mid l \in \mathbb{IL}_\omega(x_s) \}$
.9 **else** $\{\ \}$

Annotations to *NonDetIter*:

236. This function, given a subset x_s of X and a function $comp : X \times X \to X$ continuous with respect to the second argument, returns another subset of X which is a fixed point of a functional $\lambda A.\{ comp(x,\ a) \mid x \in x_s \wedge a \in A \}$.
However, this is not claimed to be a least fixed point, in fact no order between subsets of X is introduced. Actually, it is sufficient that *comp* be defined only on $x_s \times X$.

.4–.7 An auxiliary function, h, responsible for iterations over all sequences is defined.

.4–.6 Its definition for a finite sequence goes by induction over the length of the sequence.

.7 h is defined for an infinite sequence as the least upper bound of its values over all finite approximations of the sequence. In order to be correct the clause h has to be monotone, which holds due to the monotonicity of *comp* with respect to the second argument.

.8 The value of the main function is defined as an image of the set of infinite sequences under the auxiliary function h. This set is a fixed point of the above mentioned functional due to the continuity of *comp* with respect to the second argument.

End of annotations

237. $ApprList : \mathbb{IL}_\omega(X) \to \mathbb{N} \to \mathbb{IL}(X)$

237.1 $ApprList(l) \triangleq$
.2 λn.**if** $n = 0$
.3 **then** $[\ ]$
.4 **else** $\underline{\text{join}}([\underline{\text{hd}}(l)],\ ApprList(\underline{\text{tl}}(l))(n - 1))$

Annotations to *ApprList*:

237. This function, given an infinite sequence as an argument and a natural number as a second (Curried) argument, returns a finite sequence of length, n, which consists of the first n elements of the original infinite sequence.

End of annotations

238. $PropErr : \mathbb{P}(X \to \mathbb{P}(Y)) \to \mathbb{P}(X \to (Y \cup \{\underline{\text{err}}\}))$

238.1 $PropErr(mul_ldef_s) \triangleq$
.2 $\{ def \mid def \in (X \to Y \cup \{\underline{\text{err}}\}) \cdot$
.3 $\exists m_ldef \in mul_ldef_s \cdot \forall x \in X \cdot$ **if** $m_ldef(x) = \{\ \}$
.4 **then** $def(x) = \underline{\text{err}}$
.5 **else** $def(x) \in m_ldef(x) \}$

Annotations to *PropErr*:

238. This auxiliary function propagates looseness one level out. It is polymorphic because such a propagation is often required. It takes a set of functions from some domain X to a codomain which is a set (possibly of another domain, but not necessarily). The result is a set of functions which map the domain elements to a member of the previous set. If the set is empty, err is returned. Note that all places where *PropErr* is called and where err cannot be used, it is ensured that the sets from the codomain are non-empty.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

End of annotations

239. $PropS : \mathbb{P}(X \to \mathbb{P}(Y)) \to \mathbb{P}(X \to (Y \cup (X \times \{\underline{\text{ret}}\} \times \{\bot\})))$

239.1 $PropS(mul_ldef_s) \triangleq$
.2 $\{ def \mid def \in (X \to Y \cup \{\underline{\text{err}}\}) \cdot$
.3 $\exists m_ldef \in mul_ldef_s \cdot \forall x \in X \cdot$ **if** $m_ldef(x) = \{\}$
.4 **then** $def(x) = (x, \underline{\text{ret}}, \bot)$
.5 **else** $def(x) \in m_ldef(x) \}$

Annotations to *PropS*:

239. This auxiliary function propagates looseness one level out. It is polymorphic because such a propagation is often required, but it is mainly used by evaluation functions for statements. It takes a set of functions from some domain X to a codomain which is a set (possibly of another domain, but not necessarily). The result is a set of functions which map the domain elements to a member of the previous set. If the set is empty a three-tuple with the value from the X domain, $\underline{\text{ret}}$ and $\bot$, is returned. Note that all places where *PropS* is called, where this three-tuple cannot be used, it is ensured that the sets from the codomain are non-empty.

End of annotations

240. $PropE : \mathbb{P}(X \to \mathbb{P}(Y)) \to \mathbb{P}(X \to (Y \cup \{\bot\}))$

240.1 $PropE(mul_ldef_s) \triangleq$
.2 $\{ def \mid def \in (X \to Y \cup \{\underline{\text{err}}\}) \cdot$
.3 $\exists m_ldef \in mul_ldef_s \cdot \forall x \in X \cdot$ **if** $m_ldef(x) = \{\}$
.4 **then** $def(x) = \bot$
.5 **else** $def(x) \in m_ldef(x) \}$

Annotations to *PropE*:

240. This auxiliary function propagates looseness one level out. It is polymorphic because such a propagation is often required. It takes a set of functions from some domain X to a codomain which is a set (possibly of another domain, but not necessarily). The result is a set of functions which maps the domain elements to a member of the previous set. If the set was empty bottom is returned.

End of annotations

241. $Compatible$: **Pred**($\mathbb{P}(\mathbb{M}(X, Y))$)

241.1 $Compatible(map_s) \triangleq$
.2 $\underline{\text{card}}(\bigcup\{ \underline{\text{dom}}(map) \mid map \in map_s \}) < \omega \wedge$
.3 $\forall\ map_1, map_2 \in map_s \cdot$
.4 $\forall\ x \in (\underline{\text{dom}}(map_1) \cap \underline{\text{dom}}(map_2)) \cdot map_1(x) = map_2(x)$

Annotations to *Compatible*:

241. This auxiliary predicate takes a set of maps and checks that the maps are compatible such that a distributed merge can be used.
.2 The size of such a distributed merge must be finite.
.3–.4 All overlapping domain values must be bound to the same range value.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

242. $Partition : \mathbb{IP}(\mathbb{IP}(X)) \to (\mathbb{IP}(\mathbb{IP}(X)) \cup \{\underline{\text{err}}\})$

242.1 $Partition(set_s) \triangleq$
.2 **if** $\{\ \} \in set_s$
.3 **then** $\underline{\text{err}}$
.4 **else** $\{ s \mid s \in \mathbb{IP}(X) \cdot \underline{\text{card}}(s) \leq \underline{\text{card}}(set_s) \wedge$
.5 $\forall\ set \in set_s \cdot \exists\ elem \in s \cdot elem \in set \wedge$
.6 $\underline{\text{card}}(\{ e \mid e \in s \cdot \forall s_1 \in set_s \backslash \{ set \} \cdot e \notin s_1 \}) \leq 1 \}$

Annotations to *Partition*:

242. This auxiliary function takes a set of sets, and partitions it into a set of sets where each set contains an element from all the original sets.

.4 The cardinality of each set must be smaller than, or equal to, the cardinality of the given sets (otherwise more than one from each argument set may be included). A set will be smaller if some of the given sets overlap.

.5 For all of the given sets there must exists an element which belongs to the (element) set, s.

.6 For all of the given sets it must also hold that at most one element in the (element) set does not belong to any of the other elements of the given sets. This ensures that multiple (new) elements are not chosen from one of the given sets (this could be possible when overlapping is present).

End of annotations

243. $Permutations : \mathbb{IL}(X) \to \mathbb{IF}(\mathbb{IL}(X))$

243.1 $Permutations(list) \triangleq$
.2 $\{ l \mid l \in \mathbb{IL}(X) \cdot \underline{\text{len}}(list) = \underline{\text{len}}(l) \wedge$
.3 $\forall e \in \underline{\text{elems}}(l) \cdot \underline{\text{card}}(\{ j \mid j \in \underline{\text{inds}}(l) \wedge l(j) = e \}) =$
.4 $\underline{\text{card}}(\{ j \mid j \in \underline{\text{inds}}(list) \wedge list(j) = e \}) \}$

Annotations to *Permutations*:

243. This auxiliary function takes a list of elements and yields a set of permutations of that list.

End of annotations

244. $AllPairComb : \mathbb{IL}(\mathbb{IP}(X) \times \mathbb{IP}(X)) \to \mathbb{IP}(\mathbb{IL}(X \times X))$

244.1 $AllPairComb(set_p_l) \triangleq$
.2 $\{ l \mid l \in \mathbb{IL}(X \times X) \cdot \underline{\text{len}}(l) = \underline{\text{len}}(set_p_l) \wedge$
.3 $\forall\ i, j \in \underline{\text{inds}}(l) \cdot$
.4 **let** $(el_{11}, el_{12}) = l(i),$
.5 $(el_{21}, el_{22}) = l(j),$
.6 $(set_{11}, set_{12}) = set_p_l(i),$
.7 $(set_{21}, set_{22}) = set_p_l(j)$ **in**
.8 $el_{11} \in set_{11} \wedge el_{12} \in set_{12} \wedge$
.9 $el_{21} \in set_{21} \wedge el_{22} \in set_{22} \}$

Annotations to *AllPairComb*:

244. This auxiliary function transforms a list of pairs (of sets of values belonging to some type X) into a set of lists of pairs (of values belonging to X). Looseness inside the list is propagated one level up.

End of annotations

245. $True : () \to BOOL_VAL$

245.1 $True() \triangleq$
.2 $MkTag('bool', \mathsf{T})$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annotations to *True*:

245. This auxiliary function is just a short hand for the 'truth' element in *BOOL_VAL*.

End of annotations

246. $False : () \rightarrow BOOL_VAL$

246.1 $False() \triangleq$

.2 $MkTag(\text{'bool'}, \mathsf{F})$

Annotations to *False*:

246. This auxiliary function is just a short hand for the 'falsehood' element in *BOOL_VAL*.

End of annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9 The Mathematical Concrete Syntax

9.1 Document

document = definition block, { definition block } ;

9.2 Definitions

definition block = type definitions
| state definition
| value definitions
| function definitions
| operation definitions ;

9.2.1 Type Definitions

type definitions = 'types', type definition, { ';', type definition } ;
type definition = untagged type definition | tagged type definition ;
untagged type definition = identifier, '=', type, [invariant] ;
tagged type definition = identifier, '::', field list, [invariant] ;
type = bracketed type
| basic type
| quote type
| composite type
| union type
| product type
| optional type
| set type
| seq type
| map type
| function type
| type name
| type variable ;
bracketed type = '(', type, ')' ;
basic type = '$\mathbb{B}$' | '$\mathbb{N}$' | '$\mathbb{N}_1$' | '$\mathbb{Z}$' | '$\mathbb{Q}$' | '$\mathbb{R}$' | 'char' | 'token' ;
quote type = quote literal ;
composite type = 'compose', identifier, 'of', field list, 'end' ;
field list = { field } ;
field = [identifier, ':'], type ;
union type = type, '|', type, { '|', type } ;
product type = type, '×', type, { '×', type } ;
optional type = '[', type, ']' ;
set type = type, '-set' ;
seq type = seq0 type | seq1 type ;
seq0 type = type, '*' ;
seq1 type = type, '+' ;
map type = general map type | injective map type ;
general map type = type, '$\xrightarrow{m}$', type ;
injective map type = type, '$\xleftrightarrow{m}$', type ;
function type = partial function type | total function type ;
partial function type = discretionary type, '→', type ;
total function type = discretionary type, '$\xrightarrow{t}$', type ;
discretionary type = type | '(', ')' ;
type name = name ;
type.variable = type variable identifier ;
type variable identifier = '@', identifier ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.2.2 State Definition

state definition = 'state', identifier, 'of',
field list,
[invariant],
[initialization],
'end' ;
invariant = 'inv', invariant initial function ;
initialization = 'init', invariant initial function ;
invariant initial function = pattern, ' $\triangleq$ ', expression ;

9.2.3 Value Definitions

value definitions = 'values', value definition, { ';', value definition } ;
value definition = pattern, [':', type], '=', expression ;

9.2.4 Function Definitions

function definitions = 'functions', function definition, { ';', function definition } ;
function definition = explicit function definition | implicit function definition ;
explicit function definition = identifier, [type variable list], ':', function type,
identifier, parameters list,
' $\triangleq$ ', expression,
['pre', expression] ;
implicit function definition = identifier, [type variable list],
parameter types, identifier type pair,
['pre', expression],
'post', expression ;
type variable list = '[', type variable identifier, { ',', type variable identifier }, ']' ;
identifier type pair = identifier, ':', type ;
parameter types = '(', [pattern type pair list], ')' ;
pattern type pair list = pattern type pair, { ',', pattern type pair } ;
pattern type pair = pattern list, ':', type ;
parameters list = parameters, { parameters } ;
parameters = '(', [pattern list], ')' ;

9.2.5 Operation Definitions

operation definitions = 'operations', operation definition, { ';', operation definition } ;
operation definition = explicit operation definition | implicit operation definition ;
explicit operation definition = identifier, ':', operation type,
identifier, parameters,
' $\triangleq$ ', statement,
['pre', expression] ;
implicit operation definition = identifier, parameter types,
[identifier type pair],
[externals],
['pre', expression],
'post', expression,
[exceptions] ;
operation type = discretionary type, '$\xrightarrow{o}$', discretionary type ;
externals = 'ext', var information, { var information } ;
var information = mode, name list, [':', type] ;
mode = 'rd' | 'wr' ;
exceptions = 'errs', error list ;
error list = error, { error } ;
error = identifier, ':', expression, '$\rightarrow$', expression ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.3 Expressions

expression list = expression, { ',', expression } ;
expression = bracketed expression
| let expression
| let be expression
| def expression
| if expression
| cases expression
| unary expression
| binary expression
| quantified expression
| iota expression
| set enumeration
| set comprehension
| set range expression
| sequence enumeration
| sequence comprehension
| subsequence
| map enumeration
| map comprehension
| tuple constructor
| record constructor
| record modifier
| apply
| field select
| function type instantiation
| lambda expression
| is expression
| name
| old name
| symbolic literal ;

9.3.1 Bracketed Expressions

bracketed expression = '(', expression, ')' ;

9.3.2 Local Binding Expressions

let expression = 'let', local definition { ',', local definition }, 'in', expression ;
let be expression = 'let', bind, ['be', 'st', expression], 'in', expression ;
def expression = 'def', def bind, { ';', def bind }, 'in', expression ;
def bind = pattern bind, '=', expression ;

9.3.3 Conditional Expressions

if expression = 'if', expression, 'then', expression, { elseif expression }, 'else', expression ;
elseif expression = 'elseif', expression, 'then', expression ;
cases expression = 'cases', expression, ':',
cases expression alternatives,
[',', others expression],
'end' ;
cases expression alternatives = cases expression alternative, { ',', cases expression alternative } ;
cases expression alternative = pattern list, '→', expression ;
others expression = 'others', '→', expression ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.3.4 Unary Expressions

unary expression = prefix expression | map inverse expression ;
prefix expression = unary operator, expression ;
unary operator = unary plus
| unary minus
| arithmetic abs
| floor
| not
| set cardinality
| finite power set
| distributed set union
| distributed set intersection
| sequence head
| sequence tail
| sequence length
| sequence elements
| sequence indices
| distributed sequence concatenation
| map domain
| map range
| distributed map merge ;
unary plus = '+' ;
unary minus = '−' ;
arithmetic abs = 'abs' ;
floor = 'floor' ;
not = '¬' ;
set cardinality = 'card' ;
finite power set = '$\mathcal{F}$' ;
distributed set union = '$\bigcup$' ;
distributed set intersection = '$\bigcap$' ;
sequence head = 'hd' ;
sequence tail = 'tl' ;
sequence length = 'len' ;
sequence elements = 'elems' ;
sequence indices = 'inds' ;
distributed sequence concatenation = 'conc' ;
map domain = 'dom' ;
map range = 'rng' ;
distributed map merge = 'merge' ;
map inverse expression = expression, '$^{-1}$' ;

9.3.5 Binary Expressions

binary expression = expression, binary operator, expression ;
binary operator = arithmetic plus
| arithmetic minus
| arithmetic multiplication
| arithmetic divide
| arithmetic integer division
| arithmetic rem
| arithmetic mod
| less than
| less than or equal
| greater than
| greater than or equal
| equal
| not equal
| or
| and

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

| imply
| logical equivalence
| in set
| not in set
| subset
| proper subset
| set union
| set difference
| set intersection
| sequence concatenate
| map or sequence modify
| map merge
| map domain restrict to
| map domain restrict by
| map range restrict to
| map range restrict by
| composition
| iterate ;

arithmetic plus = '+' ;
arithmetic minus = '−' ;
arithmetic multiplication = '×' ;
arithmetic divide = '/' ;
arithmetic integer division = 'div' ;
arithmetic rem = 'rem' ;
arithmetic mod = 'mod' ;
less than = '<' ;
less than or equal = '≤' ;
greater than = '>' ;
greater than or equal = '≥' ;
equal = '=' ;
not equal = '≠' ;
or = '∨' ;
and = '∧' ;
imply = ' ⇒ ' ;
logical equivalence = ' ⇔ ' ;
in set = '∈' ;
not in set = '∉' ;
subset = '⊆' ;
proper subset = '⊂' ;
set union = '∪' ;
set difference = '\' ;
set intersection = '∩' ;
sequence concatenate = '⌢' ;
map or sequence modify = '†' ;
map merge = '⨿' ;
map domain restrict to = '◁' ;
map domain restrict by = '⩤' ;
map range restrict to = '▷' ;
map range restrict by = '⩥' ;
composition = '∘' ;
iterate = '↑' ;

NOTE — The ↑ infix operator can be replaced by a superscript: $m \uparrow n$ can be written as m^n.

9.3.6 Quantified Expressions

quantified expression = all expression
| exists expression
| exists unique expression ;
all expression = '∀', bind list, '·', expression ;
exists expression = '∃', bind list, '·', expression ;
exists unique expression = '∃!', bind, '·', expression ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.3.7 Iota Expression

iota expression = 'ι', bind, '·', expression ;

9.3.8 Set Expressions

set enumeration = '{', [expression list], '}' ;
set comprehension = '{', expression, '|', bind list, ['·', expression], '}' ;
set range expression = '{', expression, ',', '. . . ', ',', expression, '}' ;

9.3.9 Sequence Expressions

sequence enumeration = '[', [expression list], ']' ;
sequence comprehension = '[', expression, '|', set bind, ['·', expression], ']' ;
subsequence = expression, '(', expression, ',', '. . . ', ',', expression, ')' ;

9.3.10 Map Expressions

map enumeration = '{', maplet, { ',', maplet }, '}' | '{', '↦', '}' ;
maplet = expression, '↦', expression ;
map comprehension = '{', maplet, '|', bind list, ['·', expression], '}' ;

9.3.11 Tuple Constructor Expression

tuple constructor = '*mk*-', '(', expression, ',', expression list, ')' ;

9.3.12 Record Expressions

record constructor = name, '(', [expression list], ')' ;
record modifier = 'μ', '(', expression, ',', record modification, { ',', record modification }, ')' ;
record modification = identifier, '↦', expression ;

9.3.13 Apply Expressions

apply = expression, '(', [expression list], ')' ;
field select = expression, '.', identifier ;
function type instantiation = name, '[', type, { ',', type }, ']' ;

9.3.14 Lambda Expression

lambda expression = 'λ', type bind list, '·', expression ;

9.3.15 Is Expressions

is expression = is defined type expression | is basic type expression ;
is defined type expression = identifier, '(', expression, ')' ;
is basic type expression = is basic type, '(', expression, ')' ;
is basic type = '*is*-', ('$\mathbb{B}$' | '$\mathbb{N}$' | '$\mathbb{N}_1$' | '$\mathbb{Z}$' | '$\mathbb{Q}$' | '$\mathbb{R}$' | 'char' | 'token') ;

9.3.16 Names

name = identifier ;
name list = name, { ',', name } ;
old name = identifier, '$\overleftarrow{\ }$' ;

NOTE — An old name such as $identifier\overleftarrow{\ }$ can also be written as $\overleftarrow{identifier}$.

9.4 State Designators

state designator = name
| field reference
| map or sequence reference ;
field reference = state designator, '.', identifier ;
map or sequence reference = state designator, '(', expression, ')' ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.5 Statements

statement = let statement
| let be statement
| def statement
| block statement
| assign statement
| if statement
| cases statement
| sequence for loop
| set for loop
| index for loop
| while loop
| nondeterministic statement
| call statement
| return statement
| always statement
| trap statement
| recursive trap statement
| exit statement
| identity statement ;

9.5.1 Local Binding Statements

let statement = 'let', local definition, { ',', local definition }, 'in', statement ;
local definition = value definition | function definition ;
let be statement = 'let', bind, ['be', 'st', expression], 'in', statement ;
def statement = 'def', equals definition, { ';', equals definition }, 'in', statement ;
equals definition = pattern bind, '=', (expression | call statement) ;

9.5.2 Block and Assignment Statements

block statement = '(', { dcl statement }, statement, { ';', statement }, ')' ;
dcl statement = 'dcl', assignment definition, ';' ;
assignment definition = identifier, ':', type, [': =', expression | ': =', call statement] ;
assign statement = state designator, ': =', (expression | call statement) ;

9.5.3 Conditional Statements

if statement = 'if', expression, 'then', statement, { elseif statement }, ['else', statement] ;
elseif statement = 'elseif', expression, 'then', statement ;
cases statement = 'cases', expression, ':',
cases statement alternatives,
[',', others statement],
'end' ;
cases statement alternatives = cases statement alternative, { ',', cases statement alternative } ;
cases statement alternative = pattern list, '→', statement ;
others statement = 'others', '→', statement ;

9.5.4 Loop Statements

sequence for loop = 'for', pattern bind, 'in', ['reverse'], expression,
'do', statement ;
set for loop = 'for', 'all', pattern, '∈', expression,
'do', statement ;
index for loop = 'for', identifier, '=', expression, 'to', expression, ['by', expression],
'do', statement ;
while loop = 'while', expression, 'do', statement ;

9.5.5 Nondeterministic Statement

nondeterministic statement = '||', '(', statement, { ',', statement }, ')' ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.5.6 Call and Return Statements

call statement = name, '(', [expression list], ')', ['using', state designator] ;
return statement = 'return', [expression] ;

9.5.7 Exception Handling Statements

always statement = 'always', statement, 'in', statement ;
trap statement = 'trap', pattern bind, 'with', statement, 'in', statement ;
recursive trap statement = 'tixe', traps, 'in', statement ;
traps = '{', trap, { ',', trap }, '}' ;
trap = pattern bind, '↦', statement ;
exit statement = 'exit', [expression] ;

9.5.8 Identity Statement

identity statement = 'skip' ;

9.6 Patterns and Bindings

9.6.1 Patterns

pattern = pattern identifier
| match value
| set enum pattern
| set union pattern
| seq enum pattern
| seq conc pattern
| tuple pattern
| record pattern ;
pattern identifier = identifier | '-' ;
match value = '(', expression, ')' | symbolic literal ;
set enum pattern = '{', pattern list, '}' ;
set union pattern = pattern, '∪', pattern ;
seq enum pattern = '[', pattern list, ']' ;
seq conc pattern = pattern, '⌢', pattern ;
tuple pattern = '*mk-*', '(', pattern, ',', pattern list, ')' ;
record pattern = name, '(', [pattern list], ')' ;
pattern list = pattern, { ',', pattern } ;

9.6.2 Bindings

pattern bind = pattern | bind ;
bind = set bind | type bind ;
set bind = pattern, '∈', expression ;
type bind = pattern, ':', type ;
bind list = multiple bind, { ',', multiple bind } ;
multiple bind = multiple set bind | multiple type bind ;
multiple set bind = pattern list, '∈', expression ;
multiple type bind = pattern list, ':', type ;
type bind list = type bind, { ',', type bind } ;

9.7 Lexical Specification

9.7.1 General

The text of a VDM-SL document in the mathematical concrete representation may be considered at three levels: as marks on paper, as a sequence of characters and as a sequence of symbols.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.7.2 Characters

The transformation from marks on paper to a sequence of characters is not defined here, but must be unambiguous. The usual English orthographic conventions for interpreting printed text are assumed (division into pages and lines, direction of reading, ignoring of page furniture such as headings and page numbers, identification of printed or written characters, and so on). Sequences of non-VDM-SL text may be interspersed with VDM-SL text using any convention of presentation that allows the VDM-SL text to be unambiguously identified.

The character set is shown in table 1, with the forms of characters used in this part of ISO/IEC 13817; these characters must all be unambiguously identifiable.

```
character = plain letter
          | keyword letter
          | distinguished letter
          | Greek letter
          | digit
          | delimiter character
          | other character
          | separator ;
plain letter = (* see table 1 *) ;
keyword letter = (* see table 1 *) ;
distinguished letter = (* see table 1 *) ;
Greek letter = (* see table 1 *) ;
digit = (* see table 1 *) ;
delimiter character = (* see table 1 *) ;
other character = (* see table 1 *) ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 1 — Character set

plain letter:

a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z
A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

keyword letter:

a	b	c	d	e	f	g	h	i	j	k	l	m
n	o	p	q	r	s	t	u	v	w	x	y	z

distinguished letter:

A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	B	C	D	E	F	G	H	I	J	K	L	M
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

Greek letter:

α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ
ν	ξ	o	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω
A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M
N	Ξ	O	Π	P	Σ	T	Υ	Φ	X	Ψ	Ω

digit:

0	1	2	3	4	5	6	7	8	9

delimiter character:

,	:	;	=	(	)	|	$\overset{-}{\leftarrow}$	[	]	{	}
+	/	<	>	$\leq$	$\geq$	$\neq$		$\forall$	$\exists$	.	*
+	$\overset{m}{\longleftrightarrow}$	$\overset{m}{\rightarrow}$	$\rightarrow$	$\overset{t}{\rightarrow}$	$\overset{o}{\rightarrow}$	$\|\|$	$\Rightarrow$	$\Leftrightarrow$	$\in$	$\mapsto$	$\bigcap$
$\bigcup$	$^{-1}$	$\lhd$	$\rhd$	$\trianglelefteq$	$\trianglerighteq$	ι	λ	μ	$\cdot$	$\times$	$\triangleq$
$\neg$	$\cap$	$\cup$	$\uplus$	$\uparrow$	$\subset$	$\subseteq$	$\mathcal{F}$	$\notin$	$\curvearrowright$	$\dagger$	$\circ$
$\wedge$	$\vee$	$\mathbb{B}$	$\mathbb{N}$	$\mathbb{N}_1$	$\mathbb{Z}$	$\mathbb{Q}$	$\mathbb{R}$				

other character:

-	‘	’	“	”	@	!	′

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.7.3 Symbols

The following kinds of symbols exist: keywords, delimiters, identifiers, symbolic literals, and comments. The transformation from characters to symbols is given by the following rules; these use the same notation as the syntax definition but differ in meaning in that no separators may appear between adjacent terminals. Where ambiguity is possible otherwise, two consecutive symbols must be separated by a separator.

keyword = 'abs' | 'all' | 'always' | 'be' | 'by' | 'card' | 'cases' | 'char'
| 'compose' | 'conc' | 'dcl' | 'def' | 'div' | 'do'
| 'dom' | 'elems' | 'else' | 'elseif' | 'end' | 'errs' | 'exit'
| 'ext' | 'false' | 'floor' | 'for' | 'functions' | 'hd'
| 'if' | 'in' | 'inds' | 'init' | 'inv' | 'len' | 'let'
| 'merge' | 'mod' | 'nil' | 'of' | 'operations' | 'others'
| 'post' | 'pre' | 'rd' | 'rem' | 'return' | 'reverse'
| 'rng' | 'skip' | 'st' | 'state' | 'then' | 'tixe' | 'token'
| 'tl' | 'to' | 'trap' | 'true' | 'types' | 'using' | 'values'
| 'while' | 'with' | 'wr' | '-set' ;

delimiter = delimiter character | compound delimiter | comment ;

compound delimiter = '::' | ':=' | '...' | '∃!' ;

separator = newline | white space ;

NOTE — There are two separators: with line break (newline) and without line break (white space) — these have no graphic form.

identifier = (plain letter | Greek letter), { (plain letter | Greek letter) | digit | '/' | '-' } ;

NOTE — Because of the rules given above, any component of an identifier, except the first mark, may be written as a subscript.

symbolic literal = numeric literal
| boolean literal
| nil literal
| character literal
| text literal
| quote literal ;

numeral = digit, { digit } ;

numeric literal = numeral, ['.', digit, { digit }], [exponent] ;

exponent = '×10↑', ['+' | '−'], numeral ;

NOTE — $\times 10\uparrow n$ can be written $\times 10^n$.

boolean literal = 'true' | 'false' ;

nil literal = 'nil' ;

character literal = ''', character − newline, ''' ;

text literal = '"', { '""' | character − ('"' | newline) }, '"' ;

quote literal = distinguished letter, { distinguished letter | digit | '-' | '/' } ;

comment = '--', { character − newline }, newline ;

9.8 Operator Precedence

The precedence ordering for operators in the concrete syntax is defined using a two-level approach: operators are divided into families, and an upper-level precedence ordering, $>$, is given on the families, such that if families F_1 and F_2 satisfy

$F_1 > F_2$

then every operator in the family F_1 is of higher precedence than every operator in the family F_2.

The relative precedences of the operators within families is determined by considering type information, and this is used to resolve ambiguity. The type constructors are treated separately, and are not placed in a precedence ordering with the other operators.

There are six families of operators, namely Combinators, Applicators, Evaluators, Relations, Connectives and Constructors.

Combinators: operations that allow function and mapping values to be combined, and function, mapping and numeric values to be iterated.

Applicators: function application, field selection, sequence indexing, etc.

Evaluators: operators that are non-predicates.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Relations: operators that are relations.

Connectives: the logical connectives.

Constructors: operators that are used, implicitly or explicitly, in the construction of expressions; e.g. if-then-elsif-else, '$\mapsto$', '...', etc.

The precedence ordering on the families is:

combinators > applicators > evaluators > relations > connectives > constructors

9.8.1 The Family of Combinators

These combinators have the highest family priority.

combinator = iterate | composition ;
iterate = '$\uparrow$' ;
composition = '$\circ$' ;

precedence level	combinator
1	$\circ$
2	$\uparrow$

9.8.2 The Family of Applicators

All applicators have equal precedence.

applicator = subsequence
| apply
| function type instantiation
| field select ;
subsequence = expression, '(', expression, ',', '...', ',', expression, ')' ;
apply = expression, '(', [expression list], ')' ;
function type instantiation = name, '[', type, { ',', type }, ']' ;
field select = expression, '.', identifier ;

9.8.3 The Family of Evaluators

The family of evaluators is divided into nine groups, according to the type of expression they are used in.

evaluator = arithmetic prefix operator
| set prefix operator
| sequence prefix operator
| map prefix operator
| map inverse
| arithmetic infix operator
| set infix operator
| sequence infix operator
| map infix operator ;
arithmetic prefix operator = '+' | '−' | 'abs' | 'floor' ;
set prefix operator = 'card' | '$\mathcal{F}$' | '$\bigcup$' | '$\bigcap$' ;
sequence prefix operator = 'hd' | 'tl' | 'len' | 'inds' | 'elems' | 'conc' ;
map inverse = expression, '$^{-1}$' ;
map prefix operator = 'dom' | 'rng' | 'merge' ;
arithmetic infix operator = '+' | '−' | '×' | '/' | 'rem' | 'mod' | 'div' ;
set infix operator = '$\cup$' | '$\cap$' | '\' ;
sequence infix operator = '$\frown$' ;
map infix operator = '$\uplus$' | '†' | '$\lhd$' | '$\unlhd$' | '$\rhd$' | '$\unrhd$' ;

The precedence ordering follows a pattern of analogous operators. The family is defined in the following table.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

precedence level	arithmetic	set	map	sequence
1	$+$ $-$	$\cup$ $\setminus$	$\uplus$ $\dagger$	$\curvearrowright$
2	$\times$ /	$\cap$		
	'rem'			
	'mod'			
	'div'			
3			$^{-1}$	
4			$\lhd$ $\lhd\!\!\!\!-$	
			$\rhd$ $-\!\!\!\!\rhd$	
5	(unary) $+$	'card'	'dom'	'len'
	(unary) $-$	$\mathcal{F}$	'rng'	'elems'
	'abs'	$\bigcap$	'merge'	'hd' 'tl'
	'floor'	$\bigcup$		'conc'
				'inds'

9.8.4 The Family of Relations

This family includes all the relational operators whose result is of type $\mathbb{B}$.

relation = relational infix operator | set relational operator ;
relational infix operator = '$=$' | '$\neq$' | '$<$' | '$\leq$' | '$>$' | '$\geq$' ;
set relational operator = '$\subseteq$' | '$\subset$' | '$\in$' | '$\notin$' ;

precedence level	relation	
1	$\leq$	$<$
	$\geq$	$>$
	$=$	$\neq$
	$\subseteq$	$\subset$
	$\in$	$\notin$

All operators in the Relations family have equal precedence.

9.8.5 The Family of Connectives

This family includes all the logical operators whose result is of type $\mathbb{B}$.

connective = logical prefix operator | logical infix operator ;
logical prefix operator = '$\neg$' ;
logical infix operator = '$\wedge$' | '$\vee$' | '$\Rightarrow$' | '$\Leftrightarrow$' ;

precedence level	connective
1	$\Leftrightarrow$
2	$\Rightarrow$
3	$\vee$
4	$\wedge$
5	$\neg$

9.8.6 The Family of Constructors

This family includes all the operators used to construct a value. Their priority is given either by brackets, which are an implicit part of the operator, or by the syntax.

9.8.7 Grouping

The grouping of operands of operators are as follows:

Combinators: right grouping.

Applicators: left grouping.

Connectives: the '$\Rightarrow$' operator has right grouping. The other operators are associative and therefore right and left grouping are equivalent.

Evaluators: left grouping.

Relations: left grouping.

Constructors: no grouping, as it has no meaning.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

9.8.8 The Type Operators

Type operators have their own separate precedence ordering, as follows:

a) Function and map types: $\rightarrow \xrightarrow{t} \longleftrightarrow, \xrightarrow{m}$ (right grouping).

b) Union type: | (left grouping).

c) Other binary type operators: × (no grouping).

d) Unary type operators: *, $^+$, '-set'.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

10 The Interchange Concrete Syntax

10.1 Introduction

The interchange concrete syntax is an alternative to the mathematical concrete syntax of clause 9, based on the coded character set of ISO 8859–1 – 1987. The two concrete syntaxes differ only in the representations of terminal symbols, with a very few exceptions noted below. The interchange syntax is defined by giving the representation for each symbol of the mathematical syntax, and by defining the lexis, i.e. the rules for representing a sequence of symbols as a sequence of characters from the character set.

10.2 Lexis

The character set consists of the following subset of the coded character set defined in ISO 8859–1:

— The format effectors horizontal tabulation, line feed, vertical tabulation, form feed and carriage return (0/9 to 0/13).

— The character space (2/0).

— All the graphic characters (2/1 to 7/14), with allocations from BS 4730:1985 for the alternative or unallocated combinations.

The separators of the mathematical concrete syntax are represented as follows:

— The *white space* separator is represented by any sequence of one or more space characters and/or horizontal tabulation characters, in any order.

— The *newline* separator is represented by any sequence of the following character combinations

a) a space character

b) a horizontal tabulation character

c) a carriage return character followed by a line feed, vertical tabulation, or form feed character containing at least one of the third type. [See note 1]

10.3 Symbols

Table 2 below shows the equivalent interchange syntax representation of each symbol of the mathematical concrete syntax.

Table 2: Interchange syntax: representation of symbols

mathematical syntax	interchange syntax
(1) Identifiers	
plain letter	the corresponding letter: `A` to `Z`, `a` to `z` (capital letter A to capital letter Z, 4/1 to 5/10, and small letter a to small letter z, 6/1 to 7/10)
digit	the corresponding digit: `0` to `9` (digit zero to digit nine, 3/0 to 3/9)
Greek letter	`#` (number sign, 2/3), followed by the corresponding letter as shown below

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 2: (continued)

mathematical syntax	interchange syntax
α β γ δ ϵ ζ η θ ι κ λ μ a b g d e z h q i k l m ν ξ o π ρ σ τ υ ϕ χ ψ ω n x o p r s t u f c y w A B Γ Δ E Z H Θ I K Λ M A B G D E Z H Q I K L M N Ξ O Π P Σ T Υ Φ X Ψ Ω N X O P R S T U F C Y W	
prime: ′	` (grave accent, 6/0)
hyphen: -	_ (low line, underline, 5/15)
identifier with hook	identifier followed by ~ (tilde, overline, 7/14) If an identifier coincides with a keyword, then it is preceded by $ (dollar sign, 2/4)
(2) Keywords	
keyword letters a to z	corresponding small letters
(3) Quote literals	
digit	the corresponding digit: 0 to 9 (digit zero to digit nine, 3/0 to 3/9)
prime	` (grave accent, 6/0)
hyphen	_ (lowline, underline, 5/15)
distinguished letters A to Z A to Z	corresponding capital and lower-case letters
	The whole quote literal preceded by < (less than sign, 3/12) and followed by > (greater than sign, 3/14).
(4) Numerals and real literals	
digits	as above
decimal point .	. (full stop, 2/14)
exponent sign, $\times 10\uparrow$	E (capital letter E, 4/5)
(5) Character and string literals	
character quotes ‘ ’	’ (apostrophe, 2/7)
string quotes “ ”	" (quotation mark, 2/2)
graphic character	only graphic characters from the ISO 646 character set are allowed in character and string literals. The character " (quotation mark, 2/2) is represented in a string literal by two consecutive quotation marks.
(6) Comments	
from -- to next newline	from -- (minus sign, minus sign) to next carriage return
annotation	from `annotation` to next `end annotation`
(7) Other symbols	
,	, (comma, 2/12)
:	: (colon, 3/10)
;	; (semicolon, 3/11)
=	= (equals sign, 3/13)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 2: (continued)

mathematical syntax	interchange syntax	
)	) (left parenthesis, 2/8)	
(	((right parenthesis, 2/9)	
|	| (vertical line, 7/12)	
−	- (minus sign, 2/13)	
[	[(left square bracket, 5/11)	
]	] (right square bracket, 5/13)	
@	@ (commercial at, 4/0)	
{	{ (left curly bracket, 7/11)	
}	} (right curly bracket, 7/13)	
+	+ (plus sign, 2/11)	
/	/ (solidus, 2/15)	
<	< (less than sign, 3/12)	
>	> (greater than sign, 3/14)	
.	. (full stop, 2/14)	
·	& (ampersand, 2/6)	
×	* (asterisk, 2/10)	
′	` (grave accent, 6/0)	
≤	<=	
≥	>=	
≠	<>	
$\xrightarrow{o}$	==>	
→	->	
$\xrightarrow{t}$	-t>	
⇒	=>	
⇔	<=>	
↦	|->	
$\triangleq$	==	
‖	||	
: =	:=	
. . .	. . .	
::	::	
↑	**	
†	++	
◁	<:	
▷	:>	
⩤	<-:	
⩥	:->	
⊎	munion	
⊂	psubset	
⊆	subset	
↷	^	
$\bigcap$	dinter	
$\bigcup$	dunion	
$^{-1}$	inverse	[see note 3]
$\mathcal{F}$	power	
*	seq of ..	[see note 2]
+	seq1 of ..	[see note 2]
$\xleftrightarrow{m}$	inmap .. to ..	[see note 2]
$\xrightarrow{m}$	map .. to ..	[see note 2]
ι	iota	
λ	lambda	

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Table 2: (continued)

mathematical syntax	interchange syntax
μ	`mu`
$\mathbb{B}$	`bool`
$\mathbb{N}$	`nat`
$\mathbb{N}_1$	`nat1`
$\mathbb{Z}$	`int`
$\mathbb{Q}$	`rat`
$\mathbb{R}$	`real`
$\neg$	`not`
$\cap$	`inter`
$\cup$	`union`
$\in$	`in set`
$\notin$	`not in set`
$\circ$	`comp`
$\wedge$	`and`
$\vee$	`or`
$\forall$	`forall`
$\exists$	`exists`
$\exists!$	`exists1`

NOTES

1 This is intended to reflect the implication of paragraph 4.1.2.2 of ISO 646.

2 The representations of the sequence types T^* and T^+ are `seq of T` and `seq1 of T` respectively; and of the maps $T1 \xrightarrow{m} T2$ and $T1 \overset{m}{\longleftrightarrow} T2$ are `map T1 to T2` and `inmap T1 to T2` respectively.

3 The representation of M^{-1} is `inverse M`.

4 The representation of T-set is `set of T`.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11 The Outer Abstract Syntax

11.1 Document

$Document = DefinitionBlock^+$

11.2 Definitions

$$
\begin{aligned}
DefinitionBlock = {} & TypeDefinitions \\
| {} & StateDef \\
| {} & ValueDefinitions \\
| {} & FunctionDefinitions \\
| {} & OperationDefinitions
\end{aligned}
$$

11.2.1 Type Definitions

$TypeDefinitions :: typedefs : TypeDef^+$

$TypeDef = UnTaggedTypeDef \mid TaggedTypeDef$

$$
\begin{aligned}
UnTaggedTypeDef :: {} & id : Id \\
& shape : Type \\
& typeinv : [Invariant]
\end{aligned}
$$

$$
\begin{aligned}
TaggedTypeDef :: {} & id : Id \\
& fields : FieldList \\
& typeinv : [Invariant]
\end{aligned}
$$

$$
\begin{aligned}
Type = {} & BracketedType \\
| {} & BasicType \\
| {} & QuoteType \\
| {} & CompositeType \\
| {} & UnionType \\
| {} & ProductType \\
| {} & OptionalType \\
| {} & SetType \\
| {} & SeqType \\
| {} & MapType \\
| {} & FnType \\
| {} & TypeName \\
| {} & TypeVar
\end{aligned}
$$

$BracketedType :: type : Type$

$BasicType = \text{BOOLEAN} \mid \text{NAT} \mid \text{NATONE} \mid \text{INTEGER} \mid \text{RAT} \mid \text{REAL} \mid \text{CHAR} \mid \text{TOKEN}$

$QuoteType :: lit : QuoteLit$

$$
\begin{aligned}
CompositeType :: {} & id : Id \\
& fields : FieldList
\end{aligned}
$$

$FieldList = Field^*$

$$
\begin{aligned}
Field :: {} & sel : [Id] \\
& type : Type
\end{aligned}
$$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$UnionType :: summands : Type^+$

inv $mk\text{-}UnionType(ts) \triangleq \mathsf{len}\ ts \geq 2$

$ProductType :: factors : Type^+$

inv $mk\text{-}ProductType(ts) \triangleq \mathsf{len}\ ts \geq 2$

$OptionalType :: type : Type$

$SetType :: elemtp : Type$

$SeqType = Seq0\,Type \mid Seq1\,Type$

$Seq0\,Type :: elemtp : Type$

$Seq1\,Type :: elemtp : Type$

$MapType = GeneralMapType \mid InjectiveMapType$

$GeneralMapType :: mapdom : Type$
$\qquad\qquad\qquad\qquad maprng : Type$

$InjectiveMapType :: mapdom : Type$
$\qquad\qquad\qquad\qquad\quad maprng : Type$

$FnType = PartialFnType \mid TotalFnType$

$PartialFnType :: fndom : DiscretionaryType$
$\qquad\qquad\qquad\quad fnrng : Type$

$TotalFnType :: fndom : DiscretionaryType$
$\qquad\qquad\qquad\quad fnrng : Type$

$DiscretionaryType = Type \mid \text{UNITTYPE}$

$TypeName = Name$

$TypeVar = TypeVarId$

$TypeVarId :: Id$

11.2.2 State Definition

$StateDef :: stid : Id$
$\qquad\qquad fields : FieldList$
$\qquad\qquad stinv : [Invariant]$
$\qquad\qquad stinit : [Initialization]$

$Invariant = InvInitFn$

$Initialization = InvInitFn$

$InvInitFn :: pat : Pattern$
$\qquad\qquad\quad expr : Expr$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.2.3 Value Definitions

$ValueDefinitions :: valdefs : ValueDef^+$

$$\begin{array}{l} ValueDef :: pat \ : Pattern \\ \qquad\qquad\ type : [Type] \\ \qquad\qquad\ val \ : Expr \end{array}$$

11.2.4 Function Definitions

$FunctionDefinitions :: funcdefs : FunctionDef^+$

$FunctionDef = ExplFnDef \mid ImplFnDef$

$$\begin{array}{l} ExplFnDef :: id \ : Id \\ \qquad\qquad\ tpparms \ : OTypeVarList \\ \qquad\qquad\ type \ : FnType \\ \qquad\qquad\ idrepeated : Id \\ \qquad\qquad\ parms \ : ParametersList \\ \qquad\qquad\ body \ : Expr \\ \qquad\qquad\ fnpre \ : [Expr] \end{array}$$

$$\begin{array}{l} ImplFnDef :: id \ : Id \\ \qquad\qquad\ tpparms \ : OTypeVarList \\ \qquad\qquad\ partps \ : ParameterTypes \\ \qquad\qquad\ residtype : IdType \\ \qquad\qquad\ fnpre \ : [Expr] \\ \qquad\qquad\ fnpost \ : Expr \end{array}$$

$OTypeVarList = TypeVar^*$

$$\begin{array}{l} IdType :: id \ : Id \\ \qquad\ type : Type \end{array}$$

$ParameterTypes = OPatTypePairList$

$OPatTypePairList = PatTypePair^*$

$$\begin{array}{l} PatTypePair :: pat \ : Patlist \\ \qquad\qquad\ type : Type \end{array}$$

$ParametersList = Parameters^+$

$Parameters = OPatList$

11.2.5 Operation Definitions

$OperationDefinitions :: opdefs : OperationDef^+$

$OperationDef = ExplOprtDef \mid ImplOprtDef$

$$\begin{array}{l} ExplOprtDef :: id \ : Id \\ \qquad\qquad\ optype \ : OpType \\ \qquad\qquad\ idrepeated : Id \\ \qquad\qquad\ params \ : Parameters \\ \qquad\qquad\ body \ : Stmt \\ \qquad\qquad\ oppre \ : [Expr] \end{array}$$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$ImplOprtDef :: id : Id$
$params : ParameterTypes$
$residtype : [IdType]$
$opext : OExternals$
$oppre : [Expr]$
$oppost : Expr$
$excps : OExceptions$

$OpType :: opdom : DiscretionaryType$
$oprng : DiscretionaryType$

$OExternals = VarInf^*$

$VarInf :: mode : Mode$
$vars : NameList$
$type : [Type]$

$Mode = \text{READ} \mid \text{READWRITE}$

$OExceptions = Error^*$

$Error :: id : Id$
$cond : Expr$
$action : Expr$

11.3 Expressions

$ExprList = Expr^+$

$OExprList = Expr^*$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$$
\begin{aligned}
Expr = \; & BracketedExpr \\
| \; & LetExpr \\
| \; & LetBeSTExpr \\
| \; & DefExpr \\
| \; & IfExpr \\
| \; & CasesExpr \\
| \; & UnaryExpr \\
| \; & BinaryExpr \\
| \; & QuantExpr \\
| \; & IotaExpr \\
| \; & SetEnumeration \\
| \; & SetComprehension \\
| \; & SetRange \\
| \; & SeqEnumeration \\
| \; & SeqComprehension \\
| \; & SubSequence \\
| \; & MapEnumeration \\
| \; & MapComprehension \\
| \; & TupleConstructor \\
| \; & RecordConstructor \\
| \; & RecordModifier \\
| \; & Apply \\
| \; & FieldSelect \\
| \; & FctTypeInst \\
| \; & Lambda \\
| \; & IsExpr \\
| \; & Name \\
| \; & OldName \\
| \; & Literal
\end{aligned}
$$

11.3.1 Bracketed Expressions

$BracketedExpr :: expr : Expr$

11.3.2 Local Binding Expressions

$$
\begin{aligned}
LetExpr :: \; & localdefs : LocalDef^{+} \\
& body \quad : Expr
\end{aligned}
$$

$$
\begin{aligned}
LetBeSTExpr :: \; & bind : Bind \\
& cond : [Expr] \\
& body : Expr
\end{aligned}
$$

$$
\begin{aligned}
DefExpr :: \; & defs \; : DefBind^{+} \\
& body : Expr
\end{aligned}
$$

$$
\begin{aligned}
DefBind :: \; & lhs \; : PatternBind \\
& rhs : Expr
\end{aligned}
$$

11.3.3 Conditional Expressions

$$
\begin{aligned}
IfExpr :: \; & test \quad : Expr \\
& cons \quad : Expr \\
& elsifaltn : ElsifExpr^{*} \\
& altn \quad : Expr
\end{aligned}
$$

$$
\begin{aligned}
ElsifExpr :: \; & test \; : Expr \\
& cons : Expr
\end{aligned}
$$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

CasesExpr :: *sel* : *Expr*
altns : *CaseExprAlternatives*
other : [*OthersExpr*]

CaseExprAlternatives = *CaseAltn*$^+$

CaseAltn :: *match* : *PatList*
body : *Expr*

OthersExpr = *Expr*

11.3.4 Unary Expressions

UnaryExpr = *PrefixExpr* | *MapInverseExpr*

PrefixExpr :: *op* : *UnaryOp*
expr : *Expr*

UnaryOp = NUMPLUS
| NUMMINUS
| NUMABS
| FLOOR
| NOT
| SETCARD
| SETPOWER
| SETDISTRUNION
| SETDISTRINTERSECT
| SEQHEAD
| SEQTAIL
| SEQLEN
| SEQELEMS
| SEQINDICES
| SEQDISTRCONC
| MAPDOM
| MAPRNG
| MAPDISTRMERGE

MapInverseExpr :: *operand* : *Expr*

11.3.5 Binary Expressions

BinaryExpr :: *left* : *Expr*
op : *BinaryOp*
right : *Expr*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

BinaryOp = NUMPLUS
| NUMMINUS
| NUMMULT
| NUMDIV
| INTDIV
| NUMREM
| NUMMOD
| NUMLT
| NUMLE
| NUMGT
| NUMGE
| EQ
| NE
| OR
| AND
| IMPLY
| EQUIV
| INSET
| NOTINSET
| SUBSET
| PROPERSUBSET
| SETUNION
| SETDIFFERENCE
| SETINTERSECT
| SEQCONC
| MAPORSEQMOD
| MAPMERGE
| MAPDOMRESTRTO
| MAPDOMRESTRBY
| MAPRNGRESTRTO
| MAPRNGRESTRBY
| COMPOSE
| ITERATE

11.3.6 Quantified Expressions

QuantExpr = *AllExpr*
| *ExistsExpr*
| *ExistsUniqueExpr*

AllExpr :: *bind* : *BindList*
pred : *Expr*

ExistsExpr :: *bind* : *BindList*
pred : *Expr*

ExistsUniqueExpr :: *bind* : *Bind*
pred : *Expr*

11.3.7 Iota Expression

IotaExpr :: *bind* : *Bind*
pred : *Expr*

11.3.8 Set Expressions

SetEnumeration :: *els* : *OExprList*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

SetComprehension :: *elem* : *Expr*
bind : *BindList*
pred : [*Expr*]

SetRange :: *lb* : *Expr*
ub : *Expr*

11.3.9 Sequence Expressions

SeqEnumeration :: *els* : *OExprList*

SeqComprehension :: *elem* : *Expr*
bind : *SetBind*
pred : [*Expr*]

SubSequence :: *sequence* : *Expr*
frompos : *Expr*
topos : *Expr*

11.3.10 Map Expressions

MapEnumeration :: *els* : $Maplet^*$

Maplet :: *mapdom* : *Expr*
maprng : *Expr*

MapComprehension :: *elem* : *Maplet*
bind : *BindList*
pred : [*Expr*]

11.3.11 Tuple Constructor Expression

TupleConstructor :: *fields* : *ExprList*

inv *mk-TupleConstructor*(*fs*) $\triangleq$ len *fs* ≥ 2

11.3.12 Record Expressions

RecordConstructor :: *tag* : *Id*
fields : *OExprList*

RecordModifier :: *rec* : *Expr*
modifiers : $RecordModification^+$

RecordModification :: *field* : *Id*
new : *Expr*

11.3.13 Apply Expressions

Apply :: *fct* : *Expr*
arg: *OExprList*

FieldSelect :: *record* : *Expr*
field : *Id*

FctTypeInst :: *polyfct* : *Name*
inst : $Type^+$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.3.14 Lambda Expression

Lambda :: *parms* : *TypeBindList*
body : *Expr*

11.3.15 Is Expressions

IsExpr = *IsDefTypeExpr* | *IsBasicTypeExpr*

IsDefTypeExpr :: *deftype* : *Id*
arg : *Expr*

IsBasicTypeExpr :: *type* : *IsBasicType*
arg : *Expr*

IsBasicType = BOOLEAN | NAT | NATONE | INTEGER | RAT | REAL | CHAR | TOKEN

11.3.16 Names

Name :: *name* : *Id*

NameList = *Name*$^+$

OldName :: *name* : *Id*

11.4 State Designators

StateDesignator = *Name*
| *FieldRef*
| *MapOrSeqRef*

FieldRef :: *var* : *StateDesignator*
sel : *Id*

MapOrSeqRef :: *var* : *StateDesignator*
arg : *Expr*

11.5 Statements

Stmt = *LetStmt*
| *LetBeSTStmt*
| *DefStmt*
| *BlockStmt*
| *AssignStmt*
| *IfStmt*
| *CasesStmt*
| *SeqForLoop*
| *SetForLoop*
| *IndexForLoop*
| *WhileLoop*
| *NonDetStmt*
| *Call*
| *ReturnStmt*
| *AlwaysStmt*
| *TrapStmt*
| *RecTrapStmt*
| *ExitStmt*
| *IdentStmt*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.5.1 Local Binding Statements

$LetStmt :: letdefs : LocalDef^+$
$\quad\quad\quad

11.5.4 Loop Statements

$SeqForLoop :: cv : PatternBind$
$\quad dirn : [\text{REVERSE}]$
$\quad forseq : Expr$
$\quad body : Stmt$

$SetForLoop :: cv : Pattern$
$\quad forset : Expr$
$\quad body : Stmt$

$IndexForLoop :: cv : Id$
$\quad lb : Expr$
$\quad ub : Expr$
$\quad step : [Expr]$
$\quad body : Stmt$

$WhileLoop :: test : Expr$
$\quad body : Stmt$

11.5.5 Nondeterministic Statement

$NonDetStmt :: stmts : Stmt^+$

11.5.6 Call and Return Statements

$Call :: oprt : Name$
$\quad args : OExprList$
$\quad callst : [StateDesignator]$

$ReturnStmt :: expr : [Expr]$

11.5.7 Exception Handling Statements

$AlwaysStmt :: alwpost : Stmt$
$\quad body : BlockStmt$

$TrapStmt :: pat : PatternBind$
$\quad trappost : Stmt$
$\quad body : BlockStmt$

$RecTrapStmt :: traps : Traps$
$\quad body : BlockStmt$

$Traps = Trap^+$

$Trap :: match : PatternBind$
$\quad trappost : Stmt$

$ExitStmt :: expr : [Expr]$

11.5.8 Identity Statement

$IdentStmt = \text{SKIP}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

11.6 Patterns and Bindings

11.6.1 Patterns

$Pattern = PatternId$
$\quad | \ MatchVal$
$\quad | \ SetEnumPattern$
$\quad | \ SetUnionPattern$
$\quad | \ SeqEnumPattern$
$\quad | \ SeqConcPattern$
$\quad | \ TuplePattern$
$\quad | \ RecordPattern$

$PatternId :: name : [Id]$

$MatchVal :: val : Expr$

$SetEnumPattern :: els : PatList$

$SetUnionPattern :: lp : Pattern$
$\quad rp : Pattern$

$SeqEnumPattern :: els : PatList$

$SeqConcPattern :: lp : Pattern$
$\quad rp : Pattern$

$TuplePattern :: fields : PatList$

inv $mk\text{-}TuplePattern(pl) \triangleq \mathsf{len}\ pl \geq 2$

$RecordPattern :: id : Id$
$\quad fields : OPatList$

$PatList = Pattern^{+}$

$OPatList = Pattern^{*}$

11.6.2 Bindings

$PatternBind = Pattern \mid Bind$

$Bind = SetBind \mid TypeBind$

$SetBind :: pat : Pattern$
$\quad bindset : Expr$

$TypeBind :: pat : Pattern$
$\quad type : Type$

$BindList = MultBind^{+}$

$MultBind = MultSetBind \mid MultTypeBind$

$MultSetBind :: pats : PatList$
$\quad expr : Expr$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

MultTypeBind :: *pats* : *PatList*
tp : *Type*

TypeBindList = *TypeBind*$^+$

11.7 Lexical Specification

Id = *ValueId* | *PreId* | *PostId* | *InvId* | *InitId* | *MkId* | *IsId*

The identifiers in MCS and OAS are related as described in the following table:

MCS identifier has prefix which is	OAS identifier belongs to
non–reserved	*ValueId*
pre-	*PreId*
post-	*PostId*
inv-	*InvId*
init-	*InitId*
mk-	*MkId*
is-	*IsId*

The different tokens used in the kinds of identifiers should be in a natural relationship to each other. To give an example it is assumed by the semantics that the identifier for a pre-condition for a function is using the same token as the identifier for the function itself (i.e. the identifier value *mk-PreId*(f_1) is using the same token as the name of the corresponding function which will be called *mk-ValueId*(f_1)).

ValueId :: token

PreId :: token

PostId :: token

InvId :: token

InitId :: token

MkId :: token

IsId :: token

Literal = *NumLit*
| *BoolLit*
| *NilLit*
| *CharLit*
| *TextLit*
| *QuoteLit*

NumLit :: *val* : $\mathbb{R}$

BoolLit :: *val* : $\mathbb{B}$

NilLit ::

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

$CharLit :: val$: char

$TextLit :: val$: char*

$QuoteLit :: val$: char*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12 The Syntax Mapping

12.1 Structure and Style of the Definition

The syntax mapping is defined by means of a collection of *explicit VDM-SL functions*, (recursively) calling one another. This style of defining the syntax mapping was chosen (instead of e.g. term rewriting) because it is very similar to the way both the static semantics and the dynamic semantics have been defined. Explicit VDM-SL functions were used because it was felt that an applicative, explicit style of specification suits problems of which the main purpose is to map one sort of construct to another.

12.1.1 Division into Modules

The specification of the syntax mapping has been divided into four modules. This use of 'modules' is necessary in order to control the name space of the specification: the OAS and the CAS mutually contain non-unique identifiers, the structuring mechanism allows to distinguish between them by prefixing these identifiers with the name of the module from which they have been exported and a dot ('.'). For example, a type named t, defined in a module m_1, can be exported by that module and be imported by another module, and can then be used in the importing module as $m_1.t$. The way module constructs are used is such that an intuitive interpretation is sufficient to understand the meaning of the constructs.

The 4 modules are shown in figure 1, with arrows denoting the import relations between the modules.

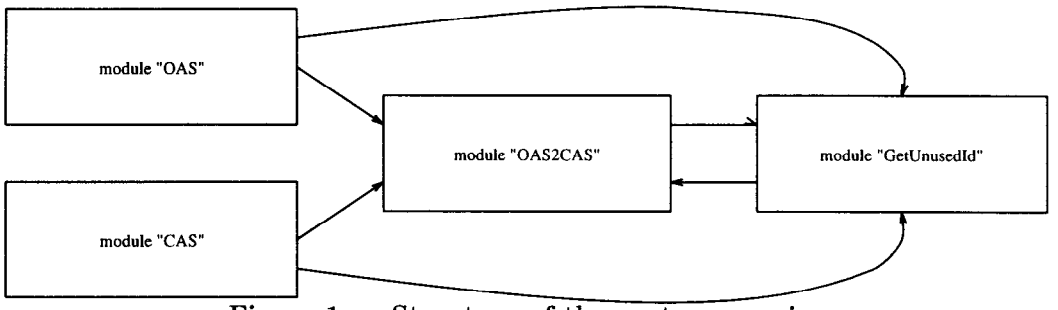


Figure 1 — Structure of the syntax mapping

12.1.1.1 Module "OAS2CAS"

This module is the main module. It contains one function (*OAS2CAS*) which takes a specification in terms of OAS constructs as its input, and produces a specification in terms of CAS constructs as its output.

12.1.1.2 Module "OAS"

This module contains type definitions for the standard abstract syntax (OAS). These type definitions can be found in clause 11.

12.1.1.3 Module "CAS"

This module contains type definitions for the core abstract syntax (CAS). These type definitions can be found in clause 6.

12.1.1.4 Module "GetUnusedId"

This module contains a declaration of the function *GetUnusedId*. The function itself has been left undefined, because its functionality — returning an identifier that is not used at any place in the specification — is rather trivial compared

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

to the space which a complete formal definition of this function would take. See clause 12.1.3.1 for a description of the situations in which the function is used.

12.1.2 Pre-conditions in the VDM-SL Definition of the Syntax Mapping

At some points in the transformation process the syntax of the constructs is changed in such a way that 'information can get lost' if no care is taken. Typical situations occur because in the OAS constructs of the same kind are usually kept in *sequences*, whereas in the CAS *maps* are used. E.g. in the OAS it is possible to have function definitions for functions with the same name, whereas in the CAS function definitions are kept as mappings from their name to their definition, in this way assuring that no two functions can have the same name. Pre-conditions are used in the syntax mapping to avoid such situations.

12.1.3 Transformation of a Document to CAS.Definitions

The syntax mapping will reconfigure a specification written in the OAS to a 'new' specification compatible with the dynamic semantics. If meaning is to be attached to the CAS specification then the syntax mapping must produce a series of maps which are compatible with the dynamic semantics of VDM-SL. Care must be taken to ensure that each map is disjoint as information may be lost when the maps are merged, and that all implicit constructs are redefined explicitly.

The syntax mapping sorts a specification made up of type, state, value, function, and operation definitions and re-organizes it into the following format:

$$
\begin{array}{lll}
\mathit{Definitions}:: & \mathit{typem} & : \mathit{ValueId} \xrightarrow{m} \mathit{TypeDef} \\
& \mathit{expofnm} & : \mathit{Id} \xrightarrow{m} \mathit{ExplPolyFnDef} \\
& \mathit{exmofnm} & : \mathit{Id} \xrightarrow{m} \mathit{ExplFnDef} \\
& \mathit{impofnm} & : \mathit{Id} \xrightarrow{m} \mathit{ImplPolyFnDef} \\
& \mathit{immofnm} & : \mathit{Id} \xrightarrow{m} \mathit{ImplFnDef} \\
& \mathit{valuem} & : \mathit{Pattern} \xrightarrow{m} \mathit{ValDef} \\
& \mathit{explopm} & : \mathit{ValueId} \xrightarrow{m} \mathit{ExplOpDef} \\
& \mathit{implopm} & : \mathit{ValueId} \xrightarrow{m} \mathit{ImplOpDef} \\
& \mathit{state} & : [\mathit{StateDef}]
\end{array}
$$

Information on how each part is transformed now follows.

12.1.3.1 The Introduction of Additional Identifiers

At some places, an identifier is needed which is not already used at any other place in the specification, to avoid name clashes. These identifiers are needed because of the normalized shape of the CAS.

The most obvious example is the transformation of parameter lists. Consider the function *fn*:

$$
\begin{array}{ll}
1.0 & fn_{OAS} : I_1 \times \ldots \times I_n \rightarrow \ldots \\
.1 & fn_{OAS}\,(i_1, \ldots, i_n) \triangleq \\
.2 & \quad \ldots
\end{array}
$$

The equivalent of this function in the CAS is:

$$
\begin{array}{ll}
2.0 & fn_{CAS} : (I_1 \times \ldots \times I_n) \rightarrow \ldots \\
.1 & fn_{CAS}\,(i) \triangleq \\
.2 & \quad \mathsf{let}\ mk\text{-}(i_1, \ldots, i_n) = i\ \mathsf{in} \\
.3 & \quad \ldots
\end{array}
$$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

where i is a newly introduced identifier. As already explained, the function *GetUnusedId* is used to produce such identifiers.

12.1.3.2 The Generation of Quoted Pre- and Post-conditions for Functions

The syntax mapping generates quoted pre- and post-conditions in the following cases.

For all functions *fn* in the specification, defined as:

3.0 $fn : I \to O$
.1 $fn\,(i) \triangleq$
.2 ...
.3 pre $Pre_{fn}(i)$

or

4.0 $fn\,(i : I)\ o : O$
.1 pre $Pre_{fn}(i)$
.2 post ...

a definition for a function called *pre-fn* is generated, which is defined as:

5.0 $pre\text{-}fn : I \to \mathbb{B}$
.1 $pre\text{-}fn\,(i) \triangleq$
.2 $Pre_{fn}(i)$

For all functions *fn* in the specification, defined as:

6.0 $fn\,(i : I)\ o : O$
.1 pre ...
.2 post $Post_{fn}(i, o)$

a definition for a function called *post-fn* is generated, which is defined as:

7.0 $post\text{-}fn : I \times O \to \mathbb{B}$
.1 $post\text{-}fn\,(i, o) \triangleq$
.2 $Post_{fn}(i, o)$

12.1.3.3 The Generation of Quoted Pre- and Post-conditions for Operations

For all operations *op* in the specification, defined as:

8.0 $op : I \xrightarrow{o} O$
.1 $op\,(i)$...
.2 pre $Pre_{op}(i, \sigma)$

and

9.0 $op\,(i : I)\ o : O$
.1 ext rd $\sigma_1 : \Sigma_1$
.2 wr $\sigma_2 : \Sigma_2$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.3 pre $Pre_{op}(i, \sigma)$
.4 post ...

given a state definition

10.0 state Σ of
.1 $\sigma_1 : \Sigma_1$
.2 $\sigma_2 : \Sigma_2$
.3 $\sigma_3 : \Sigma_3$
.4 end

a definition for a function called *pre-op* is generated, which is defined as:

11.0 $pre\text{-}op : I \times \Sigma \to \mathbb{B}$
.1 $pre\text{-}op\,(i, \sigma) \triangleq$
.2 $Pre_{op}(i, \sigma)$

For all operations *op* in the specification, defined as:

12.0 $op\,(i : I)\ o : O$
.1 ext rd $\sigma_1 : \Sigma_1$
.2 wr $\sigma_2 : \Sigma_2$
.3 pre ...
.4 post $Post_{op}(i, o, \overleftarrow{\sigma_1}, \overleftarrow{\sigma_2}, \sigma_1, \sigma_2)$

a definition for a function called *post-op* is generated, which is defined as:

13.0 $post\text{-}op : I \times O \times \Sigma \times \Sigma \to \mathbb{B}$
.1 $post\text{-}op\,(i, o, \sigma', \sigma) \triangleq$
.2 let $mk\text{-}\Sigma(\sigma'_1, \sigma'_2, \sigma'_3) = \sigma'$,
.3 $mk\text{-}\Sigma(\sigma_1, \sigma_2, \sigma_3) = \sigma$ in
.4 $Post_{op}(i, o, \sigma'_1, \sigma'_2, \sigma_1, \sigma_2)$

The references to 'old values' in the post-condition ($\overleftarrow{\sigma_1}$ and $\overleftarrow{\sigma_2}$) which are only meaningful in the context of an operation (remember that *post-op* is a function, not an operation) have been systematically replaced by newly introduced identifiers (σ'_1 and σ'_2, respectively).

12.1.3.4 The Transformation of Expressions

The transformation of expressions differs slightly from the transformations of other constructs, because the former transformation is context sensitive. The context-sensitivity arises from the fact that in some situations identifiers occurring in expressions — although syntactically the same — refer to different objects, whereas the dynamic semantics tries to treat these objects in a more consistent way.

12.1.3.5 Pre-conditions of Operations

In the pre-condition of an implicitly defined operation, the state component a refers to the *old* value of that state component, whereas in the post-condition of that operation a denotes the *new* value of that state component, and $\overleftarrow{a}$ refers to its old value. The syntax mapping, therefore, transforms all occurrences of state components in pre-conditions of operations into their corresponding old values.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.1.3.6 Guards of Error Handlers

A similar situation as in the previous clauseoccurs with guards of error handlers in operation definitions; syntactically, a reference is made in such a guard to the 'new' (undecorated) value of a state component, but in fact a reference should be made to its old (decorated) value. Such a transformation is performed by the syntax mapping.

12.1.3.7 Quoted Post-conditions of Implicit Operations

The construction of the body for the function *post-op*, where *op* is an implicitly defined operation, requires the systematic replacement of references to old values of state components by new identifiers (see also clause 12.1.3.2).

All these transformations are dealt with by adding an extra argument to the transformation functions for expressions, called ρ. This argument is of the type

14.0 $Env = OAS.Id \xrightarrow{m} CAS.Id$

and it contains information on how each identifier encountered while transforming an expression and its subexpressions should be transformed. If an OAS identifier is not in the domain of ρ, then it is simply directly transformed into its CAS counterpart. This means that in all situations other than the ones described in the clauses above ρ is empty ($\{\mapsto\}$).

12.1.3.8 The Transformation of Type Definitions

Type definitions are re-organized into two maps:

- $(CAS.ValueId \xrightarrow{m} CAS.TypeDef)$
- $(CAS.Id \xrightarrow{m} CAS.ExplFnDef)$

The first map has $CAS.ValueId$ as its domain and is the types identifier in the CAS. This is used to reference the type definition after it has been translated into the CAS. $CAS.Id$ is the the type's identifier pre-fixed by '*inv-*' and is used to reference the type's invariant. The invariant is changed to an explicit function definition, via a lambda expression, and this forms the range of the second map. Simplification of types is achieved by removing the distinction between tagged and untagged types by combining a tagged type's field list with its identifier, and reducing these to a composite type. This composite type can then be treated in an identical manner to the *shape* of an untagged type.

12.1.3.9 The Transformation of Value Definitions

All value definitions are extracted from the OAS definition so that the following map can be produced:

- $CAS.Pattern \xrightarrow{m} CAS.ValDef$

The domain holds the pattern of the value which must be unique, and the range contains the type and value expression of the pattern after both type and expression have been translated to the CAS.

12.1.3.10 The Transformation of the State

The syntax mapping must convert the state from its OAS form; $name_{OAS}$, *field* $reference_{OAS}$, and *map or sequence* $reference_{OAS}$, to the CAS representation of; $state_{CAS}$, $fieldlist_{CAS}$, $invariant_{CAS}$, and $initialization_{CAS}$. Transforming the state results in the following maps:

- $CAS.Id \xrightarrow{m} CAS.TypeDef$
- $CAS.Id \xrightarrow{m} CAS.ExplFnDef$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

and an optional definition:

— *StateDef*

The map, $CAS.Id \xrightarrow{m} CAS.TypeDef$, represents the state's identifier in the domain and the type of the state, reconfigured to a tagged type definition, in the range. The type of the state in the OAS comprises of the state's identifier, its fields, and its invariant, making it identical to a tagged type in the OAS. Because of this, it is translated by the same functions used to translate other tagged types, and the result merged with the other type definition maps.

The second map, $CAS.Id \xrightarrow{m} CAS.ExplFnDef$ is the product of merging two maps containing detail on the invariant and initialization of the state. These are translated to the CAS as explicit function definition maps.

The third result, *StateDef*, contains the transformed state definition. As a state definition in the CAS comprises of; *stid* - the name of the state, *tp* - the name of the state's type, and *init* - a function describing possible initial states, it is quite simple to transform. A lambda expression is used to represent the state's initialization function, and is transformed along with the type of the state.

12.1.4 Notational Conventions

The syntax mapping has been defined using standard VDM-SL, the only exception being the use of a provisionary structuring mechanism (see clause 12.1.1).

12.2 Syntaxes and Auxiliary Functions

12.2.1 Module "OAS"

```
module OAS
  exports
15.0  types all

  definitions
    types
.1      — See clause 11 on page 196, in which the Outer Abstract Syntax has been defined.
.2

end OAS
```

12.2.2 Module "CAS"

```
module CAS
  exports
16.0  types all

  definitions
    types
.1      — See clause 6 on page 17, in which the Core Abstract Syntax has been defined.
.2

end CAS
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.2.3 Module "GetUnusedId"

module *GetUnusedId*

imports

17.0 from *OAS*

18.0 types *Id* ,

19.0 from *CAS*

20.0 types *Id* ,

21.0 from *OAS2CAS*

22.0 functions *Id2Id*

exports

23.0 functions *GetUnusedId*

definitions

functions

24.0 $GetUnusedId : () \to OAS`Id \times CAS`Id$

.1 $GetUnusedId\,() \triangleq$

.2 let $id_{OAS} : OAS`Id$ be st $IsUniqueId\,(id_{OAS})$ in

.3 let $id_{CAS} = Id2Id\,(id_{OAS}, \{\mapsto\})$ in

.4 $mk\text{-}(id_{OAS}, id_{CAS})$

annotations

24.0 The function *GetUnusedId* returns an OAS identifier (and its equivalent in the CAS) that is nowhere used in the specification. Even more, it never returns the same value twice.

end annotations;

25.0 $IsUniqueId : OAS`Id \to \mathbb{B}$

.1 $IsUniqueId\,(\text{-}) \triangleq$

.2 undefined

annotations

25.0 The function *IsUniqueId* is a miracle function (and is therefore left undefined). It checks whether an identifier presented as an argument has been used at any place in the specification or whether the identifier has been given as an argument before.

end annotations

end *GetUnusedId*

12.3 The Syntax Mapping Functions

module *OAS2CAS*

imports

26.0 from *OAS* all ,

27.0 from *CAS* all ,

28.0 from *GetUnusedId* all

exports

29.0 functions *Document2Definitions*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

definitions

types

30.0 $Env = OAS`Id \xrightarrow{m} CAS`Id$

annotations

30.0 See clause 12.1.3.4 for a description of the use of type *Env*.

end annotations

12.3.1 Document

functions

```
31.0  Document2Definitions : OAS`Document → CAS`Definitions
 .1   Document2Definitions (dbs) ≜
 .2     let mk-(def-typem, quote-typem)          = DefinitionBlocks2TypeDefMap (dbs),
 .3         mk-(def-expofnm, quote-expofnm)      = DefinitionBlocks2ExplPolyFnDefMap (dbs),
 .4         mk-(def-exmofnm, quote-exmofnm)      = DefinitionBlocks2ExplFnDefMap (dbs),
 .5         mk-(impofnm, quote-impofnm)          = DefinitionBlocks2ImplPolyFnDefMap (dbs),
 .6         mk-(immofnm, quote-immofnm)          = DefinitionBlocks2ImplFnDefMap (dbs),
 .7         valuem                               = DefinitionBlocks2ValDefMap (dbs),
 .8         mk-(explopm, quote-explopm)          = DefinitionBlocks2ExplOpDefMap (dbs, tdm(state.stid)),
 .9         mk-(implopm, quote-implopm)          = DefinitionBlocks2ImplOpDefMap (dbs, tdm(state.stid)),
.10         mk-(state, tdm, quote-state)         = DefinitionBlocks2StateDef (dbs),
.11         typem                                = def-typem ⩊ tdm,
.12         expofnm                              = def-expofnm ⩊ quote-expofnm ⩊ quote-impofnm,
.13         exmofnm                              = def-exmofnm ⩊ quote-exmofnm ⩊ quote-immofnm ⩊
.14                                                quote-explopm ⩊ quote-implopm ⩊ quote-typem ⩊
.15                                                quote-state in
.16       mk-CAS`Definitions (typem, expofnm, exmofnm, impofnm, immofnm,
.17                           valuem, explopm, implopm, state)
```

annotations

.11 The type of the state re-defined for the CAS is combined with the state identifier into a map, this map is merged with the other type definition maps.

.12 Combine all maps with polymorphic functions in their range. (Includes all pre and post-conditions from polymorphic functions as well.)

.13 Combine all maps with monomorphic functions in their range. (Includes pre and post-conditions from monomorphic functions, pre, post, and exceptions from operations, all invariants from type definitions, and the invaraint, and invariant initial function of the state.)

end annotations ;

```
32.0  DefinitionBlocks2TypeDefMap :
 .1       OAS`DefinitionBlock+ →
 .2       (CAS`Id ⟶m CAS`TypeDef) × (CAS`Id ⟶m CAS`ExplFnDef)
 .3    DefinitionBlocks2TypeDefMap (dbs) ≜
 .4      let defs = { TypeDefinitions2TypeDefMap (tds) |
 .5                   mk-OAS`TypeDefinitions (tds) ∈ elems dbs } in
 .6      mk-(merge {tdm | mk-(tdm, -) ∈ defs}, merge {tdm_q | mk-(-, tdm_q) ∈ defs})
 .7    pre let defs = { TypeDefinitions2TypeDefMap (tds) |
 .8                     mk-OAS`TypeDefinitions (tds) ∈ elems dbs } in
 .9      ∀ mk-(tdm, tdm_q) ∈ defs ·
.10          ∀ mk-(tdm', -) ∈ defs \ {mk-(tdm, tdm_q)} · dom tdm ∩ dom tdm' = { }
```

annotations

.7 The pre-condition ensures that all type identifiers are unique.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

end annotations ;

33.0 *DefinitionBlocks2ExplPolyFnDefMap* :
.1 $OAS`DefinitionBlock^+ \rightarrow$
.2 $(CAS`Id \xrightarrow{m} CAS`ExplPolyFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplPolyFnDef)$
.3 *DefinitionBlocks2ExplPolyFnDefMap* $(dbs) \triangleq$
.4 let $defs = \{FunctionDefinitions2ExplPolyFnDefMap\,(fds) \mid$
.5 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.6 $mk\text{-}(\text{merge } \{fdm \mid mk\text{-}(fdm, \text{-}) \in defs\}, \text{merge } \{fdm_q \mid mk\text{-}(\text{-}, fdm_q) \in defs\})$
.7 pre let $defs = \{FunctionDefinitions2ExplPolyFnDefMap\,(fds) \mid$
.8 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.9 $\forall\, mk\text{-}(fdm, fdm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(fdm', \text{-}) \in defs \setminus \{mk\text{-}(fdm, fdm_q)\} \cdot \text{dom } fdm \cap \text{dom } fdm' = \{\}$

annotations

.7 The pre-condition ensures that all explicit polymorphic function identifiers are unique.

end annotations ;

34.0 *DefinitionBlocks2ExplFnDefMap* :
.1 $OAS`DefinitionBlock^+ \rightarrow$
.2 $(CAS`Id \xrightarrow{m} CAS`ExplFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 *DefinitionBlocks2ExplFnDefMap* $(dbs) \triangleq$
.4 let $defs = \{FunctionDefinitions2ExplFnDefMap\,(fds) \mid$
.5 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.6 $mk\text{-}(\text{merge } \{fdm \mid mk\text{-}(fdm, \text{-}) \in defs\}, \text{merge } \{fdm_q \mid mk\text{-}(\text{-}, fdm_q) \in defs\})$
.7 pre let $defs = \{FunctionDefinitions2ExplFnDefMap\,(fds) \mid$
.8 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.9 $\forall\, mk\text{-}(fdm, fdm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(fdm', \text{-}) \in defs \setminus \{mk\text{-}(fdm, fdm_q)\} \cdot \text{dom } fdm \cap \text{dom } fdm' = \{\}$

annotations

.7 The pre-condition ensures that all explicit function identifiers are unique.

end annotations ;

35.0 *DefinitionBlocks2ImplPolyFnDefMap* :
.1 $OAS`DefinitionBlock^+ \rightarrow$
.2 $(CAS`Id \xrightarrow{m} CAS`ImplPolyFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplPolyFnDef)$
.3 *DefinitionBlocks2ImplPolyFnDefMap* $(dbs) \triangleq$
.4 let $defs = \{FunctionDefinitions2ImplPolyFnDefMap\,(fds) \mid$
.5 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.6 $mk\text{-}(\text{merge } \{fdm \mid mk\text{-}(fdm, \text{-}) \in defs\}, \text{merge } \{fdm_q \mid mk\text{-}(\text{-}, fdm_q) \in defs\})$
.7 pre let $defs = \{FunctionDefinitions2ImplPolyFnDefMap\,(fds) \mid$
.8 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.9 $\forall\, mk\text{-}(fdm, fdm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(fdm', \text{-}) \in defs \setminus \{mk\text{-}(fdm, fdm_q)\} \cdot \text{dom } fdm \cap \text{dom } fdm' = \{\}$

annotations

.7 The pre-condition ensures that all implicit polymorphic function identifiers are unique.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

36.0 $DefinitionBlocks2ImplFnDefMap$:
.1 $OAS`DefinitionBlock^+ \to$
.2 $(CAS`Id \xrightarrow{m} CAS`ImplFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $DefinitionBlocks2ImplFnDefMap\,(dbs) \triangleq$
.4 let $defs = \{FunctionDefinitions2ImplFnDefMap\,(fds) \mid$
.5 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.6 $mk\text{-}(\text{merge}\,\{fdm \mid mk\text{-}(fdm, \text{-}) \in defs\}, \text{merge}\,\{fdm_q \mid mk\text{-}(\text{-}, fdm_q) \in defs\})$
.7 pre let $defs = \{FunctionDefinitions2ImplFnDefMap\,(fds) \mid$
.8 $mk\text{-}OAS`FunctionDefinitions\,(fds) \in \text{elems } dbs\}$ in
.9 $\forall\, mk\text{-}(fdm, fdm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(fdm', \text{-}) \in defs \setminus \{mk\text{-}(fdm, fdm_q)\} \cdot \text{dom } fdm \cap \text{dom } fdm' = \{\}$

annotations

.7 The pre-condition ensures that all implicit function identifiers are unique.

end annotations ;

37.0 $DefinitionBlocks2ValDefMap$:
.1 $OAS`DefinitionBlock^+ \to (CAS`Pattern \xrightarrow{m} CAS`ValDef)$
.2 $DefinitionBlocks2ValDefMap\,(dbs) \triangleq$
.3 merge $\{\, ValueDefinitions2ValDefMap\,(vds) \mid$
.4 $mk\text{-}OAS`ValueDefinitions\,(vds) \in \text{elems } dbs\}$
.5 pre let $defs = \{\, ValueDefinitions2ValDefMap\,(vds) \mid$
.6 $mk\text{-}OAS`ValueDefinitions\,(vds) \in \text{elems } dbs\}$ in
.7 $\forall\, vdm \in defs \cdot \forall\, vdm' \in defs \setminus \{vdm\} \cdot \text{dom } vdm \cap \text{dom } vdm' = \{\}$

annotations

.5 The pre-condition ensures that all value patterns are unique.

end annotations ;

38.0 $DefinitionBlocks2ExplOpDefMap$:
.1 $OAS`DefinitionBlock^+ \times CAS`TypeDef \to$
.2 $(CAS`Id \xrightarrow{m} CAS`ExplOpDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $DefinitionBlocks2ExplOpDefMap\,(dbs, \sigma) \triangleq$
.4 let $defs = \{OperationDefinitions2ExplOpDefMap\,(ods, \sigma) \mid$
.5 $mk\text{-}OAS`OperationDefinitions\,(ods) \in \text{elems } dbs\}$ in
.6 $mk\text{-}(\text{merge}\,\{odm \mid mk\text{-}(odm, \text{-}) \in defs\}, \text{merge}\,\{odm_q \mid mk\text{-}(\text{-}, odm_q) \in defs\})$
.7 pre let $defs = \{OperationDefinitions2ExplOpDefMap\,(ods, \sigma) \mid$
.8 $mk\text{-}OAS`OperationDefinitions\,(ods) \in \text{elems } dbs\}$ in
.9 $\forall\, mk\text{-}(odm, odm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(odm', \text{-}) \in defs \setminus \{mk\text{-}(odm, odm_q)\} \cdot \text{dom } odm \cap \text{dom } odm' = \{\}$

annotations

.7 The pre-condition ensures that all explicit operation identifiers are unique.

end annotations ;

39.0 $DefinitionBlocks2ImplOpDefMap$:
.1 $OAS`DefinitionBlock^+ \times CAS`TypeDef \to$
.2 $(CAS`Id \xrightarrow{m} CAS`ImplOpDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $DefinitionBlocks2ImplOpDefMap\,(dbs, \sigma) \triangleq$
.4 let $defs = \{OperationDefinitions2ImplOpDefMap\,(ods, \sigma) \mid$
.5 $mk\text{-}OAS`OperationDefinitions\,(ods) \in \text{elems } dbs\}$ in
.6 $mk\text{-}(\text{merge}\,\{odm \mid mk\text{-}(odm, \text{-}) \in defs\}, \text{merge}\,\{odm_q \mid mk\text{-}(\text{-}, odm_q) \in defs\})$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.7 pre let $defs = \{OperationDefinitions2ImplOpDefMap(ods, \sigma) \mid$
.8 $mk\text{-}OAS`OperationDefinitions(ods) \in \text{elems } dbs\}$ in
.9 $\forall mk\text{-}(odm, odm_q) \in defs \cdot$
.10 $\forall mk\text{-}(odm', \text{-}) \in defs \setminus \{mk\text{-}(odm, odm_q)\} \cdot \text{dom } odm \cap \text{dom } odm' = \{\}$

annotations

.7 The pre-condition ensures that all implicit operation identifiers are unique.

end annotations ;

40.0 $DefinitionBlocks2StateDef$:
.1 $OAS`DefinitionBlock^+ \to$
.2 $[CAS`StateDef] \times (CAS`ValueId \xrightarrow{m} CAS`TypeDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $DefinitionBlocks2StateDef(dbs) \triangleq$
.4 cases dbs :
.5 $\text{-}^\frown[mk\text{-}OAS`StateDef(sd)]^\frown\text{-} \to StateDef2StateDef(mk\text{-}OAS`StateDef(stid, fields, stinv, stinit))$,
.6 others $\to mk\text{-}(\text{nil}, \{\mapsto\}, \{\mapsto\})$
.7 end
.8 pre len $[dbs(i) \mid i \in \text{inds } dbs \cdot is\text{-}OAS`StateDef(dbs(i))] \leq 1$

annotations

.8 The pre-condition ensures that a specification contains at most one state definition.

end annotations

12.3.2 Definitions

12.3.2.1 Type Definitions

functions

41.0 $TypeDefinitions2TypeDefMap$:
.1 $OAS`TypeDef^+ \to$
.2 $(CAS`Id \xrightarrow{m} CAS`TypeDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $TypeDefinitions2TypeDefMap(tds) \triangleq$
.4 let $defs = \{TypeDefinition2TypeDefMap(td) \mid td \in \text{elems } tds\}$ in
.5 $mk\text{-}(\text{merge } \{tdm \mid mk\text{-}(tdm, \text{-}) \in defs\}, \text{merge } \{tdm_q \mid mk\text{-}(\text{-}, tdm_q) \in defs\})$
.6 pre let $defs = \{TypeDefinition2TypeDefMap(td) \mid td \in \text{elems } tds\}$ in
.7 $\forall mk\text{-}(tdm, tdm_q) \in defs \cdot$
.8 $\forall mk\text{-}(tdm', \text{-}) \in defs \setminus \{mk\text{-}(tdm, tdm_q)\} \cdot \text{dom } tdm \cap \text{dom } tdm' = \{\}$

annotations

.6 The pre-condition ensures that all type identifiers are unique.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
42.0  TypeDefinition2TypeDefMap :
 .1       OAS‘TypeDef →
 .2       (CAS‘ValueId -m-> CAS‘TypeDef) × (CAS‘Id -m-> CAS‘ExplFnDef)
 .3  TypeDefinition2TypeDefMap (td) △
 .4    let mk-(id_OAS, shape_CAS, typeinv_OAS) =
 .5          cases td :
 .6            mk-OAS‘UnTaggedTypeDef (id, shape, typeinv) →
 .7                mk-(id, Type2Type (shape), typeinv),
 .8            mk-OAS‘TaggedTypeDef (id, fields, typeinv) →
 .9                mk-(id, Type2Type (mk-OAS‘CompositeType (id, fields)), typeinv)
.10          end,
.11       lambda_CAS    = InvInitFn2Lambda (typeinv_OAS, shape_CAS),
.12       typeid_CAS    = Id2Id (id_OAS, {↦}),
.13       inv-id_CAS    = let mk-OAS‘ValueId(id) = id_OAS in
.14                       Id2Id (mk-OAS‘InvId(id), {↦}),
.15       typedef_CAS   = mk-CAS‘TypeDef(shape_CAS, mk-(inv-id_CAS, lambda_CAS)),
.16       explfndef_CAS = mk-CAS‘ExplFnDef (shape_CAS, mk-CAS‘BasicType(BOOLEAN),
.17                                         mk-CAS‘BoolLit(true), lambda_CAS, TOTAL) in
.18    mk-({typeid_CAS ↦ typedef_CAS}, {inv-id_CAS ↦ explfndef_CAS})
```

annotations

.4-.10 The type definition *td* is matched to a tagged or untagged type with id_{OAS} and $typeinv_{OAS}$ copied straight from the type's identifier, and invariant. $shape_{CAS}$ is matched with either *Type2Type*(*shape*) or *Type2Type*(*mk-OAS‘CompositeType*(*id*, *fields*)) depending on *td* being tagged or untagged.

.15 In the CAS a type definition consists of *shape*, and the invariant of *shape*; *inv*. *inv* has two components, namely an identifier and a lambda expression, where the lambda expression is the body of the invariant.

.16 The invariant is re-arranged into a total explicit function.

end annotations ;

```
43.0  InvInitFn2Lambda : [OAS‘InvInitFn] × CAS‘Type → CAS‘Lambda
 .1   InvInitFn2Lambda (invinitfn, type_CAS) △
 .2     let mk-(id_OAS, id_CAS) = GetUnusedId (),
 .3         parm_CAS            = mk-CAS‘Par (id_CAS, type_CAS),
 .4         body_CAS            = if invinitfn = nil then mk-CAS‘BoolLit (true)
 .5                               else let mk-OAS‘InvInitFn (pat, expr) = invinitfn in
 .6                                    MakeLetExpr (pat, nil, id_OAS, expr, {↦}) in
 .7     mk-CAS‘Lambda (parm_CAS, body_CAS)
```

annotations

.3 The Lambda function's parameters include the *shape* or *stid* (depending on *InvInitFn2Lambda* being called by a type definition or state definition).

.4-.6 The invariant's body (either from a type or state definition) must be converted to a lambda expression in the CAS. To achieve this, the invariant is split into its pattern and expression components. The pattern is combined with a 'brand new' identifier, to form a local value definition which is part of a let expression. The invariant expression is added to this let expression which is then translated to the CAS via *MakeLetExpr*.

end annotations ;

```
44.0  MakeLetExpr : OAS‘Pattern × [OAS‘Type] × OAS‘Expr × OAS‘Expr × Env →
 .1                  CAS‘LetExpr
 .2   MakeLetExpr (pat, type, val, body, ρ) △
 .3     let localdefs = [mk-OAS‘ValueDef (pat, type, val)] in
 .4     LetExpr2LetExpr (mk-OAS‘LetExpr (localdefs, body), ρ);
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
45.0  Type2Type : [OAS‘Type] → CAS‘Type
 .1   Type2Type (type) ≜
 .2     cases type :
 .3       mk-OAS‘BracketedType (type)    → Type2Type (type),
 .4       (BOOLEAN), (NAT), (NATONE),
 .5       (INTEGER), (RAT), (REAL),
 .6       (CHAR), (TOKEN)                → mk-CAS‘BasicType(type),
 .7       mk-OAS‘QuoteType (-)           → QuoteType2QuoteType (type),
 .8       mk-OAS‘CompositeType (-, -)    → CompositeType2CompositeType (type),
 .9       mk-OAS‘UnionType (-)           → UnionType2UnionType (type),
 .10      mk-OAS‘ProductType (-)         → ProductType2ProductType (type),
 .11      mk-OAS‘OptionalType (-)        → OptionalType2OptionalType (type),
 .12      mk-OAS‘SetType (-)             → SetType2SetType (type),
 .13      mk-OAS‘Seq0Type (-)            → Seq0Type2Seq0Type (type),
 .14      mk-OAS‘Seq1Type (-)            → Seq1Type2Seq1Type (type),
 .15      mk-OAS‘GeneralMapType (-, -)   → GeneralMapType2GeneralMapType (type),
 .16      mk-OAS‘InjectiveMapType (-, -) → InjectiveMapType2InjectiveMapType (type),
 .17      mk-OAS‘PartialFnType (-, -)    → FnType2FnType (type),
 .18      mk-OAS‘Name (-)                → TypeName2TypeId (type),
 .19      mk-OAS‘TypeVarId (-)           → TypeVar2TypeVar (type),
 .20      others                         → mk-CAS‘BasicType (UNIT)
 .21    end ;

46.0  QuoteType2QuoteType : OAS‘QuoteType → CAS‘QuoteType
 .1   QuoteType2QuoteType (mk-OAS‘QuoteType (lit)) ≜
 .2     let mk-OAS‘QuoteLit (val) = lit,
 .3         val_CAS               = [mk-CAS‘CharLit(val(i)) | i ∈ inds val],
 .4         lit_CAS               = mk-CAS‘QuoteLit (val_CAS) in
 .5     mk-CAS‘QuoteType (lit_CAS) ;

47.0  CompositeType2CompositeType : OAS‘CompositeType → CAS‘CompositeType
 .1   CompositeType2CompositeType (mk-OAS‘CompositeType (id, fields)) ≜
 .2     let id_CAS     = Id2Id (id, {↦}),
 .3         fields_CAS = [Field2Field (fields(i)) | i ∈ inds fields] in
 .4     mk-CAS‘CompositeType (id_CAS, fields_CAS)
 .5   pre is-OAS‘ValueId (id)

annotations
  .5 The pre-condition ensures that the tag of the composite type has no reserved prefix.
end annotations ;

48.0  Field2Field : OAS‘Field → CAS‘Field
 .1   Field2Field (mk-OAS‘Field (sel, type)) ≜
 .2     let sel_CAS  = if sel = nil then nil
 .3                    else Id2Id (sel, {↦}),
 .4         type_CAS = Type2Type (type) in
 .5     mk-CAS‘Field (sel_CAS, type_CAS)
 .6   pre is-OAS‘ValueId (sel)

annotations
  .6 The pre-condition ensures that the field name has no reserved prefix.
end annotations ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
49.0  UnionType2UnionType : OAS`UnionType → CAS`UnionType
 .1   UnionType2UnionType (mk-OAS`UnionType (summands)) ≜
 .2     let tps_CAS = { Type2Type (summands(i)) | i ∈ inds summands } in
 .3     mk-CAS`UnionType (tps_CAS)
 .4   pre len summands = card elems summands
```

annotations

.4 The pre-condition ensures that there are no multiple union type components.

end annotations ;

```
50.0  ProductType2ProductType : OAS`ProductType → CAS`ProductType
 .1   ProductType2ProductType (mk-OAS`ProductType (factors)) ≜
 .2     let fields_CAS = [Type2Type (factors (i)) | i ∈ inds factors] in
 .3     mk-CAS`ProductType (fields_CAS) ;

51.0  OptionalType2OptionalType : OAS`OptionalType → CAS`OptionalType
 .1   OptionalType2OptionalType (mk-OAS`OptionalType (type)) ≜
 .2     let tp_CAS = Type2Type (type) in
 .3     mk-CAS`OptionalType (tp_CAS) ;

52.0  SetType2SetType : OAS`SetType → CAS`SetType
 .1   SetType2SetType (mk-OAS`SetType (elemtp)) ≜
 .2     let elemtp_CAS = Type2Type (elemtp) in
 .3     mk-CAS`SetType (elemtp_CAS) ;

53.0  Seq0Type2Seq0Type : OAS`Seq0Type → CAS`Seq0Type
 .1   Seq0Type2Seq0Type (mk-OAS`Seq0Type (elemtp)) ≜
 .2     let elemtp_CAS = Type2Type (elemtp) in
 .3     mk-CAS`Seq0Type (elemtp_CAS) ;

54.0  Seq1Type2Seq1Type : OAS`Seq1Type → CAS`SeqType
 .1   Seq1Type2Seq1Type (mk-OAS`Seq1Type (elemtp)) ≜
 .2     let elemtp_CAS = Type2Type (elemtp) in
 .3     mk-CAS`Seq1Type (elemtp_CAS) ;

55.0  GeneralMapType2GeneralMapType : OAS`GeneralMapType → CAS`GeneralMapType
 .1   GeneralMapType2GeneralMapType (mk-OAS`GeneralMapType (mapdom, maprng)) ≜
 .2     let dom_CAS = Type2Type (mapdom),
 .3         rng_CAS = Type2Type (maprng) in
 .4     mk-CAS`GeneralMapType (dom_CAS, rng_CAS) ;

56.0  InjectiveMapType2InjectiveMapType : OAS`InjectiveMapType → CAS`InjectiveMapType
 .1   InjectiveMapType2InjectiveMapType (mk-OAS`InjectiveMapType (mapdom, maprng)) ≜
 .2     let dom_CAS = Type2Type (mapdom),
 .3         rng_CAS = Type2Type (maprng) in
 .4     mk-CAS`InjectiveMapType (dom_CAS, rng_CAS) ;

57.0  FnType2FnType : OAS`FnType → CAS`FnType
 .1   FnType2FnType (fntype) ≜
 .2     let dom_CAS = if fntype.fndom = UNITTYPE
 .3                   then mk-CAS`BasicType (UNIT)
 .4                   else Type2Type (fntype.fndom),
 .5         rng_CAS = Type2Type (fntype.fnrng) in
 .6     mk-CAS`FnType (dom_CAS, rng_CAS) ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

58.0 $TypeName2TypeId : OAS`TypeName \to CAS`TypeId$

.1 $TypeName2TypeId\,(mk\text{-}OAS`Name\,(name)) \triangleq$

.2 let $id_{CAS} = Id2Id\,(name, \{\mapsto\})$ in

.3 $mk\text{-}CAS`TypeId\,(id_{CAS})$

.4 pre $is\text{-}OAS`ValueId\,(name)$

annotations

.4 The pre-condition ensures that the type name has no reserved prefix.

end annotations ;

59.0 $TypeVar2TypeVar : OAS`TypeVar \to CAS`TypeVar$

.1 $TypeVar2TypeVar\,(mk\text{-}OAS`TypeVarId\,(id)) \triangleq$

.2 let $id' = Id2Id\,(id, \{\mapsto\})$,

.3 $mk\text{-}OAS`ValueId\,(id_{CAS}) = id'$ in

.4 $mk\text{-}CAS`TypeVar\,(mk\text{-}CAS`TypeVarId(id_{CAS}))$

.5 pre $is\text{-}OAS`ValueId\,(id)$

annotations

.5 The pre-condition ensures that the type variable identifier has no reserved prefix.

end annotations

12.3.2.2 State Definition

functions

60.0 $StateDef2StateDef : OAS`StateDef \to CAS`StateDef \times$

.1 $(CAS`ValueId \xrightarrow{m} CAS`TypeDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$

.2 $StateDef2StateDef\,(mk\text{-}OAS`StateDef\,(stid, fields, stinv, stinit)) \triangleq$

.3 let $mk\text{-}(typedefmap_{CAS}, inv\text{-}stidm_{CAS}) =$

.4 $TypeDefinition2TypeDefMap\,(mk\text{-}OAS`TaggedTypeDef(stid, fields, stinv))$,

.5 $stid_{CAS} = Id2Id\,(stid, \{\mapsto\})$,

.6 $tp_{CAS} = stid_{CAS}$,

.7 $init\text{-}stid_{CAS}$ = let $mk\text{-}OAS`ValueId(id) = stid$ in

.8 $Id2Id\,(mk\text{-}OAS`InitId(id), \{\mapsto\})$,

.9 $lambda_{CAS} = InvInitFn2Lambda\,(stinit, tp_{CAS})$,

.10 $init_{CAS} = mk\text{-}(init\text{-}stid_{CAS}, lambda_{CAS})$,

.11 $statedef_{CAS} = mk\text{-}CAS`StateDef\,(stid_{CAS}, tp_{CAS}, init_{CAS})$,

.12 $explfndef_{CAS} = mk\text{-}CAS`ExplFnDef\,(tp_{CAS}, mk\text{-}CAS`BasicType(\text{BOOL}),$

.13 $mk\text{-}CAS`BoolLit(\text{true}), lambda_{CAS}, \text{TOTAL})$,

.14 $init\text{-}stidm_{CAS} = \{init\text{-}stid_{CAS} \mapsto explfndef_{CAS}\}$ in

.15 $mk\text{-}(statedef_{CAS}, typedefmap_{CAS}, inv\text{-}stidm_{CAS} \cup init\text{-}stidm_{CAS})$

.16 pre $is\text{-}OAS`ValueId\,(stid)$

annotations

.3-.4 Without the initial invariant function, the state resembles a tagged type, which can be transformed into the CAS by *TypeDefiniton2TypeDefMap*. This produces two maps; a map of the converted state identifier with the state's type, and a map of the state identifier with the invariant.

.9 The state's initial component is combined with the name of the state and transformed into the CAS as a lambda expression.

.12-.15 The initial condition is transformed into a map whose domain consists of the state's identifier prefixed by '*init-*', and range consists of a truth-valued function. This map is then merged with the invariant function's map.

.16 The pre-condition ensures that the state name has no reserved prefix.

end annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.2.3 Value Definitions

functions

61.0 $ValueDefinitions2ValDefMap : OAS`ValueDef^+ \to (CAS`Pattern \xrightarrow{m} CAS`ValDef)$

.1 $ValueDefinitions2ValDefMap\,(vds) \triangleq$

.2 merge $\{ ValueDef2ValDefMap\,(vd) \mid vd \in \mathsf{elems}\ vds \}$

.3 pre let $defs = \{ ValueDef2ValDefMap\,(vd) \mid vd \in \mathsf{elems}\ vds \}$ in

.4 $\forall\, vdm \in defs \cdot \forall\, vdm' \in defs \setminus \{vdm\} \cdot \mathsf{dom}\ vdm \cap \mathsf{dom}\ vdm' = \{\}$

annotations

.3 The pre-condition ensures that all value patterns are unique.

end annotations ;

62.0 $ValueDef2ValDefMap : OAS`ValueDef \to (CAS`Pattern \xrightarrow{m} CAS`ValDef)$

.1 $ValueDef2ValDefMap\,(mk\text{-}OAS`ValueDef\,(pat, type, expr)) \triangleq$

.2 let $pat_{CAS} = Pattern2Pattern\,(pat, \{\mapsto\})$,

.3 $type_{CAS} = \mathsf{if}\ type = \mathsf{nil}\ \mathsf{then}\ \mathsf{nil}$

.4 $\mathsf{else}\ Type2Type\,(type)$,

.5 $val_{CAS} = Expr2Expr\,(expr, \{\mapsto\})$ in

.6 $\{pat_{CAS} \mapsto mk\text{-}CAS`ValDef\,(type_{CAS}, val_{CAS})\}$

12.3.2.4 Function Definitions

functions

63.0 $FunctionDefinitions2ExplPolyFnDefMap :$

.1 $OAS`FunctionDef^+ \to$

.2 $(CAS`Id \xrightarrow{m} CAS`ExplPolyFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplPolyFnDef)$

.3 $FunctionDefinitions2ExplPolyFnDefMap\,(fds) \triangleq$

.4 let $defs = \{FunctionDef2ExplPolyFnDefMap\,(fd) \mid$

.5 $fd : OAS`ExplFnDef \cdot fd \in \mathsf{elems}\ fds \wedge IsPolymorphic\,(fd)\}$ in

.6 $mk\text{-}(\mathsf{merge}\ \{fnm \mid mk\text{-}(fnm, \text{-}) \in defs\}, \mathsf{merge}\ \{fnm_q \mid mk\text{-}(\text{-}, fnm_q) \in defs\})$

.7 pre let $defs = \{FunctionDef2ExplPolyFnDefMap\,(fd) \mid$

.8 $fd : OAS`ExplFnDef \cdot fd \in \mathsf{elems}\ fds \wedge IsPolymorphic\,(fd)\}$ in

.9 $\forall\, mk\text{-}(tdm, tdm_q) \in defs \cdot$

.10 $\forall\, mk\text{-}(tdm', \text{-}) \in defs \setminus \{mk\text{-}(tdm, tdm_q)\} \cdot$

.11 $\mathsf{dom}\ tdm \cap \mathsf{dom}\ tdm' = \{\}$

annotations

.7 The pre-condition ensures that all explicit polymorphic function identifiers are unique.

end annotations ;

64.0 $IsPolymorphic : OAS`FunctionDef \to \mathbb{B}$

.1 $IsPolymorphic\,(fd) \triangleq$

.2 $fd.tpparms \neq [\,]$

annotations

64.0 This is a test to see whether a function definition is polymorphic or not by looking for the presence of type parameters within the definition.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
65.0  FunctionDef2ExplPolyFnDefMap :
.1        OAS'ExplFnDef → (CAS'Id -m-> CAS'ExplPolyFnDef) × (CAS'Id -m-> CAS'ExplPolyFnDef)
.2    FunctionDef2ExplPolyFnDefMap (efd) ≜
.3      let mk-OAS'ExplFnDef (id, tpparms, type, -, parms, body, pre) = efd in
.4      let id_CAS                = Id2Id (id, {↦}),
.5          explpolyfndef_CAS     = ExplFnDef2ExplFnDef (efd, {↦}),
.6          pre-id_CAS            = let mk-OAS'ValueId(pre-id) = id in
.7                                  Id2Id (mk-OAS'PreId(pre-id), {↦}),
.8          pre-explpolyfndef_CAS =
.9              let pre-id_OAS = let mk-OAS'ValueId(pre-id) = id in mk-OAS'PreId(pre-id),
.10                 type_pre-id  = mk-OAS'TotalFnType(type.fndom, BOOLEAN),
.11                 pre-efd      = mk-OAS'ExplFnDef (pre-id_OAS, tpparms, type_pre-id,
.12                                                  pre-id_OAS, parms, pre, nil) in
.13             ExplFnDef2ExplFnDef (pre-efd, {↦}) in
.14      mk-({id_CAS ↦ explpolyfndef_CAS}, {pre-id_CAS ↦ pre-explpolyfndef_CAS})
.15   pre let mk-OAS'ExplFnDef (id, -, -, id_repeated, -, -, -) = efd in
.16       id = id_repeated ∧ (is-OAS'ValueId (id) ∨ is-OAS'PreId (id) ∨ is-OAS'PostId (id))
```

annotations

.5 Initiate the transformation of the explicit function definition.

.8-.13 Generate an explicit function from the function's pre-condition with the name *pre-id*.

.15 The pre-condition expresses the requirement that the two occurrences of the function name in the function header are the same, and that the function name either has no prefix, or has the prefix '*pre-*' or '*post-*'.

end annotations ;

```
66.0  MakeLambda : OAS'ParametersList × OAS'FnType × OAS'Expr × Env → CAS'Lambda
.1    MakeLambda (parms, type, body, ρ) ≜
.2      let patternlist     = hd parms,
.3          mk-(-, id_CAS)  = GetUnusedId (),
.4          type_CAS        = Type2Type (type.fndom),
.5          par_CAS         = mk-CAS'Par (id_CAS, type_CAS),
.6          body_CAS        = let body_Let  = if len parms = 1 then Expr2Expr (body, ρ)
.7                                           else MakeLambda (tl parms, type.fnrng, body, ρ),
.8                                value_Let = mk-CAS'ValDef(type_CAS, id_CAS),
.9                                pat_Let   = if len patternlist = 1
.10                                           then Pattern2Pattern (hd patternlist, ρ)
.11                                           else Pattern2Pattern (mk-OAS'TuplePattern (patternlist), ρ),
.12                               vals_Let  = {pat_Let ↦ value_Let} in
.13                            mk-CAS'LetExpr(vals_Let, {↦}, {↦}, body_Let) in
.14      mk-CAS'Lambda (par_CAS, body_CAS)
.15   pre NumberOfParsMatchesFnType (parms, type)
```

annotations

66.0 The *MakeLambda* function is used in the transformation of explicit functions. When an explicit function is transformed, the body must be re-organized as a lambda expression, allowing the function's parameters to be applied to the function body one at a time .

.2 When a function is curried, there are more than one set of parameters. The first parameter is pulled off the list. This parameter may contain a list of patterns.

.4 *type.fndom* contains the input type parameters from the *FnType* of the explicit function definition undergoing transformation.

.6-.7 When transforming the body of the explicit function imported from function definitions, the parameters list must be checked to see if the function is curried. If this is the case, *MakeLambda* is recursively called applying each parameter to the body separately.

.8 Construct a value definition with a 'new' identifier.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.12 Turn the input parameters' pattern lists into the domain of a value definition map with the previously defined value definition in the range.

.13 Use a let expression to represent the lambda's body.

.15 The pre-condition ensures that the parameter profile matches the type profile of the function heading.

end annotations ;

```
67.0  NumberOfParsMatchesFnType : OAS'ParametersList × OAS'FnType → 𝔹
 .1   NumberOfParsMatchesFnType (parms, fntype) ≙
 .2     let FirstPairOK = cases len hd parms :
 .3                              0      → fntype.fndom = UNITTYPE,
 .4                              1      → ¬ is-OAS'ProductType (fntype.fndom),
 .5                              others → is-OAS'ProductType (fntype.fndom) ∧
 .6                                       len hd parms = len fntype.fndom.factors
 .7                       end in
 .8     if len parms = 1 then FirstPairOK
 .9     else FirstPairOK ∧
 .10        (is-OAS'PartialFnType (fntype.fnrng) ∨ is-OAS'TotalFnType (fntype.fnrng)) ∧
 .11        NumberOfParsMatchesFnType (tl parms, fntype.fnrng)
```

annotations

67.0 For this function to return the boolean value 'true', the number of parameters of the function under test must match the number of types in the domain of the function's *FnType*. In the case of curried functions, this function has to match each parameter in the *ParametersList* with each domain of the 'nested' function type.

end annotations ;

```
68.0  OTypeVarList2TypeVarList : OAS'TypeVar* → CAS'TypeVar*
 .1   OTypeVarList2TypeVarList (tvars) ≙
 .2     [TypeVar2TypeVar (tvars(i)) | i ∈ inds tvars]
```

annotations

68.0 Used by polymorphic functions to convert type parameters.

end annotations ;

```
69.0  FunctionDefinitions2ExplFnDefMap :
 .1       OAS'FunctionDef+ → (CAS'Id -m→ CAS'ExplFnDef) × (CAS'Id -m→ CAS'ExplFnDef)
 .2   FunctionDefinitions2ExplFnDefMap (fds) ≙
 .3     let defs = {FunctionDef2ExplFnDefMap (fd) |
 .4                   fd : OAS'ExplFnDef · fd ∈ elems fds ∧ ¬ IsPolymorphic (fd)} in
 .5     mk-(merge {fnm | mk-(fnm, -) ∈ defs}, merge {fnm_q | mk-(-, fnm_q) ∈ defs})
 .6   pre let defs = {FunctionDef2ExplFnDefMap (fd) |
 .7                   fd : OAS'ExplFnDef · fd ∈ elems fds ∧ ¬ IsPolymorphic (fd)} in
 .8       ∀ mk-(tdm, tdm_q) ∈ defs ·
 .9           ∀ mk-(tdm', -) ∈ defs \ {mk-(tdm, tdm_q)} ·
 .10              dom tdm ∩ dom tdm' = { }
```

annotations

.6 The pre-condition ensures that all explicit function identifiers are unique.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

70.0 $FunctionDef2ExplFnDefMap$:
.1 $\quad OAS`ExplFnDef \to (CAS`Id \xrightarrow{m} CAS`ExplFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.2 $FunctionDef2ExplFnDefMap\,(efd) \triangleq$
.3 $\quad$ let $mk\text{-}OAS`ExplFnDef\,(id, \text{-}, type, \text{-}, parms, body, pre) = efd$ in
.4 $\quad$ let $id_{CAS} = Id2Id\,(id, \{\mapsto\})$,
.5 $\quad\quad explfndef_{CAS} = ExplFnDef2ExplFnDef\,(efd, \{\mapsto\})$,
.6 $\quad\quad pre\text{-}id_{CAS} =$ let $mk\text{-}OAS`ValueId(pre\text{-}id) = id$ in
.7 $\quad\quad\quad\quad Id2Id\,(mk\text{-}OAS`PreId(pre\text{-}id), \{\mapsto\})$,
.8 $\quad\quad pre\text{-}explfndef_{CAS} =$
.9 $\quad\quad\quad$ let $pre\text{-}id_{OAS} =$ let $mk\text{-}OAS`ValueId(pre\text{-}id) = id$ in $mk\text{-}OAS`PreId(pre\text{-}id)$,
.10 $\quad\quad\quad\quad type_{pre\text{-}id} = mk\text{-}OAS`TotalFnType(type.fndom, \text{BOOLEAN})$,
.11 $\quad\quad\quad\quad pre\text{-}efd = mk\text{-}OAS`ExplFnDef\,(pre\text{-}id_{OAS}, [\,], type_{pre\text{-}id},$
.12 $\quad\quad\quad\quad\quad\quad\quad\quad pre\text{-}id_{OAS}, parms, pre, \text{nil})$ in
.13 $\quad\quad\quad ExplFnDef2ExplFnDef\,(pre\text{-}efd, \{\mapsto\})$ in
.14 $\quad mk\text{-}(\{id_{CAS} \mapsto explfndef_{CAS}\}, \{pre\text{-}id_{CAS} \mapsto pre\text{-}explfndef_{CAS}\})$
.15 pre let $mk\text{-}OAS`ExplFnDef\,(id, \text{-}, \text{-}, id_repeated, \text{-}, \text{-}, \text{-}) = efd$ in
.16 $\quad id = id_repeated \wedge (is\text{-}OAS`ValueId\,(id) \vee is\text{-}OAS`PreId\,(id) \vee is\text{-}OAS`PostId\,(id) \vee$
.17 $\quad is\text{-}OAS`InvId\,(id) \vee is\text{-}OAS`InitId\,(id))$

annotations

.8-.13 Generate an explicit function for the pre-condition of the function with the name *pre-id*.

.15 The pre-condition expresses the requirement that the two occurrences of the function name in the function header are the same, and that the function name either has no prefix, or has the prefix '*pre-*' or '*post-*', '*init-*' or '*inv-*'.

end annotations ;

71.0 $ExplFnDef2ExplFnDef$:
.1 $\quad OAS`ExplFnDef \times Env \to (CAS`ExplFnDef \mid CAS`ExplPolyFnDef)$
.2 $ExplFnDef2ExplFnDef\,(mk\text{-}OAS`ExplFnDef\,(\text{-}, tpparms, type, \text{-}, parms, body, pre), \rho) \triangleq$
.3 $\quad$ let $tpparms_{CAS} = OTypeVarList2TypeVarList\,(tpparms)$,
.4 $\quad\quad domtp_{CAS} = Type2Type\,(type.fndom)$,
.5 $\quad\quad restp_{CAS} = Type2Type\,(type.fnrng)$,
.6 $\quad\quad polytp_{CAS} = FnType2FnType\,(type)$,
.7 $\quad\quad pre_{CAS} = OExpr2TrueOrExpr\,(pre, \{\mapsto\})$,
.8 $\quad\quad body_{CAS} = MakeLambda\,(parms, type, body, \rho)$,
.9 $\quad\quad kind_{CAS} =$ if $is\text{-}OAS`PartialFnType\,(type)$ then PARTIAL
.10 $\quad\quad\quad\quad$ else TOTAL in
.11 $\quad$ if $tpparms_{CAS} = [\,]$
.12 $\quad$ then $mk\text{-}CAS`ExplFnDef\,(domtp_{CAS}, restp_{CAS}, pre_{CAS}, body_{CAS}, kind_{CAS})$
.13 $\quad$ else $mk\text{-}CAS`ExplPolyFnDef\,(tpparms_{CAS}, polytp_{CAS}, pre_{CAS}, body_{CAS}, kind_{CAS})$

annotations

.3-.9 Generate each part of an explicit function in the CAS.

.11-.13 Test to see if the explicit function is polymorphic or not. Exchange *domtp* with *tpparams*, and *restp* with *polytp* if this is the case.

end annotations ;

72.0 $FunctionDefinitions2ImplPolyFnDefMap$:
.1 $\quad OAS`FunctionDef^{+} \to$
.2 $\quad (CAS`Id \xrightarrow{m} CAS`ImplPolyFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplPolyFnDef)$
.3 $FunctionDefinitions2ImplPolyFnDefMap\,(fds) \triangleq$
.4 $\quad$ let $defs = \{FunctionDef2ImplPolyFnDefMap\,(fd) \mid$
.5 $\quad\quad\quad fd : OAS`ImplFnDef \cdot fd \in \text{elems } fds \wedge IsPolymorphic\,(fd)\}$ in
.6 $\quad mk\text{-}(\text{merge}\,\{fnm \mid mk\text{-}(fnm, \text{-}) \in defs\}, \text{merge}\,\{fnm_q \mid mk\text{-}(\text{-}, fnm_q) \in defs\})$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
.7      pre let defs = {FunctionDef2ImplPolyFnDefMap (fd) |
.8                        fd : OAS'ImplFnDef · fd ∈ elems fds ∧ IsPolymorphic (fd)} in
.9          ∀ mk-(tdm, tdm_q) ∈ defs ·
.10             ∀ mk-(tdm', -) ∈ defs \ {mk-(tdm, tdm_q)} ·
.11                 dom tdm ∩ dom tdm' = { }
```

annotations

.7 The pre-condition ensures that all implicit polymorphic function identifiers are unique.

end annotations ;

```
73.0    FunctionDef2ImplPolyFnDefMap :
.1          OAS'ImplFnDef → (CAS'Id -m→ CAS'ImplPolyFnDef) × (CAS'Id -m→ CAS'ExplPolyFnDef)
.2      FunctionDef2ImplPolyFnDefMap (ifd) ≜
.3        let mk-OAS'ImplFnDef (id, tpparms, partps, residtype, fnpre, fnpost) = ifd in
.4        let id_CAS                 = Id2Id (id, {↦}),
.5            implpolyfndef_CAS      = ImplFnDef2ImplFnDef (ifd),
.6            pre-id_CAS             = let mk-OAS'ValueId(pre-id) = id in
.7                                     Id2Id (mk-OAS'PreId(pre-id), {↦}),
.8            pre-implpolyfndef_CAS =
.9                let pre-id_OAS = let mk-OAS'ValueId(pre-id) = id in mk-OAS'PreId(pre-id),
.10                   mk-(parms_pre-id, domtype_pre-id) =
.11                       SplitOPatTypePairList (partps),
.12                   type_pre-id  = mk-OAS'TotalFnType(domtype_pre-id, BOOLEAN),
.13                   pre-ifd      = mk-OAS'ExplFnDef (pre-id_OAS, tpparms, type_pre-id,
.14                                                    pre-id_OAS, parms_pre-id, pre, nil) in
.15               ExplFnDef2ExplFnDef (pre-ifd, {↦}),
.16           post-id_CAS            = let mk-OAS'ValueId(post-id) = id in
.17                                    Id2Id (mk-OAS'PostId(post-id), {↦}),
.18           post-implpolyfndef_CAS =
.19               let post-id_OAS = let mk-OAS'ValueId(post-id) = id in mk-OAS'PostId(post-id),
.20                   mk-(parms_post-id, domtype_post-id) =
.21                       SplitOPatTypePairList (partps),
.22                   type_post-id  = mk-OAS'TotalFnType
.23                                       (mk-OAS'ProductType ([domtype_post-id, residtype.type]), BOOLEAN),
.24                   post-ifd      = mk-OAS'ExplFnDef (post-id_OAS, [], type_post-id, post-id_OAS,
.25                                                     parms_post-id ↷ [residtype.id], post, nil) in
.26               ExplFnDef2ExplFnDef (post-ifd, {↦}) in
.27       mk- ({id_CAS ↦ implpolyfndef_CAS},
.28            {pre-id_CAS ↦ pre-implpolyfndef_CAS} ⩊ {post-id_CAS ↦ post-implpolyfndef_CAS})
.29     pre let mk-OAS'ImplFnDef (id, -, -, mk-OAS'IdType (rid, -), -, -) = ifd in
.30       is-OAS'ValueId (id) ∧ is-OAS'ValueId (rid)
```

annotations

.8-.15 Generate an explicit function for the pre-condition of the function with the name *pre-id*.

.18-.26 Generate an explicit function for the post-condition of the function with the name *post-id*.

.29 The pre-condition expresses that the function name and the result identifier have no reserved prefix.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
74.0  ImplFnDef2ImplFnDef : OAS'ImplFnDef → (CAS'ImplFnDef | CAS'ImplPolyFnDef)
.1    ImplFnDef2ImplFnDef (ifd) ≜
.2      let mk-OAS'ImplFnDef (id, tpparms, partps, residtype, fnpre, fnpost) = ifd,
.3          mk-OAS'IdType (rid, fnrng) = residtype in
.4      let mk-(patlist, fndom) = SplitOPatTypePairList (partps),
.5          mk-(id_OAS, id_CAS) = GetUnusedId (),
.6          tpparms_CAS         = OTypeVarList2TypeVarList (tpparms),
.7          heading_CAS         = let fndom_CAS = if fndom = nil
.8                                                then mk-CAS'BasicType(UNIT)
.9                                                elseif len fndom = 1
.10                                               then Type2Type (hd fndom)
.11                                               else ProductType2ProductType
.12                                                      (mk-OAS'ProductType(fndom)),
.13                                    par_CAS    = mk-CAS'Par (id_CAS, fndom_CAS),
.14                                    restp_CAS  = Type2Type (fnrng),
.15                                    resum_CAS  = Id2Id (rid, {↦}) in
.16                                  mk-CAS'ImplFnHeading (par_CAS, restp_CAS, resum_CAS),
.17          pre_CAS             = if fnpre = nil then mk-CAS'BoolLit(true)
.18                                else MakeLetExpr (patlist, fnrng, id_OAS, fnpre, {↦}),
.19          post_CAS            = MakeLetExpr (patlist, fnrng, id_OAS, fnpost, {↦}) in
.20     if tpparms_CAS = [ ]
.21     then mk-CAS'ImplFnDef (heading_CAS, pre_CAS, post_CAS)
.22     else mk-CAS'ImplPolyFnDef (tpparms_CAS, heading_CAS, pre_CAS, post_CAS)
```

annotations

.3 Separate the function type into two parts; *rid*, and *fnrng*.

.4 *SplitOPatTypePairList* removes all the patterns from the parameter types and construct a sequence of patterns. Similarly the patterns' types are removed and reformed as a sequence of types.

.7-.16 Generate an implicit function heading with a unique identifier (of the same type as the input parameters), a result type, and a value identifier.

.17-.19 Both pre_{CAS} and $post_{CAS}$ are assigned the result of calls to *MakeLetExpr*. As *MakeLetExpression* calls *LetExpr2LetExpr* the result is a let expression consisting of a local value definition map; $Pattern \xrightarrow{m} ValDef$, and the transformed pre/post-condition.

.20-.22 Test for polymorphic functions. The result of *ImplFnDef2ImplFnDef* is dependent on the presence of polymorphic type parameters.

end annotations ;

```
75.0  SplitOPatTypePairList : OAS'ParameterTypes → ([OAS'Pattern] × [OAS'Type])
.1    SplitOPatTypePairList (ptl) ≜
.2      let ptl_e    = conc [[let mk-OAS'PatTypePair (ps, t) = ptl (j) in mk- (ps (i), t) |
.3                            i ∈ let mk-OAS'PatTypePair (ps, -) = ptl (j) in inds ps] |
.4                           j ∈ inds ptl],
.5          patterns = [let mk- (p, -) = ptl_e (i) in p | i ∈ inds ptl_e],
.6          types    = [let mk- (-, t) = ptl_e (i) in t | i ∈ inds ptl_e] in
.7      cases len patterns :
.8        0      → mk- (nil, nil),
.9        1      → mk- (hd patterns, hd types),
.10       others → mk- (mk-OAS'TuplePattern(patterns), mk-OAS'ProductType(types))
.11     end
```

annotations

.2-.4 The conversion of the parameter type list to a sequence of patterns each with its own type involves selecting a pattern type pair list, and whilst keeping this constant, each pattern (indexed by i) is paired with the type of pattern type pair, ie $ptl(j)$'s type. The result is a sequence of pattern type pairs.

.5-.6 A sequence of patterns and a sequence of types are pulled out of the pattern type pair list.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

end annotations ;

76.0 *FunctionDefinitions2ImplFnDefMap* :
.1 $OAS`FunctionDef^+ \to$
.2 $(CAS`Id \xrightarrow{m} CAS`ImplFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $FunctionDefinitions2ImplFnDefMap\,(fds) \triangleq$
.4 let $defs = \{FunctionDef2ImplFnDefMap\,(fd) \mid$
.5 $fd : OAS`ImplFnDef \cdot fd \in \mathsf{elems}\ fds \wedge \neg\, IsPolymorphic\,(fd)\}$ in
.6 $mk\text{-}(\mathsf{merge}\ \{fnm \mid mk\text{-}(fnm, \text{-}) \in defs\}, \mathsf{merge}\ \{fnm_q \mid mk\text{-}(\text{-}, fnm_q) \in defs\})$
.7 pre let $defs = \{FunctionDef2ImplFnDefMap\,(fd) \mid$
.8 $fd : OAS`ImplFnDef \cdot fd \in \mathsf{elems}\ fds \wedge \neg\, IsPolymorphic\,(fd)\}$ in
.9 $\forall\, mk\text{-}(tdm, tdm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(tdm', \text{-}) \in defs \setminus \{mk\text{-}(tdm, tdm_q)\} \cdot$
.11 $\mathsf{dom}\ tdm \cap \mathsf{dom}\ tdm' = \{\,\}$

annotations

.7 The pre-condition ensures that all implicit function identifiers are unique.

end annotations ;

77.0 *FunctionDef2ImplFnDefMap* :
.1 $OAS`ImplFnDef \to (CAS`Id \xrightarrow{m} CAS`ImplFnDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.2 $FunctionDef2ImplFnDefMap\,(ifd) \triangleq$
.3 let $mk\text{-}OAS`ImplFnDef\,(id, \text{-}, partps, mk\text{-}OAS`IdType\,(rid, rtype), fnpre, fnpost) = ifd$ in
.4 let $id_{CAS} = Id2Id\,(id, \{\mapsto\})$,
.5 $implfndef_{CAS} = ImplFnDef2ImplFnDef\,(ifd)$,
.6 $pre\text{-}id_{CAS} =$ let $mk\text{-}OAS`ValueId(pre\text{-}id) = id$ in
.7 $Id2Id\,(mk\text{-}OAS`PreId(pre\text{-}id), \{\mapsto\})$,
.8 $pre\text{-}implfndef_{CAS} =$
.9 let $pre\text{-}id_{OAS} =$ let $mk\text{-}OAS`ValueId(pre\text{-}id) = id$ in $mk\text{-}OAS`PreId(pre\text{-}id)$,
.10 $mk\text{-}(parms_{pre\text{-}id}, domtype_{pre\text{-}id}) =$
.11 $SplitOPatTypePairList\,(partps)$,
.12 $type_{pre\text{-}id} = mk\text{-}OAS`TotalFnType(domtype_{pre\text{-}id}, \text{BOOLEAN})$,
.13 $pre\text{-}ifd = mk\text{-}OAS`ExplFnDef\,(pre\text{-}id_{OAS}, [\,], type_{pre\text{-}id},$
.14 $pre\text{-}id_{OAS}, parms_{pre\text{-}id}, fnpre, \mathsf{nil})$ in
.15 $ExplFnDef2ExplFnDef\,(pre\text{-}ifd, \{\mapsto\})$,
.16 $post\text{-}id_{CAS} =$ let $mk\text{-}OAS`ValueId(post\text{-}id) = id$ in
.17 $Id2Id\,(mk\text{-}OAS`PostId(post\text{-}id), \{\mapsto\})$,
.18 $post\text{-}implfndef_{CAS} =$
.19 let $post\text{-}id_{OAS} =$ let $mk\text{-}OAS`ValueId(post\text{-}id) = id$ in $mk\text{-}OAS`PostId(post\text{-}id)$,
.20 $mk\text{-}(parms_{post\text{-}id}, domtype_{post\text{-}id}) = SplitOPatTypePairList\,(partps)$,
.21 $type_{post\text{-}id} = mk\text{-}OAS`TotalFnType\,(mk\text{-}OAS`ProductType\,([domtype_{post\text{-}id}, rtype]),$
.22 $\text{BOOLEAN})$,
.23 $post\text{-}ifd = mk\text{-}OAS`ExplFnDef\,(post\text{-}id_{OAS}, [\,], type_{post\text{-}id}, post\text{-}id_{OAS},$
.24 $parms_{post\text{-}id} \curvearrowright [rid], fnpost, \mathsf{nil})$ in
.25 $ExplFnDef2ExplFnDef\,(post\text{-}ifd, \{\mapsto\})$ in
.26 $mk\text{-}(\{id_{CAS} \mapsto implfndef_{CAS}\},$
.27 $\{pre\text{-}id_{CAS} \mapsto pre\text{-}implfndef_{CAS}\} \cup \{post\text{-}id_{CAS} \mapsto post\text{-}implfndef_{CAS}\})$
.28 pre let $mk\text{-}OAS`ImplFnDef\,(id, \text{-}, \text{-}, mk\text{-}OAS`IdType\,(rid, \text{-}), \text{-}, \text{-}) = ifd$ in
.29 $is\text{-}OAS`ValueId\,(id) \wedge is\text{-}OAS`ValueId\,(rid)$

annotations

.8-.15 Generate an explicit function for the pre-condition of the function with the name *pre-id*.

.18-.25 Generate an explicit function for the post-condition of the function with the name *post-id*.

.28 The pre-condition expresses the requirement that the function name and the result identifier have no reserved prefix.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

end annotations

12.3.2.5 Operation Definitions

functions

78.0 $OperationDefinitions2ExplOpDefMap$:
.1 $OAS`OperationDef^{+} \times CAS`TypeDef \rightarrow$
.2 $(CAS`Id \xrightarrow{m} CAS`ExplOpDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $OperationDefinitions2ExplOpDefMap\,(ods, \sigma) \triangleq$
.4 let $defs = \{OperationDef2ExplOpDefMap\,(od, \sigma) \mid$
.5 $od : OAS`ExplOprtDef \cdot od \in \text{elems } ods\}$ in
.6 $mk\text{-}(\text{merge } \{op \mid mk\text{-}(op, \text{-}) \in defs\}, \text{merge } \{op_q \mid mk\text{-}(\text{-}, op_q) \in defs\})$
.7 pre let $defs = \{OperationDef2ExplOpDefMap\,(od, \sigma) \mid$
.8 $od : OAS`ExplOprtDef \cdot od \in \text{elems } ods\}$ in
.9 $\forall\, mk\text{-}(tdm, tdm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(tdm', \text{-}) \in defs \setminus \{mk\text{-}(tdm, tdm_q)\} \cdot \text{dom } tdm \cap \text{dom } tdm' = \{\,\}$

annotations

.7 The pre-condition ensures that all explicit operation identifiers are unique.

end annotations ;

79.0 $OperationDef2ExplOpDefMap$:
.1 $OAS`ExplOprtDef \times CAS`TypeDef \rightarrow$
.2 $(CAS`Id \xrightarrow{m} CAS`ExplOpDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $OperationDef2ExplOpDefMap\,(eod, \sigma) \triangleq$
.4 let $mk\text{-}OAS`ExplOprtDef\,(id, optype, \text{-}, params, body, oppre) = eod$ in
.5 let $id_{CAS} = Id2Id\,(id, \{\mapsto\})$,
.6 $explopdef_{CAS} = ExplOprtDef2ExplOpDef\,(eod, \sigma)$,
.7 $pre\text{-}id_{CAS} = \text{let } mk\text{-}OAS`ValueId(pre\text{-}id) = id \text{ in}$
.8 $Id2Id\,(mk\text{-}OAS`PreId(pre\text{-}id), \{\mapsto\})$,
.9 $pre\text{-}explopdef_{CAS} =$
.10 let $pre\text{-}id_{OAS} = \text{let } mk\text{-}OAS`ValueId(pre\text{-}id) = id \text{ in } mk\text{-}OAS`PreId(pre\text{-}id)$,
.11 $type_{pre\text{-}id} = \text{if } optype.dom = \text{UNITTYPE}$
.12 then $mk\text{-}OAS`TotalFnType(\sigma, \text{BOOLEAN})$
.13 else $mk\text{-}OAS`TotalFnType$
.14 $(mk\text{-}OAS`ProductType\,([optype.dom, \sigma]), \text{BOOLEAN})$,
.15 $parms_{pre\text{-}id} =$
.16 let $scomps = [mk\text{-}OAS`PatternId\,(mk\text{-}OAS`ValueId\,(\sigma.shape.fields(i).sel)) \mid$
.17 $i \in \text{inds } \sigma.shape.fields]$,
.18 $mk\text{-}OAS`ValueId(tag) = \sigma.shape.id$ in
.19 if $params = [\,]$ then $mk\text{-}OAS`RecordPattern(tag, scomps)$
.20 else $mk\text{-}OAS`TuplePattern\,([params, mk\text{-}OAS`RecordPattern(tag, scomps)])$,
.21 $body_{pre\text{-}id} = \text{if } oppre = \text{nil then } mk\text{-}OAS`BoolLit\,(\text{true}) \text{ else } oppre$,
.22 $pre\text{-}eod = mk\text{-}OAS`ExplFnDef\,(pre\text{-}id_{OAS}, [\,], type_{pre\text{-}id}, pre\text{-}id_{OAS},$
.23 $parms_{pre\text{-}id}, body_{pre\text{-}id}, \text{nil})$ in
.24 $ExplFnDef2ExplFnDef\,(pre\text{-}eod, \{\mapsto\})$ in
.25 $mk\text{-}(\{id_{CAS} \mapsto explopdef_{CAS}\}, \{pre\text{-}id_{CAS} \mapsto pre\text{-}explopdef_{CAS}\})$
.26 pre let $mk\text{-}OAS`ExplOprtDef\,(id, optype, id_repeated, params, \text{-}, \text{-}) = eod$ in
.27 $id = id_repeated \wedge is\text{-}OAS`ValueId\,(id) \wedge NumberOfParsMatchesOpType\,(params, optype)$

annotations

.9-.24 Generate an explicit function for the pre-condition of the operation with the name *pre-id*.

.26 The pre-condition ensures that the operation names in the heading are the same, that the operation name has no reserved prefix, and that the parameter profile matches the type profile of the operation.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

80.0 $NumberOfParsMatchesOpType : OAS`Parameters \times OAS`OpType \to \mathbb{B}$

.1 $NumberOfParsMatchesOpType\,(params, mk\text{-}OAS`OpType\,(opdom, \text{-})) \triangleq$

.2 cases len $params$:

.3 $0 \to opdom = \text{UNITTYPE}$,

.4 $1 \to \neg\, is\text{-}OAS`ProductType\,(opdom)$,

.5 others $\to is\text{-}OAS`ProductType\,(opdom) \wedge$

.6 let $mk\text{-}OAS`ProductType\,(factors) = opdom$ in

.7 len $params$ = len $factors$

.8 end

annotations

80.0 This function is used by function 65's pre-condition to ensure that in an explicit operation's heading the number of patterns in the patternlist equals the number of input parameter types.

end annotations ;

81.0 $ExplOprtDef2ExplOpDef : OAS`ExplOprtDef \times CAS`TypeDef \to CAS`ExplOpDef$

.1 $ExplOprtDef2ExplOpDef\,(eod, \sigma) \triangleq$

.2 let $mk\text{-}OAS`ExplOprtDef\,(id, optype, \text{-}, params, body, oppre) = eod$ in

.3 let $heading_{CAS} = MakeOpHeading\,(optype, oppre, [\,], \sigma)$,

.4 $body_{CAS} = Stmt2Stmt\,(body)$ in

.5 $mk\text{-}CAS`ExplOpDef\,(heading_{CAS}, body_{CAS})$

annotations

81.0 An explicit operation consists of a heading and body in the CAS, and these parts are constructed by *MakeOpHeading*, and *Stmt2Stmt* respectively.

end annotations ;

82.0 $MakeOpHeading$:

.1 $OAS`OpType \times [OAS`Expr] \times OAS`VarInf^* \times CAS`TypeDef \to CAS`OpHeading$

.2 $MakeOpHeading\,(mk\text{-}OAS`OpType\,(dom, rng), pre, extrdwr, \sigma) \triangleq$

.3 let $mk\text{-}(\text{-}, id_{CAS}) = GetUnusedId\,()$,

.4 $par_{CAS} = mk\text{-}CAS`Par(id_{CAS}, Type2Type\,(dom))$,

.5 $restp_{CAS} = Type2Type\,(rng)$,

.6 $\rho = \{mk\text{-}OAS`ValueId(\sigma.shape.fields(i).sel) \mapsto$

.7 $mk\text{-}CAS`OldId(\sigma.shape.fields(i).sel) \mid$

.8 $i \in \text{inds } stype.shape.fields\}$,

.9 $pre_{CAS} = OExpr2TrueOrExpr\,(pre, \rho)$,

.10 $ext_{CAS} = VarInfs2ExtVarInfs\,(extrdwr, \sigma)$ in

.11 $mk\text{-}CAS`OpHeading\,(par_{CAS}, restp_{CAS}, pre_{CAS}, ext_{CAS})$;

83.0 $VarInfs2ExtVarInfs : OAS`VarInf^* \times CAS`TypeDef \to CAS`ExtVarInf\text{-set}$

.1 $VarInfs2ExtVarInfs\,(extrdwr, \sigma) \triangleq$

.2 $\bigcup\{\{$let $mk\text{-}OAS`VarInf\,(mode, vars, type) = extrdwr\,(j)$,

.3 $rest = Id2Id\,(vars\,(i), \{\mapsto\})$,

.4 tp = if $type$ = nil

.5 then let $tp : CAS`Type$ be st $mk\text{-}CAS`Field\,(rest, tp) \in$ elems $\sigma.shape.fields$ in tp

.6 else $Type2Type\,(type)$ in

.7 $mk\text{-}CAS`ExtVarInf\,(mode, rest, tp) \mid i \in \text{inds } extrdwr\,(j).vars\} \mid j \in \text{inds } extrdwr\}$

.8 pre $\forall\, id \in \bigcup\{\{extrdwr\,(j) \cdot vars\,(i) \mid i \in \text{inds } extrdwr\,(j) \cdot vars\} \mid j \in \text{inds } extrdwr\} \cdot$

.9 $is\text{-}OAS`ValueId\,(id)$

annotations

83.0 When this function is called with $OAS`VarInf^*$ equal to the empty sequence (ie, from an explicit operation or an implicit operation without externals), the body of this function becomes meaningless, hence the empty sequence is returned.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.4-.5 If there is no type expression in the external variable item under test, the type of this external variable when translated to the CAS must be the same as the corresponding variable definition in the state.

.8 The pre-condition expresses the requirement that the external components have no reserved prefixes.

end annotations ;

84.0 $OperationDefinitions2ImplOpDefMap$:
.1 $OAS`OperationDef^{+} \times CAS`TypeDef \to$
.2 $(CAS`Id \xrightarrow{m} CAS`ImplOpDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $OperationDefinitions2ImplOpDefMap\,(ods, \sigma) \triangleq$
.4 let $defs = \{OperationDef2ImplOpDefMap\,(od, \sigma) \mid$
.5 $od : OAS`ImplOprtDef \cdot od \in \mathsf{elems}\ ods\}$ in
.6 $mk\text{-}(\mathsf{merge}\ \{op \mid mk\text{-}(op, \text{-}) \in defs\}, \mathsf{merge}\ \{op_q \mid mk\text{-}(\text{-}, op_q) \in defs\})$
.7 pre let $defs = \{OperationDef2ImplOpDefMap\,(od, \sigma) \mid$
.8 $od : OAS`ImplOprtDef \cdot od \in \mathsf{elems}\ ods\}$ in
.9 $\forall\, mk\text{-}(tdm, tdm_q) \in defs \cdot$
.10 $\forall\, mk\text{-}(tdm', \text{-}) \in defs \setminus \{mk\text{-}(tdm, tdm_q)\} \cdot$
.11 $\mathsf{dom}\ tdm \cap \mathsf{dom}\ tdm' = \{\}$

annotations

.7 The pre-condition ensures that all implicit operation identifiers are unique.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

85.0 $OperationDef2ImplOpDefMap$:
.1 $OAS`ImplOprtDef \times CAS`TypeDef \to$
.2 $(CAS`Id \xrightarrow{m} CAS`ImplOpDef) \times (CAS`Id \xrightarrow{m} CAS`ExplFnDef)$
.3 $OperationDef2ImplOpDefMap\,(iod, \sigma) \triangleq$
.4 let $mk\text{-}OAS`ImplOprtDef\,(id, params, residtype, opext, oppre, oppost, excps) = iod$,
.5 $\rho = \{mk\text{-}OAS`ValueId(\sigma.shape.fields(i).sel) \mapsto GetUnusedId\,() \mid$
.6 $i \in \text{inds}\ \sigma.shape.fields\}$ in
.7 let $id_{CAS} = Id2Id\,(id, \{\mapsto\})$,
.8 $implopdef_{CAS} = ImplOprtDef2ImplOpDef\,(iod, \sigma, \rho)$,
.9 $mk\text{-}(patlist, pattype) = SplitOPatTypePairList\,(params)$,
.10 $mk\text{-}(pre\text{-}id_{OAS}, pre\text{-}id_{CAS},$
.11 $post\text{-}id_{OAS}, post\text{-}id_{CAS}) =$ let $mk\text{-}OAS`ValueId(id_{OAS}) = id$ in
.12 $mk\text{-}(mk\text{-}OAS`PreId(id_{OAS}), Id2Id\,(pre\text{-}id_{OAS}, \{\mapsto\})$,
.13 $mk\text{-}OAS`PostId(id_{OAS}), Id2Id\,(post\text{-}id_{OAS}, \{\mapsto\}))$,
.14 $scomps = [mk\text{-}OAS`PatternId(mk\text{-}OAS`ValueId(\sigma.shape.fields(i).sel)) \mid$
.15 $i \in \text{inds}\ \sigma.shape.fields]$,
.16 $pre\text{-}implopdef_{CAS} =$
.17 let $pre\text{-}id_{OAS} =$ let $mk\text{-}OAS`ValueId(pre\text{-}id) = id$ in $mk\text{-}OAS`PreId(pre\text{-}id)$,
.18 $type_{pre\text{-}id} =$ if $pattype = \text{nil}$ then $mk\text{-}OAS`TotalFnType(\sigma, \text{BOOLEAN})$
.19 else $mk\text{-}OAS`TotalFnType\,(mk\text{-}OAS`ProductType([pattype, \sigma]), \text{BOOLEAN})$,
.20 $parms_{pre\text{-}id} =$ let $mk\text{-}OAS`ValueId(tag) = \sigma.shape.id$ in
.21 if $parms = [\,]$ then $mk\text{-}OAS`RecordPattern(tag, scomps)$
.22 else $mk\text{-}OAS`TuplePattern([parms, mk\text{-}OAS`RecordPattern(tag, scomps)])$,
.23 $body_{pre\text{-}id} =$ if $oppre = \text{nil}$ then $mk\text{-}OAS`BoolLit\,(\text{true})$ else $oppre$,
.24 $pre\text{-}iod = mk\text{-}OAS`ExplFnDef\,(pre\text{-}id_{OAS}, [\,], type_{pre\text{-}id}, pre\text{-}id_{OAS},$
.25 $parms_{pre\text{-}id}, body_{pre\text{-}id}, \text{nil})$ in
.26 $ExplFnDef2ExplFnDef\,(pre\text{-}iod, \{\mapsto\})$,
.27 $post\text{-}implopdef_{CAS} =$
.28 let $post\text{-}id_{OAS} =$ let $mk\text{-}OAS`ValueId(post\text{-}id) = id$ in $mk\text{-}OAS`PostId(post\text{-}id)$,
.29 $type_{post\text{-}id} =$ let $rtype =$ if $residtype = \text{nil}$ then $[\,]$ else $[residtype.type]$ in
.30 $mk\text{-}OAS`TotalFnType\,(mk\text{-}OAS`ProductType\,([pattype, \sigma, \sigma] \frown$
.31 $rtype), \text{BOOLEAN})$,
.32 $parms_{post\text{-}id} =$ let $mk\text{-}OAS`ValueId(tag) = \sigma.shape.id$ in
.33 let $scomps_old = [\rho(scomp(i)) \mid i \in \text{inds}\ scomp]$,
.34 $parms' = parms \frown [mk\text{-}OAS`RecordPattern(tag, scomps)] \frown$
.35 $[mk\text{-}OAS`RecordPattern(tag, scomps_old)]$ in
.36 if $residtype = \text{nil}$ then $parms'$
.37 else $parms' \frown [mk\text{-}OAS`PatternId(residtype.id)]$,
.38 $post\text{-}iod = mk\text{-}OAS`ExplFnDef\,(post\text{-}id_{OAS}, [\,], type_{post\text{-}id}, post\text{-}id_{OAS},$
.39 $parms_{post\text{-}id}, oppost, \text{nil})$ in
.40 $ExplFnDef2ExplFnDef\,(post\text{-}iod, \rho)$ in
.41 $mk\text{-}(\{id_{CAS} \mapsto implopdef_{CAS}\},$
.42 $\{pre\text{-}id_{CAS} \mapsto pre\text{-}implopdef_{CAS}\} \cup \{post\text{-}id_{CAS} \mapsto post\text{-}implopdef_{CAS}\})$
.43 pre let $mk\text{-}OAS`ImplOprtDef\,(id, \text{-}, mk\text{-}OAS`IdType\,(rid, \text{-}), \text{-}, \text{-}, \text{-}, \text{-}) = iod$ in
.44 $is\text{-}OAS`ValueId\,(id) \wedge is\text{-}OAS`ValueId\,(rid)$

annotations

.16-.26 Generate an explicit function for the pre-condition of the operation with the name *pre-id*.

.27-.40 Generate an explicit function for the post-condition of the operation with the name *post-id*.

.43 The pre-condition ensures that the operation name and the result identifier have no reserved prefix.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
86.0  ImplOprtDef2ImplOpDef : OAS`ImplOprtDef × CAS`TypeDef × Env → CAS`ImplOpDef
 .1   ImplOprtDef2ImplOpDef (iod, σ, ρ) ≜
 .2     let mk-OAS`ImplOprtDef (id, params, residtype, opext, oppre, oppost, excps) = iod in
 .3     let mk-(patlist, pattype) = SplitOPatTypePairList (params),
 .4         optype              = if residtype = nil
 .5                               then mk-OAS`OpType(pattype, UNITTYPE)
 .6                               else let mk-OAS`IdType (-, type) = residtype in
 .7                                    mk-OAS`OpType(pattype, type),
 .8         heading_CAS         = MakeOpHeading (optype, oppre, excps, σ),
 .9         body_CAS            = MakePostDisjExcps (residtype, oppost, excps, patlist, ρ) in
.10     mk-CAS`ImplOpDef (heading_CAS, body_CAS)
```

annotations

86.0 This function re-arranges and translates the constituent parts of an implicit operation into an implicit operation in the CAS ie, operation heading and operation body.

.4-.7 If the operation has a result type, the input parameters' types and the result type are combined to form an *OpType*.

end annotations ;

```
87.0  MakePostDisjExcps :
 .1       [OAS`IdType] × OAS`Expr × OAS`OExceptions × OAS`ParameterTypes × Env →
 .2       CAS`OpPost
 .3  MakePostDisjExcps (post, excps, patlist, ρ) ≜
 .4    let vid_CAS          = if residtype = nil then nil
 .5                           else let mk-OAS`IdType (id, -) = residtype,
 .6                                    id_CAS              = Id2Id (id, ρ) in
 .7                                mk-CAS`ValueId (id_CAS),
 .8        excps_CAS        = MakeDisjExcps (excps, ρ),
 .9        post_CAS         = Expr2Expr (post, {↦}),
.10        post_CAS-excps   = if excps_CAS = nil then post_CAS
.11                           else mk-CAS`BinaryExpr(post_CAS, OR, excps_CAS),
.12        postcond_CAS     = if patlist = []
.13                           then post_CAS-excps
.14                           else let pat    = if len patlist = 1
.15                                             then Pattern2Pattern (hd patlist, {↦})
.16                                             else let tpat = mk-OAS`TuplePattern (patlist) in
.17                                                  TuplePattern2TuplePattern (tpat, {↦}),
.18                                    v_CAS = ValueDef2ValDefMap (mk-OAS`ValueDef (pat, pattype, nid)) in
.19                                mk-CAS`LetExpr (v_CAS, {↦}, {↦}, post_CAS-excps) in
.20    mk-CAS`OpPost (vid_CAS, postcond_CAS)
```

annotations

.8 If the operation has an error list, convert it to the CAS via *MakeDisjExcps*.

.10 When there is an error list and a post-condition, the two expressions must be 'OR'ed together.

.12-.19 If there are no parameter types, return $post_{CAS}\text{-}excps$; otherwise generate a value definition from *patlist* and combine with $post_{CAS}\text{-}excps$ to form a let expression.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

88.0 $MakeDisjExcps : OAS`OExceptions \times Env \to [CAS`Expr]$

.1 $MakeDisjExcps\,(excps, \rho) \triangleq$

.2 let $mk\text{-}OAS`Error(\text{-}, cond, action) = \mathsf{hd}\ excps$ in

.3 let $left_{CAS} = Expr2Expr\,(cond, \rho)$,

.4 $right_{CAS} = Expr2Expr\,(action, \{\mapsto\})$,

.5 $error_{CAS} = mk\text{-}CAS`BinaryExpr(left_{CAS}, \text{AND}, right_{CAS})$ in

.6 if len $excps = 1$

.7 then $error_{CAS}$

.8 else $mk\text{-}CAS`BinaryExpr(error_{CAS}, \text{OR}, MakeDisjExcps\,(\mathsf{tl}\ excps, \rho))$

annotations

.5 If the expression $left_{CAS}$ is true, the expression $right_{CAS}$ must be true also. $error_{CAS}$ reflects this by using the binary operator AND.

.7 When there is more than one exception, each $error_{CAS}$ must be 'OR'ed together.

end annotations

12.3.3 Expressions

functions

89.0 $OExpr2TrueOrExpr : [OAS`Expr] \times Env \to CAS`Expr$

.1 $OExpr2TrueOrExpr\,(expr, \rho) \triangleq$

.2 if $expr = \mathsf{nil}$ then $mk\text{-}CAS`BoolLit\,(\mathsf{true})$

.3 else $Expr2Expr\,(expr, \rho)$;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
90.0  Expr2Expr : [OAS`Expr] × Env → [CAS`Expr]
 .1   Expr2Expr (expr, ρ) ≜
 .2     cases expr :
 .3       mk-OAS`BracketedExpr (expr)          → Expr2Expr (expr, ρ),
 .4       mk-OAS`LetExpr (-, -)                → LetExpr2LetExpr (expr, ρ),
 .5       mk-OAS`LetBeSTExpr (-, -, -)         → LetBeSTExpr2LetBeSTExpr (expr, ρ),
 .6       mk-OAS`DefExpr (-, -)                → DefExpr2DefExpr (expr, ρ),
 .7       mk-OAS`IfExpr (-, -, -, -)           → IfExpr2IfExpr (expr, ρ),
 .8       mk-OAS`CasesExpr (-, -, -)           → CasesExpr2CasesExpr (expr, ρ),
 .9       mk-OAS`PrefixExpr (-, -)             → PrefixExpr2UnaryExpr (expr, ρ),
 .10      mk-OAS`MapInverseExpr (-)            → MapInverseExpr2UnaryExpr (expr, ρ),
 .11      mk-OAS`BinaryExpr (-, -, -)          → BinaryExpr2BinaryExpr (expr, ρ),
 .12      mk-OAS`AllExpr (-, -)                → AllExpr2AllOrExistsExpr (expr, ρ),
 .13      mk-OAS`ExistsExpr (-, -)             → ExistsExpr2AllOrExistsExpr (expr, ρ),
 .14      mk-OAS`ExistsUniqueExpr (-, -)       → ExistsUniqueExpr2ExistsUniqueExpr (expr, ρ),
 .15      mk-OAS`IotaExpr (-, -)               → IotaExpr2IotaExpr (expr, ρ),
 .16      mk-OAS`SetEnumeration (-)            → SetEnumeration2SetEnumeration (expr, ρ),
 .17      mk-OAS`SetComprehension (-, -, -)    → SetComprehension2SetComprehension (expr, ρ),
 .18      mk-OAS`SetRange (-, -)               → SetRange2SetRange (expr, ρ),
 .19      mk-OAS`SeqEnumeration (-)            → SeqEnumeration2SeqEnumeration (expr, ρ),
 .20      mk-OAS`SeqComprehension (-, -, -)    → SeqComprehension2SeqComprehension (expr, ρ),
 .21      mk-OAS`SubSequence (-, -, -)         → SubSequence2SubSequence (expr, ρ),
 .22      mk-OAS`MapEnumeration (-)            → MapEnumeration2MapEnumeration (expr, ρ),
 .23      mk-OAS`MapComprehension (-, -, -)    → MapComprehension2MapComprehension (expr, ρ),
 .24      mk-OAS`TupleConstructor (-)          → TupleConstructor2TupleConstructor (expr, ρ),
 .25      mk-OAS`RecordConstructor (-, -)      → RecordConstructor2RecordConstructor (expr, ρ),
 .26      mk-OAS`RecordModifier (-, -)         → RecordModifier2RecordModifier (expr, ρ),
 .27      mk-OAS`Apply (-, -)                  → Apply2Apply (expr, ρ),
 .28      mk-OAS`FieldSelect (-, -)            → FieldSelect2FieldSelect (expr, ρ),
 .29      mk-OAS`FctTypeInst (-, -)            → FctTypeInst2FctTypeInst (expr, ρ),
 .30      mk-OAS`Lambda (-, -)                 → Lambda2Lambda (expr, ρ),
 .31      mk-OAS`IsDefTypeExpr (-, -)          → IsDefTypeExpr2IsExpr (expr, ρ),
 .32      mk-OAS`IsBasicTypeExpr (-, -)        → IsBasicTypeExpr2IsExpr (expr, ρ),
 .33      mk-OAS`NumLit (val)                  → mk-CAS`NumLit (val),
 .34      mk-OAS`BoolLit (val)                 → mk-CAS`BoolLit (val),
 .35      (is-OAS`NilLit (expr))               → mk-CAS`NilLit (),
 .36      mk-OAS`CharLit (val)                 → mk-CAS`CharLit (val),
 .37      mk-OAS`TextLit (val)                 → mk-CAS`TextLit
 .38                                                ([mk-CAS`CharLit (val(i)) | i ∈ inds val]),
 .39      mk-OAS`QuoteLit (val)                → mk-CAS`QuoteLit (val),
 .40      mk-OAS`Name (name)                   → Id2Id (name, ρ),
 .41      mk-OAS`OldName (name)                → OldName2OldId (name),
 .42      others                               → nil
 .43    end
```

annotations

.1 See clause 12.1.3.4 for a description of the use of argument ρ.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

91.0 $IsExpr : OAS`Expr \to \mathbb{B}$

.1 $IsExpr\,(expr) \triangleq$
.2 $is\text{-}OAS`BracketedExpr\,(expr) \lor is\text{-}OAS`LetExpr\,(expr) \lor$
.3 $is\text{-}OAS`LetBeSTExpr\,(expr) \lor is\text{-}OAS`DefExpr\,(expr) \lor$
.4 $is\text{-}OAS`IfExpr\,(expr) \lor is\text{-}OAS`CasesExpr\,(expr) \lor$
.5 $is\text{-}OAS`PrefixExpr\,(expr) \lor is\text{-}OAS`MapInverseExpr\,(expr) \lor$
.6 $is\text{-}OAS`BinaryExpr\,(expr) \lor is\text{-}OAS`AllExpr\,(expr) \lor$
.7 $is\text{-}OAS`ExistsExpr\,(expr) \lor is\text{-}OAS`ExistsUniqueExpr\,(expr) \lor$
.8 $is\text{-}OAS`IotaExpr\,(expr) \lor is\text{-}OAS`SetEnumeration\,(expr) \lor$
.9 $is\text{-}OAS`SetComprehension\,(expr) \lor is\text{-}OAS`SetRange\,(expr) \lor$
.10 $is\text{-}OAS`SeqEnumeration\,(expr) \lor is\text{-}OAS`SeqComprehension\,(expr) \lor$
.11 $is\text{-}OAS`SubSequence\,(expr) \lor is\text{-}OAS`MapEnumeration\,(expr) \lor$
.12 $is\text{-}OAS`MapComprehension\,(expr) \lor is\text{-}OAS`TupleConstructor\,(expr) \lor$
.13 $is\text{-}OAS`RecordConstructor\,(expr) \lor is\text{-}OAS`RecordModifier\,(expr) \lor$
.14 $is\text{-}OAS`Apply\,(expr) \lor is\text{-}OAS`FieldSelect\,(expr) \lor$
.15 $is\text{-}OAS`FctTypeInst\,(expr) \lor is\text{-}OAS`Lambda\,(expr) \lor$
.16 $is\text{-}OAS`IsDefTypeExpr\,(expr) \lor is\text{-}OAS`IsBasicTypeExpr\,(expr) \lor$
.17 $is\text{-}OAS`NumLit\,(expr) \lor is\text{-}OAS`BoolLit\,(expr) \lor$
.18 $is\text{-}OAS`NilLit\,(expr) \lor is\text{-}OAS`CharLit\,(expr) \lor$
.19 $is\text{-}OAS`TextLit\,(expr) \lor is\text{-}OAS`QuoteLit\,(expr) \lor$
.20 $is\text{-}OAS`Name\,(expr) \lor is\text{-}OAS`OldName\,(expr)$

12.3.3.1 Bracketed Expressions

This clause is intentionally left empty.

12.3.3.2 Local Binding Expressions

functions

92.0 $LetExpr2LetExpr : OAS`LetExpr \times Env \to CAS`LetExpr$

.1 $LetExpr2LetExpr\,(mk\text{-}OAS`LetExpr\,(localdefs, body), \rho) \triangleq$
.2 let $vals_{CAS} = \mathsf{merge}\,\{ValueDefinitions2ValDefMap\,(vds) \mid$
.3 $mk\text{-}OAS`ValueDefinitions\,(vds) \in \mathsf{elems}\ localdefs\},$
.4 $mk\text{-}(explfns_{CAS}, \text{-}) = \{FunctionDef2ExplFnDefMap\,(fd) \mid$
.5 $fd : OAS`ExplFnDef \cdot fd \in \mathsf{elems}\ localdefs \land \neg\, IsPolymorphic\,(fd)\},$
.6 $mk\text{-}(implfns_{CAS}, \text{-}) = \{FunctionDef2ImplFnDefMap\,(fd) \mid$
.7 $fd : OAS`ImplFnDef \cdot fd \in \mathsf{elems}\ localdefs \land \neg\, IsPolymorphic\,(fd)\},$
.8 $in_{CAS} = Expr2Expr\,(body, \rho)$ in
.9 $mk\text{-}CAS`LetExpr\,(vals_{CAS}, explfns_{CAS}, implfns_{CAS}, in_{CAS})$
.10 pre let $mk\text{-}(explfns_{CAS}, \text{-}) = \{FunctionDef2ExplFnDefMap\,(fd) \mid$
.11 $fd : OAS`ExplFnDef \cdot fd \in \mathsf{elems}\ localdefs \land \neg\, IsPolymorphic\,(fd)\},$
.12 $mk\text{-}(implfns_{CAS}, \text{-}) = \{FunctionDef2ImplFnDefMap\,(fd) \mid$
.13 $fd : OAS`ImplFnDef \cdot fd \in \mathsf{elems}\ localdefs \land \neg\, IsPolymorphic\,(fd)\}$ in
.14 $\forall\, id \in \mathsf{dom}\ explfns_{CAS} \cup \mathsf{dom}\ implfns_{CAS} \cdot is\text{-}OAS`ValueId\,(id)$

annotations

.2 Pull out all local value definitions from *localdefs*, and convert each one to the CAS.

.4 Pull out all local explicit function definitions from *localdefs*, and convert to the CAS.

.6 Pull out all local implicit function definitions from *localdefs*, and convert to the CAS.

.10 The pre-condition ensures that the function names do not use reserved prefixes.

end annotations ;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
93.0  LetBeSTExpr2LetBeSTExpr : OAS'LetBeSTExpr × Env → CAS'LetBeSTExpr
 .1   LetBeSTExpr2LetBeSTExpr (mk-OAS'LetBeSTExpr (bind, cond, body), ρ) ≜
 .2     let bind_CAS = Bind2Bind (bind, ρ),
 .3         st_CAS   = OExpr2TrueOrExpr (cond, ρ),
 .4         in_CAS   = Expr2Expr (body, ρ) in
 .5     mk-CAS'LetBeSTExpr (bind_CAS, st_CAS, in_CAS) ;

94.0  DefExpr2DefExpr : OAS'DefExpr × Env → CAS'DefExpr
 .1   DefExpr2DefExpr (mk-OAS'DefExpr (defs, body), ρ) ≜
 .2     let mk-OAS'DefBind (lhs, rhs) = hd defs in
 .3     let lhs_CAS = PatternBind2Pattern (lhs, ρ),
 .4         rhs_CAS = Expr2Expr (rhs, ρ),
 .5         in_CAS  = if len defs = 1 then Expr2Expr (body, ρ)
 .6                   else DefExpr2DefExpr (mk-OAS'DefExpr (tl defs, body), ρ) in
 .7     mk-CAS'DefExpr (lhs_CAS, rhs_CAS, in_CAS)
```

12.3.3.3 Conditional Expressions

functions

```
95.0  IfExpr2IfExpr : OAS'IfExpr × Env → CAS'IfExpr
 .1   IfExpr2IfExpr (mk-OAS'IfExpr (test, cons, elsifaltn, altn), ρ) ≜
 .2     let test_CAS  = Expr2Expr (test, ρ),
 .3         cons_CAS  = Expr2Expr (cons, ρ),
 .4         altn_CAS  = if elsifaltn = [] then Expr2Expr (altn, ρ)
 .5                     else let mk-OAS'ElsifExpr (test_ElsifExpr, cons_ElsifExpr) = hd elsifaltn in
 .6                          IfExpr2IfExpr (mk-OAS'IfExpr (test_ElsifExpr, cons_ElsifExpr,
 .7                                                        tl elsifaltn, altn), ρ) in
 .8     mk-CAS'IfExpr (test_CAS, cons_CAS, altn_CAS) ;

96.0  CasesExpr2CasesExpr : OAS'CasesExpr × Env → CAS'CasesExpr
 .1   CasesExpr2CasesExpr (mk-OAS'CasesExpr (sel, altns, others), ρ) ≜
 .2     let sel_CAS    = Expr2Expr (sel, ρ),
 .3         altn_CAS   = conc [let mk-OAS'CaseAltn (match, body) = altns (i) in
 .4                            [CaseAltn2CaseAltn (match(j), body, ρ) | j ∈ inds match] |
 .5                              i ∈ inds altns],
 .6         others_CAS = if others = nil then []
 .7                      else let match_CAS = mk-CAS'PatternId (nil),
 .8                               body_CAS  = Expr2Expr (others, ρ) in
 .9                           [mk-CAS'CaseAltn (match_CAS, body_CAS)] in
.10     mk-CAS'CasesExpr (sel_CAS, altn_CAS ↷ others_CAS)
```

annotations

.3-.5 A sequence of case alternatives are constructed in the CAS. When there is more than one match per body each match is paired with that body and converted by *CaseAltn2CaseAltn*. It is worth noting that the sequence's length will always equal the number of matches.

.6 In the case of *others* the pattern must match all cases not covered by the case expression alternatives, therefore the pattern identifier must be *nil*.

.10 The others expression is added to the case expression alternatives.

end annotations ;

```
97.0  CaseAltn2CaseAltn : OAS'Pattern × OAS'Expr × Env → CAS'CaseAltn
 .1   CaseAltn2CaseAltn (match, body, ρ) ≜
 .2     let match_CAS = Pattern2Pattern (match, ρ),
 .3         body_CAS  = Expr2Expr (body, ρ) in
 .4     mk-CAS'CaseAltn (match_CAS, body_CAS)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.3.4 Unary Expressions

functions

98.0 $PrefixExpr2UnaryExpr : OAS`PrefixExpr \times Env \to CAS`Expr$

.1 $PrefixExpr2UnaryExpr\,(mk\text{-}OAS`PrefixExpr\,(op, expr), \rho) \triangleq$

.2 let $arg_{CAS} = Expr2Expr\,(expr, \rho)$ in

.3 $mk\text{-}CAS`UnaryExpr\,(op, arg_{CAS})$;

99.0 $MapInverseExpr2UnaryExpr : OAS`MapInverseExpr \times Env \to CAS`UnaryExpr$

.1 $MapInverseExpr2UnaryExpr\,(mk\text{-}OAS`MapInverseExpr\,(expr), \rho) \triangleq$

.2 let $arg_{CAS} = Expr2Expr\,(expr, \rho)$ in

.3 $mk\text{-}CAS`UnaryExpr\,(\text{MAPINVERSE}, arg_{CAS})$

12.3.3.5 Binary Expressions

functions

100.0 $BinaryExpr2BinaryExpr : OAS`BinaryExpr \times Env \to CAS`Expr$

.1 $BinaryExpr2BinaryExpr\,(mk\text{-}OAS`BinaryExpr\,(left, op, right), \rho) \triangleq$

.2 let $left_{CAS} = Expr2Expr\,(left, \rho)$,

.3 $right_{CAS} = Expr2Expr\,(right, \rho)$ in

.4 $mk\text{-}CAS`BinaryExpr\,(left_{CAS}, op, right_{CAS})$

12.3.3.6 Quantified Expressions

functions

101.0 $AllExpr2AllOrExistsExpr : OAS`AllExpr \times Env \to CAS`AllOrExistsExpr$

.1 $AllExpr2AllOrExistsExpr\,(mk\text{-}OAS`AllExpr\,(bind, pred), \rho) \triangleq$

.2 let $bind_{CAS} = BindList2BindSet\,(bind, \rho)$,

.3 $pred_{CAS} = Expr2Expr\,(pred, \rho)$ in

.4 $mk\text{-}CAS`AllOrExistsExpr\,(\text{ALL}, bind_{CAS}, pred_{CAS})$;

102.0 $ExistsExpr2AllOrExistsExpr : OAS`ExistsExpr \times Env \to CAS`AllOrExistsExpr$

.1 $ExistsExpr2AllOrExistsExpr\,(mk\text{-}OAS`ExistsExpr\,(bind, pred), \rho) \triangleq$

.2 let $bind_{CAS} = BindList2BindSet\,(bind, \rho)$,

.3 $pred_{CAS} = Expr2Expr\,(pred, \rho)$ in

.4 $mk\text{-}CAS`AllOrExistsExpr\,(\text{EXISTS}, bind_{CAS}, pred_{CAS})$;

103.0 $ExistsUniqueExpr2ExistsUniqueExpr :$

.1 $OAS`ExistsUniqueExpr \times Env \to CAS`ExistsUniqueExpr$

.2 $ExistsUniqueExpr2ExistsUniqueExpr\,(mk\text{-}OAS`ExistsUniqueExpr\,(bind, pred), \rho) \triangleq$

.3 let $bind_{CAS} = Bind2Bind\,(bind, \rho)$,

.4 $pred_{CAS} = Expr2Expr\,(pred, \rho)$ in

.5 $mk\text{-}CAS`ExistsUniqueExpr\,(bind_{CAS}, pred_{CAS})$

12.3.3.7 Iota Expression

functions

104.0 $IotaExpr2IotaExpr : OAS`IotaExpr \times Env \to CAS`IotaExpr$

.1 $IotaExpr2IotaExpr\,(mk\text{-}OAS`IotaExpr\,(bind, pred), \rho) \triangleq$

.2 let $bind_{CAS} = Bind2Bind\,(bind, \rho)$,

.3 $pred_{CAS} = Expr2Expr\,(pred, \rho)$ in

.4 $mk\text{-}CAS`IotaExpr\,(bind_{CAS}, pred_{CAS})$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.3.8 Set Expressions

functions

105.0 $SetEnumeration2SetEnumeration : OAS`SetEnumeration \times Env \to CAS`SetEnumeration$

.1 $SetEnumeration2SetEnumeration\,(mk\text{-}OAS`SetEnumeration\,(els), \rho) \triangleq$

.2 let $els_{CAS} = [Expr2Expr\,(els(i), \rho) \mid i \in \mathsf{inds}\ els]$ in

.3 $mk\text{-}CAS`SetEnumeration\,(els_{CAS})$;

106.0 $SetComprehension2SetComprehension :$

.1 $OAS`SetComprehension \times Env \to CAS`SetComprehension$

.2 $SetComprehension2SetComprehension\,(mk\text{-}OAS`SetComprehension\,(elem, bind, pred), \rho) \triangleq$

.3 let $elem_{CAS} = Expr2Expr\,(elem, \rho)$,

.4 $binds_{CAS} = BindList2BindSet\,(bind, \rho)$,

.5 $pred_{CAS} = OExpr2TrueOrExpr\,(pred, \rho)$ in

.6 $mk\text{-}CAS`SetComprehension\,(elem_{CAS}, binds_{CAS}, pred_{CAS})$;

107.0 $SetRange2SetRange : OAS`SetRange \times Env \to CAS`SetRange$

.1 $SetRange2SetRange\,(mk\text{-}OAS`SetRange\,(lb, ub), \rho) \triangleq$

.2 let $lb_{CAS} = Expr2Expr\,(lb, \rho)$,

.3 $ub_{CAS} = Expr2Expr\,(ub, \rho)$ in

.4 $mk\text{-}CAS`SetRange\,(lb_{CAS}, ub_{CAS})$

12.3.3.9 Sequence Expressions

functions

108.0 $SeqEnumeration2SeqEnumeration : OAS`SeqEnumeration \times Env \to CAS`SeqEnumeration$

.1 $SeqEnumeration2SeqEnumeration\,(mk\text{-}OAS`SeqEnumeration\,(els), \rho) \triangleq$

.2 let $els_{CAS} = [Expr2Expr\,(els\,(i), \rho) \mid i \in \mathsf{inds}\ els]$ in

.3 $mk\text{-}CAS`SeqEnumeration\,(els_{CAS})$;

109.0 $SeqComprehension2SeqComprehension :$

.1 $OAS`SeqComprehension \times Env \to CAS`SeqComprehension$

.2 $SeqComprehension2SeqComprehension\,(mk\text{-}OAS`SeqComprehension\,(elem, bind, pred), \rho) \triangleq$

.3 let $elem_{CAS} = Expr2Expr\,(elem, \rho)$,

.4 $binds_{CAS} = Bind2Bind\,(bind, \rho)$,

.5 $pred_{CAS} = OExpr2TrueOrExpr\,(pred, \rho)$ in

.6 $mk\text{-}CAS`SeqComprehension\,(elem_{CAS}, binds_{CAS}, pred_{CAS})$;

110.0 $SubSequence2SubSequence : OAS`SubSequence \times Env \to CAS`SubSequence$

.1 $SubSequence2SubSequence\,(mk\text{-}OAS`SubSequence\,(sequence, frompos, topos), \rho) \triangleq$

.2 let $sequence_{CAS} = Expr2Expr\,(sequence, \rho)$,

.3 $from_{CAS} = Expr2Expr\,(frompos, \rho)$,

.4 $to_{CAS} = Expr2Expr\,(topos, \rho)$ in

.5 $mk\text{-}CAS`SubSequence\,(sequence_{CAS}, from_{CAS}, to_{CAS})$

12.3.3.10 Map Expressions

functions

111.0 $MapEnumeration2MapEnumeration : OAS`MapEnumeration \times Env \to CAS`MapEnumeration$

.1 $MapEnumeration2MapEnumeration\,(mk\text{-}OAS`MapEnumeration\,(els), \rho) \triangleq$

.2 let $els_{CAS} = [Maplet2Maplet\,(els(i), \rho) \mid i \in \mathsf{inds}\ els]$ in

.3 $mk\text{-}CAS`MapEnumeration\,(els_{CAS})$;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
112.0  Maplet2Maplet : OAS'Maplet × Env → CAS'Maplet
   .1  Maplet2Maplet (mk-OAS'Maplet (mapdom, maprng), ρ) ≜
   .2    let dom_CAS = Expr2Expr (mapdom, ρ),
   .3        rng_CAS = Expr2Expr (maprng, ρ) in
   .4    mk-CAS'Maplet (dom_CAS, rng_CAS) ;

113.0  MapComprehension2MapComprehension :
   .1      OAS'MapComprehension × Env → CAS'MapComprehension
   .2  MapComprehension2MapComprehension (mk-OAS'MapComprehension (elem, bind, pred), ρ) ≜
   .3    let elem_CAS = Maplet2Maplet (elem, ρ),
   .4        bind_CAS = BindList2BindSet (bind, ρ),
   .5        pred_CAS = OExpr2TrueOrExpr (pred, ρ) in
   .6    mk-CAS'MapComprehension (elem_CAS, bind_CAS, pred_CAS)
```

12.3.3.11 Tuple Constructor Expression

functions

```
114.0  TupleConstructor2TupleConstructor : OAS'TupleConstructor × Env → CAS'TupleConstructor
   .1  TupleConstructor2TupleConstructor (mk-OAS'TupleConstructor (fields), ρ) ≜
   .2    let fields_CAS = [Expr2Expr (fields (i), ρ) | i ∈ inds fields] in
   .3    mk-CAS'TupleConstructor (fields_CAS)
```

12.3.3.12 Record Expressions

functions

```
115.0  RecordConstructor2RecordConstructor :
   .1      OAS'RecordConstructor × Env → CAS'RecordConstructor
   .2  RecordConstructor2RecordConstructor (mk-OAS'RecordConstructor (tag, fields), ρ) ≜
   .3    let tag_CAS    = Id2Id (tag, {↦}),
   .4        fields_CAS = [Expr2Expr (fields (i), ρ) | i ∈ inds fields] in
   .5    mk-CAS'RecordConstructor (tag_CAS, fields_CAS)
   .6  pre is-OAS'MkId(tag)
```

annotations

.6 The pre-condition ensures that the tag of the records starts with the reserved prefix '*mk-*'.

end annotations ;

```
116.0  RecordModifier2RecordModifier : OAS'RecordModifier × Env → CAS'RecordModifier
   .1  RecordModifier2RecordModifier (mk-OAS'RecordModifier (rec, modifiers), ρ) ≜
   .2    let rec_CAS       = Expr2Expr (rec, ρ),
   .3        modifiers_CAS = {Id2Id (modifier.field, ρ) ↦ Expr2Expr (modifier.new, ρ) |
   .4                         modifier ∈ elems modifiers} in
   .5    mk-CAS'RecordModifier (rec_CAS, modifiers_CAS)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.3.13 Apply Expressions

functions

```
117.0  Apply2Apply : OAS'Apply × Env → CAS'Apply
   .1  Apply2Apply (mk-OAS'Apply (fct, arg), ρ) ≜
   .2    let fct_CAS = Expr2Expr (fct, ρ),
   .3        arg_CAS = cases len arg :
   .4                    0      → mk-CAS'NilLit (),
   .5                    1      → Expr2Expr (hd arg, ρ),
   .6                    others → let fields = [Expr2Expr (arg (i), ρ) | i ∈ inds arg] in
   .7                             mk-CAS'TupleConstructor (fields)
   .8                  end in
   .9    mk-CAS'Apply (fct_CAS, arg_CAS)
```

annotations

.5 A function apply takes an expression representing an argument and applies it to a function, map, or sequence. The way in which these arguments are represented in the CAS is by a sequence, which is then made into a tuple constructor.

end annotations ;

```
118.0  FieldSelect2FieldSelect : OAS'FieldSelect × Env → CAS'FieldSelect
   .1  FieldSelect2FieldSelect (mk-OAS'FieldSelect (record, field), ρ) ≜
   .2    let record_CAS = Expr2Expr (record, ρ),
   .3        field_CAS  = Id2Id (field, ρ) in
   .4    mk-CAS'FieldSelect (record_CAS, field_CAS)
   .5  pre is-OAS'ValueId (field)
```

annotations

.5 The pre-condition ensures that the field name does not use a reserved prefix.

end annotations ;

```
119.0  FctTypeInst2FctTypeInst : OAS'FctTypeInst × Env → CAS'FctTypeInst
   .1  FctTypeInst2FctTypeInst (mk-OAS'FctTypeInst (polyfct, inst), ρ) ≜
   .2    let id_CAS     = Name2Id (polyfct, ρ),
   .3        tpinst_CAS = [Type2Type (inst (i)) | i ∈ inds inst] in
   .4    mk-CAS'FctTypeInst (id_CAS, tpinst_CAS)
```

12.3.3.14 Lambda Expression

functions

```
120.0  Lambda2Lambda : OAS'Lambda × Env → CAS'Lambda
   .1  Lambda2Lambda (mk-OAS'Lambda (parms, body), ρ) ≜
   .2    let pars     = [parms (i).pat | i ∈ inds parms],
   .3        parm     = if len pars = 1 then Pattern2Pattern (hd pars, ρ)
   .4                   else TuplePattern2TuplePattern (mk-OAS'TuplePattern (pars), ρ),
   .5        types    = [parms (i).type | i ∈ inds parms],
   .6        type_CAS = if len types = 1 then Type2Type (hd types)
   .7                   else ProductType2ProductType (mk-OAS'ProductType (types)) in
   .8    cases len parms :
   .9      1      → mk-CAS'Lambda (mk-CAS'Par (parm, type_CAS), Expr2Expr (body, ρ)),
  .10      others → let mk-(id_OAS, id_CAS) = GetUnusedId () in
  .11               mk-CAS'Lambda (mk-CAS'Par (id_CAS, type_CAS),
  .12                              MakeLetExpr (parm, nil, id_OAS, body, ρ))
  .13    end
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

annotations

.2 Pull out a sequence of patterns.

.5 Pull out a sequence of types.

.10-.12 When there is more than one parameter, a unique identifier is paired with the product type generated from the input parameters' types. This becomes the parameters of the lambda expression whose body is constructed from the let expression.

end annotations

12.3.3.15 Is Expressions

functions

```
121.0  IsDefTypeExpr2IsExpr : OAS'IsDefTypeExpr × Env → CAS'IsExpr
  .1   IsDefTypeExpr2IsExpr (mk-OAS'IsDefTypeExpr (deftype, arg), ρ) ≜
  .2     let id_OAS  = let mk-OAS'IsId (id) = deftype in mk-OAS'ValueId (id),
  .3         tag_CAS = Id2Id (id_OAS, {↦}),
  .4         arg_CAS = Expr2Expr (arg, ρ) in
  .5     mk-CAS'IsExpr (tag_CAS, arg_CAS)
  .6   pre is-OAS'IsId (deftype)
```

annotations

.6 The pre-condition ensures that the tag used the reserved prefix '*is-*'.

end annotations ;

```
122.0  IsBasicTypeExpr2IsExpr : OAS'IsBasicTypeExpr × Env → CAS'IsExpr
  .1   IsBasicTypeExpr2IsExpr (mk-OAS'IsBasicTypeExpr (type, arg), ρ) ≜
  .2     let tag_CAS = mk-CAS'BasicType(type),
  .3         arg_CAS = Expr2Expr (arg, ρ) in
  .4     mk-CAS'IsExpr (tag_CAS, arg_CAS)
```

12.3.3.16 Names

functions

```
123.0  Name2Id : OAS'Name × Env → CAS'Id
  .1   Name2Id (mk-OAS'Name(name), ρ) ≜
  .2     Id2Id (name, ρ) ;

124.0  OldName2OldId : OAS'ValueId → CAS'OldId
  .1   OldName2OldId (mk-OAS'ValueId(id)) ≜
  .2     mk-CAS'OldId (id)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.4 State Designators

functions

```
125.0  StateDesignator2StateDesignator : OAS'StateDesignator → CAS'StateDesignator
  .1   StateDesignator2StateDesignator (sd) ≜
  .2     cases sd :
  .3       mk-OAS'Name (name)              → Id2Id (name, {↦}),
  .4       mk-OAS'FieldRef (var, sel) →
  .5           let sd_CAS = StateDesignator2StateDesignator (var),
  .6               sel_CAS = Id2Id (sel, {↦}) in
  .7           mk-CAS'FieldRef (sd_CAS, sel_CAS),
  .8       mk-OAS'MapOrSeqRef (var, arg) →
  .9           let sd_CAS = StateDesignator2StateDesignator (var),
  .10              sel_CAS = Expr2Expr (arg, {↦}) in
  .11          mk-CAS'MapOrSeqRef (sd_CAS, sel_CAS)
  .12    end
```

annotations

.3 As the state designator can be a field reference, a further call to *StateDesignator2StateDesignator* is needed to bring out the field identifier.

.7 If the state designator is a map or sequence reference, a further call to *StateDesignator2StateDesignator* is needed to bring out the index.

end annotations

12.3.5 Statements

functions

```
126.0  Stmt2Stmt : OAS'Stmt → CAS'Stmt
  .1   Stmt2Stmt (stmt) ≜
  .2     cases stmt :
  .3       mk-OAS'LetStmt (-, -)              → LetStmt2LetStmt (stmt),
  .4       mk-OAS'LetBeSTStmt (-, -, -)       → LetBeSTStmt2LetBeSTStmt (stmt),
  .5       mk-OAS'DefStmt (-, -)              → DefStmt2DefStmt (stmt),
  .6       mk-OAS'BlockStmt (-, -)            → BlockStmt2BlockOrSequence (stmt),
  .7       mk-OAS'AssignStmt (-, -)           → AssignStmt2Assign (stmt),
  .8       mk-OAS'IfStmt (-, -, -, -)         → IfStmt2IfStmt (stmt),
  .9       mk-OAS'CasesStmt (-, -, -)         → CasesStmt2CasesStmt (stmt),
  .10      mk-OAS'SeqForLoop (-, -, -, -)     → SeqForLoop2SeqForLoop (stmt),
  .11      mk-OAS'SetForLoop (-, -, -)        → SetForLoop2SetForLoop (stmt),
  .12      mk-OAS'IndexForLoop (-, -, -, -, -) → IndexForLoop2IndexForLoop (stmt),
  .13      mk-OAS'WhileLoop (-, -)            → WhileLoop2WhileLoop (stmt),
  .14      mk-OAS'NonDetStmt (-)              → NonDetStmt2NonDetStmt (stmt),
  .15      mk-OAS'Call (-, -, -)              → Call2Call (stmt),
  .16      mk-OAS'ReturnStmt (-)              → ReturnStmt2ReturnStmt (stmt),
  .17      mk-OAS'AlwaysStmt (-, -)           → AlwaysStmt2Always (stmt),
  .18      mk-OAS'TrapStmt (-, -, -)          → TrapStmt2TrapStmt (stmt),
  .19      mk-OAS'RecTrapStmt (-, -)          → RecTrapStmt2RecTrapStmt (stmt),
  .20      mk-OAS'ExitStmt (-)                → ExitStmt2ExitStmt (stmt),
  .21      (ERROR)                            → ERROR,
  .22      (SKIP)                             → IDENT
  .23    end
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.5.1 Local Binding Statements

functions

```
127.0  LetStmt2LetStmt : OAS‘LetStmt → CAS‘LetStmt
  .1   LetStmt2LetStmt (mk-OAS‘LetStmt (letdefs, body)) ≜
  .2     let defs_CAS              = merge {ValueDefinitions2ValDefMap (vds) |
  .3                                        mk-OAS‘ValueDefinitions (vds) ∈ elems letdefs},
  .4         mk- (explfns_CAS, -) = {FunctionDef2ExplFnDefMap (fd) |
  .5                                 fd : OAS‘ExplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)},
  .6         mk- (implfns_CAS, -) = {FunctionDef2ImplFnDefMap (fd) |
  .7                                 fd : OAS‘ImplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)},
  .8         body_CAS             = Stmt2Stmt (body) in
  .9       mk-CAS‘LetStmt (defs_CAS, explfns_CAS, implfns_CAS, body_CAS)
 .10   pre let mk- (explfns_CAS, -) = {FunctionDef2ExplFnDefMap (fd) |
 .11                                  fd : OAS‘ExplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)},
 .12         mk- (implfns_CAS, -) = {FunctionDef2ImplFnDefMap (fd) |
 .13                                 fd : OAS‘ImplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)} in
 .14       ∀ id ∈ dom explfns_CAS ∪ dom implfns_CAS · is-OAS‘ValueId (id)
```

annotations

.2 Pull out all local value definitions from *letdefs*, and convert each one to the CAS.

.4 Pull out all local explicit function definitions from *letdefs*, and convert each one to the CAS.

.6 Pull out all local implicit function definitions from *letdefs*, and convert each one to the CAS.

.10 The pre-condition ensures that the function names do not use reserved prefixes.

end annotations ;

```
128.0  LetBeSTStmt2LetBeSTStmt : OAS‘LetBeSTStmt → CAS‘LetBeSTStmt
  .1   LetBeSTStmt2LetBeSTStmt (mk-OAS‘LetBeSTStmt (bind, cond, body)) ≜
  .2     let bind_CAS = Bind2Bind (bind, {↦}),
  .3         st_CAS   = OExpr2TrueOrExpr (cond, {↦}),
  .4         in_CAS   = Stmt2Stmt (body) in
  .5       mk-CAS‘LetBeSTStmt (bind_CAS, st_CAS, in_CAS) ;

129.0  DefStmt2DefStmt : OAS‘DefStmt → CAS‘DefStmt
  .1   DefStmt2DefStmt (mk-OAS‘DefStmt (eqdefs, body)) ≜
  .2     let mk-OAS‘EqDef (lhs, rhs) = hd eqdefs in
  .3     let lhs_CAS  = PatternBind2Pattern (lhs, {↦}),
  .4         rhs_CAS = if IsExpr (rhs) then Expr2Expr (rhs, {↦})
  .5                   else let mk-OAS‘Call(oprt, args, callst) = rhs in
  .6                        Call2Call (mk-OAS‘Call (oprt, args, callst)),
  .7         in_CAS  = if len eqdefs = 1 then Stmt2Stmt (body)
  .8                   else DefStmt2DefStmt (mk-OAS‘DefStmt (tl eqdefs, body)) in
  .9       mk-CAS‘DefStmt (lhs_CAS, rhs_CAS, in_CAS)
```

annotations

.4-6 The right hand side of the equals definition can contain a call to an operation. If this is the case, the right hand side must be evaluated by *Call2Call*.

.7 A definition preamble can contain many equals definitions, hence when there are more than one, *DefStmt2DefStmt* is recursively called.

end annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

12.3.5.2 Block and Assignment Statements

functions

```
130.0  BlockStmt2BlockOrSequence : OAS'BlockStmt → (CAS'Block | CAS'Sequence)
  .1   BlockStmt2BlockOrSequence (mk-OAS'BlockStmt (dcls, stmts)) ≜
  .2     if dcls = [ ]
  .3     then let stmts_CAS = [Stmt2Stmt (stmts (i)) | i ∈ inds stmts] in
  .4          mk-CAS'Sequence (stmts_CAS)
  .5     else let mk-OAS'AssignDef (var, tp, dclinit) = hd dcls in
  .6          let var_CAS  = Id2Id (var, {↦}),
  .7              init_CAS = if dclinit = nil then nil
  .8                         elseif IsExpr (dclinit)
  .9                         then Expr2Expr (dclinit, {↦})
  .10                        else Call2Call (dclinit),
  .11             tp_CAS   = Type2Type (tp),
  .12             body_CAS = BlockStmt2BlockOrSequence (mk-OAS'BlockStmt (tl dcls, stmts)) in
  .13         mk-CAS'Block (var_CAS, init_CAS, tp_CAS, body_CAS)
```

annotations

.2-.4 When there is no declaration identifier the conversion of a block statement is simply a case of constructing a sequence of the statements.

.7-.10 The declaration initial value can be the result of a call to an operation, or an assignment to an expression.

.12 If there is more than one declaration, a recursive call back to *BlockStatement2BlockOrSeqence* is needed to convert the rest of these declarations.

end annotations ;

```
131.0  AssignStmt2Assign : OAS'AssignStmt → CAS'Assign
  .1   AssignStmt2Assign (mk-OAS'AssignStmt (lhs, rhs)) ≜
  .2     let lhs_CAS = StateDesignator2StateDesignator (lhs),
  .3         rhs_CAS = if IsExpr(rhs) then Expr2Expr (rhs, {↦})
  .4                   else Call2Call (rhs) in
  .5     mk-CAS'Assign (lhs_CAS, rhs_CAS)
```

annotations

.2 The left hand side of an assign statement comprises of a reference to a part of the state known as a state designator.

.3 The right hand side of an assign statement is either an expression or the result of an operation call.

end annotations

12.3.5.3 Conditional Statements

functions

```
132.0  IfStmt2IfStmt : OAS'IfStmt → CAS'IfStmt
  .1   IfStmt2IfStmt (mk-OAS'IfStmt (test, cons, elsifaltn, altn)) ≜
  .2     let test_CAS  = Expr2Expr (test, {↦}),
  .3         cons_CAS  = Stmt2Stmt (cons),
  .4         altn_CAS  = if elsifaltn = [ ] then Stmt2Stmt (altn)
  .5                     else let mk-OAS'ElsifStmt (test_ElsifExpr, cons_ElsifExpr) = hd elsifaltn in
  .6                          IfStmt2IfStmt (mk-OAS'IfStmt (test_ElsifExpr, cons_ElsifExpr,
  .7                                                        tl elsifaltn, altn)) in
  .8     mk-CAS'IfStmt (test_CAS, cons_CAS, altn_CAS) ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
133.0  CasesStmt2CasesStmt : OAS'CasesStmt → CAS'CasesStmt
  .1   CasesStmt2CasesStmt (mk-OAS'CasesStmt (sel, altns, other)) ≜
  .2     let sel_CAS    = Expr2Expr (sel, {↦}),
  .3         altns_CAS  = conc [let mk-OAS'CaseStmtAltn (match, body) = altns (i) in
  .4                            [CaseStmtAltn2CaseStmtAltn (match(j), body) | j ∈ inds match] |
  .5                                i ∈ inds altns],
  .6         others_CAS = if other = nil then []
  .7                      else let match_CAS = mk-CAS'PatternId (nil),
  .8                               body_CAS  = Stmt2Stmt (other) in
  .9                           [mk-CAS'CaseStmtAltn (match_CAS, body_CAS)] in
  .10    mk-CAS'CasesStmt (sel_CAS, altns_CAS ⌢ others_CAS)
```

annotations

.3-.5 A sequence of case alternatives are constructed in the CAS. When there are more than one match per body each match is paired with that body and converted by *CaseStmtAltn2CaseStmtAltn*. It is worth noting that the sequence's length will always equal the number of matches.

.6 In the case of *others* the pattern must match all cases not covered by the case statement alternatives, therefore the pattern identifier must be *nil*.

.10 The others statement is added in to the case statement alternatives.

end annotations ;

```
134.0  CaseStmtAltn2CaseStmtAltn : OAS'Pattern × OAS'Stmt → CAS'CaseStmtAltn
  .1   CaseStmtAltn2CaseStmtAltn (match, body) ≜
  .2     let match_CAS = Pattern2Pattern (match, {↦}),
  .3         body_CAS  = Stmt2Stmt (body) in
  .4     mk-CAS'CaseStmtAltn (match_CAS, body_CAS)
```

12.3.5.4 Loop Statements

functions

```
135.0  SeqForLoop2SeqForLoop : OAS'SeqForLoop → CAS'SeqForLoop
  .1   SeqForLoop2SeqForLoop (mk-OAS'SeqForLoop (cv, dirn, forseq, body)) ≜
  .2     let cv_CAS   = PatternBind2Pattern (cv, {↦}),
  .3         dirn_CAS = if dirn = nil then FORWARDS
  .4                    else BACKWARDS,
  .5         seq_CAS  = Expr2Expr (forseq, {↦}),
  .6         body_CAS = Stmt2Stmt (body) in
  .7     mk-CAS'SeqForLoop (cv_CAS, dirn_CAS, seq_CAS, body_CAS) ;

136.0  SetForLoop2SetForLoop : OAS'SetForLoop → CAS'SetForLoop
  .1   SetForLoop2SetForLoop (mk-OAS'SetForLoop (cv, forset, body)) ≜
  .2     let cv_CAS   = Pattern2Pattern (cv, {↦}),
  .3         set_CAS  = Expr2Expr (forset, {↦}),
  .4         body_CAS = Stmt2Stmt (body) in
  .5     mk-CAS'SetForLoop (cv_CAS, set_CAS, body_CAS) ;

137.0  IndexForLoop2IndexForLoop : OAS'IndexForLoop → CAS'IndexForLoop
  .1   IndexForLoop2IndexForLoop (mk-OAS'IndexForLoop (cv, lb, ub, step, body)) ≜
  .2     let cv_CAS   = Id2Id (cv, {↦}),
  .3         lb_CAS   = Expr2Expr (lb, {↦}),
  .4         ub_CAS   = Expr2Expr (ub, {↦}),
  .5         by_CAS   = if step = nil then mk-CAS'NumLit (1)
  .6                    else Expr2Expr (step, {↦}),
  .7         body_CAS = Stmt2Stmt (body) in
  .8     mk-CAS'IndexForLoop (cv_CAS, lb_CAS, ub_CAS, by_CAS, body_CAS) ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
138.0  WhileLoop2WhileLoop : OAS‘WhileLoop → CAS‘WhileLoop
  .1   WhileLoop2WhileLoop (mk-OAS‘WhileLoop (test, body)) ≜
  .2     let test_CAS = Expr2Expr (test, {↦}),
  .3         body_CAS = Stmt2Stmt (body) in
  .4     mk-CAS‘WhileLoop (test_CAS, body_CAS)
```

12.3.5.5 NonDeterministic Statement

functions

```
139.0  NonDetStmt2NonDetStmt : OAS‘NonDetStmt → CAS‘NonDetStmt
  .1   NonDetStmt2NonDetStmt (mk-OAS‘NonDetStmt (stmts)) ≜
  .2     let stmts_CAS = [Stmt2Stmt (stmt(i)) | i ∈ inds stmts] in
  .3     mk-CAS‘NonDetStmt (stmts_CAS)
```

12.3.5.6 Call and Return Statements

functions

```
140.0  Call2Call : OAS‘Call → CAS‘Call
  .1   Call2Call (mk-OAS‘Call (mk-OAS‘Name (oprt), args, callst)) ≜
  .2     let oprt_CAS = Id2Id (oprt, {↦}),
  .3         arg_CAS  = cases len args :
  .4                      0      → mk-CAS‘NilLit (),
  .5                      1      → Expr2Expr (hd args, {↦}),
  .6                      others → let fields = [Expr2Expr (args (i), {↦}) | i ∈ inds args] in
  .7                               mk-CAS‘TupleConstructor (fields)
  .8                    end,
  .9         state_CAS = if callst = nil then nil
  .10                    else StateDesignator2StateDesignator (callst) in
  .11    mk-CAS‘Call (oprt_CAS, arg_CAS, state_CAS)
```

annotations

.3-.7 If there is more than one argument, the operation named by the identifier requires the list of arguments in the form of a tuple constructor.

.9 The only time *callst* = nil is when the specification is without a state, otherwise the state designator (which points to the part of the state being acted on by the call statement) is converted into the CAS by *StateDesignator2StateDesignator*.

end annotations ;

```
141.0  ReturnStmt2ReturnStmt : OAS‘ReturnStmt → CAS‘ReturnStmt
  .1   ReturnStmt2ReturnStmt (mk-OAS‘ReturnStmt (expr)) ≜
  .2     let expr_CAS = Expr2Expr (expr, {↦}) in
  .3     mk-CAS‘ReturnStmt (expr_CAS)
```

12.3.5.7 Exception Handling Statements

functions

```
142.0  AlwaysStmt2Always : OAS‘AlwaysStmt → CAS‘Always
  .1   AlwaysStmt2Always (mk-OAS‘AlwaysStmt (alwpost, body)) ≜
  .2     let post_CAS = Stmt2Stmt (alwpost),
  .3         body_CAS = BlockStmt2BlockOrSequence (body) in
  .4     mk-CAS‘Always (post_CAS, body_CAS) ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
143.0  TrapStmt2TrapStmt : OAS'TrapStmt → CAS'TrapStmt
  .1   TrapStmt2TrapStmt (mk-OAS'TrapStmt (pat, trappost, body)) ≜
  .2     let pat_CAS  = PatternBind2Pattern (pat, {↦}),
  .3         post_CAS = Stmt2Stmt (trappost),
  .4         body_CAS = BlockStmt2BlockOrSequence (body) in
  .5     mk-CAS'TrapStmt (pat_CAS, post_CAS, body_CAS) ;

144.0  RecTrapStmt2RecTrapStmt : OAS'RecTrapStmt → CAS'RecTrapStmt
  .1   RecTrapStmt2RecTrapStmt (mk-OAS'RecTrapStmt (traps, body)) ≜
  .2     let traps_CAS = {PatternBind2Pattern (match, {↦}) ↦ Stmt2Stmt (trappost) |
  .3                      mk-OAS'Trap(match, trappost) ∈ elems traps},
  .4         body_CAS = BlockStmt2BlockOrSequence (body) in
  .5     mk-CAS'RecTrapStmt (traps_CAS, body_CAS)
  .6   pre  card {match | mk-OAS'Trap(match, -) ∈ elems traps} =
  .7       len [let mk-OAS'Trap(match, -) = traps(i) in match |
  .8            i ∈ inds traps]
```

annotations

.6 The pre-condition ensures that there is no syntactic equivalence between any two guards.

end annotations ;

```
145.0  ExitStmt2ExitStmt : OAS'ExitStmt → CAS'Exit
  .1   ExitStmt2ExitStmt (mk-OAS'ExitStmt (expr)) ≜
  .2     let expr_CAS = Expr2Expr (expr, {↦}) in
  .3     mk-CAS'Exit (expr_CAS)
```

12.3.5.8 Identity Statement

This clause is intentionally left empty.

12.3.6 Patterns and Bindings

12.3.6.1 Patterns

functions

```
146.0  Pattern2Pattern : OAS'Pattern × Env → CAS'Pattern
  .1   Pattern2Pattern (pattern, ρ) ≜
  .2     cases pattern :
  .3       mk-OAS'PatternId (-)          → PatternId2PatternId (pattern),
  .4       mk-OAS'MatchVal (val)         → Expr2Expr (val, ρ),
  .5       mk-OAS'SetEnumPattern (-)     → SetEnumPattern2SetEnumPattern (pattern, ρ),
  .6       mk-OAS'SetUnionPattern (-, -) → SetUnionPattern2SetUnionPattern (pattern, ρ),
  .7       mk-OAS'SeqEnumPattern (-)     → SeqEnumPattern2SeqEnumPattern (pattern, ρ),
  .8       mk-OAS'SeqConcPattern (-, -)  → SeqConcPattern2SeqConcPattern (pattern, ρ),
  .9       mk-OAS'TuplePattern (-)       → TuplePattern2TuplePattern (pattern, ρ),
  .10      mk-OAS'RecordPattern (-, -)   → RecordPattern2RecordPattern (pattern, ρ)
  .11    end ;

147.0  PatternId2PatternId : OAS'PatternId → CAS'PatternId
  .1   PatternId2PatternId (mk-OAS'PatternId (name)) ≜
  .2     let id_CAS = Id2Id (name, {↦}) in
  .3     mk-CAS'PatternId (id_CAS) ;

148.0  SetEnumPattern2SetEnumPattern : OAS'SetEnumPattern × Env → CAS'SetEnumPattern
  .1   SetEnumPattern2SetEnumPattern (mk-OAS'SetEnumPattern (els), ρ) ≜
  .2     let els_CAS = [Pattern2Pattern (els(i), ρ) | i ∈ inds els] in
  .3     mk-CAS'SetEnumPattern (els_CAS) ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
149.0  SetUnionPattern2SetUnionPattern : OAS'SetUnionPattern × Env → CAS'SetUnionPattern
  .1   SetUnionPattern2SetUnionPattern (mk-OAS'SetUnionPattern (lp, rp), ρ) ≜
  .2     let lp_CAS = Pattern2Pattern (lp, ρ),
  .3         rp_CAS = Pattern2Pattern (rp, ρ) in
  .4     mk-CAS'SetUnionPattern (lp_CAS, rp_CAS) ;

150.0  SeqEnumPattern2SeqEnumPattern : OAS'SeqEnumPattern × Env → CAS'SeqEnumPattern
  .1   SeqEnumPattern2SeqEnumPattern (mk-OAS'SeqEnumPattern (els), ρ) ≜
  .2     let els_CAS = [Pattern2Pattern (els (i), ρ) | i ∈ inds els] in
  .3     mk-CAS'SeqEnumPattern (els_CAS) ;

151.0  SeqConcPattern2SeqConcPattern : OAS'SeqConcPattern × Env → CAS'SeqConcPattern
  .1   SeqConcPattern2SeqConcPattern (mk-OAS'SeqConcPattern (lp, rp), ρ) ≜
  .2     let lp_CAS = Pattern2Pattern (lp, ρ),
  .3         rp_CAS = Pattern2Pattern (rp, ρ) in
  .4     mk-CAS'SeqConcPattern (lp_CAS, rp_CAS) ;

152.0  TuplePattern2TuplePattern : OAS'TuplePattern × Env → CAS'TuplePattern
  .1   TuplePattern2TuplePattern (mk-OAS'TuplePattern (fields), ρ) ≜
  .2     let fields_CAS = [Pattern2Pattern (fields (i), ρ) | i ∈ inds fields] in
  .3     mk-CAS'TuplePattern (fields_CAS) ;

153.0  RecordPattern2RecordPattern : OAS'RecordPattern × Env → CAS'RecordPattern
  .1   RecordPattern2RecordPattern (mk-OAS'RecordPattern (id, fields), ρ) ≜
  .2     let tag_CAS    = Id2Id (id, {↦}),
  .3         fields_CAS = [Pattern2Pattern (fields (i), ρ) | i ∈ inds fields] in
  .4     mk-CAS'RecordPattern (tag_CAS, fields_CAS)
  .5   pre is-OAS'MkId(id)
```

12.3.6.2 Bindings

functions

```
154.0  PatternBind2Pattern : OAS'PatternBind × Env → CAS'Pattern
  .1   PatternBind2Pattern (pb, ρ) ≜
  .2     cases pb :
  .3       mk-OAS'SetBind (pat, set)     → let pat_CAS = Pattern2Pattern (pat, ρ),
  .4                                           set_CAS = Expr2Expr (set, ρ) in
  .5                                       mk-CAS'SetConstrPattern (pat_CAS, set_CAS),
  .6       mk-OAS'TypeBind (pat, type) → let pat_CAS = Pattern2Pattern (pat, ρ),
  .7                                           type_CAS = Type2Type (type) in
  .8                                       mk-CAS'TypeConstrPattern (pat_CAS, type_CAS),
  .9       others                        → Pattern2Pattern (pb, ρ)
 .10     end ;

155.0  Bind2Bind : OAS'Bind × Env → CAS'Bind
  .1   Bind2Bind (b, ρ) ≜
  .2     cases b :
  .3       mk-OAS'SetBind (pat, bindset) → let pat_CAS = Pattern2Pattern (pat, ρ),
  .4                                           set_CAS = Expr2Expr (bindset, ρ) in
  .5                                       mk-CAS'SetBind (pat_CAS, set_CAS),
  .6       mk-OAS'TypeBind (pat, type)   → let pat_CAS = Pattern2Pattern (pat, ρ),
  .7                                           type_CAS = Type2Type (type) in
  .8                                       mk-CAS'TypeBind (pat_CAS, type_CAS)
  .9     end ;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
156.0  BindList2BindSet : OAS'BindList × Env → CAS'Bind-set
  .1   BindList2BindSet (bs, ρ) ≜
  .2     ⋃{{Bind2Bind (if is-OAS'MultSetBind (bs(i))
  .3                   then let mk-OAS'MultSetBind (pats, expr) = bs(i) in
  .4                        mk-OAS'SetBind (pats(j), expr)
  .5                   else let mk-OAS'MultSetBind (pats, tp) = bs(i) in
  .6                        mk-OAS'TypeBind (pats(j), tp), rho) |
  .7          j ∈ inds bs(i).pats} | i ∈ inds bs}
  .8   pre  card ⋃{{Bind2Bind (if is-OAS'MultSetBind (bs(i))
  .9                   then let mk-OAS'MultSetBind (pats, expr) = bs(i) in
  .10                       mk-OAS'SetBind (pats(j), expr)
  .11                  else let mk-OAS'MultSetBind (pats, tp) = bs(i) in
  .12                       mk-OAS'TypeBind (pats(j), tp), rho) |
  .13         j ∈ inds bs(i).pats} | i ∈ inds bs} =
  .14     len conc [[Bind2Bind (if is-OAS'MultSetBind (bs(i))
  .15                  then let mk-OAS'MultSetBind (pats, expr) = bs(i) in
  .16                       mk-OAS'SetBind (pats(j), expr)
  .17                  else let mk-OAS'MultSetBind (pats, tp) = bs(i) in
  .18                       mk-OAS'TypeBind (pats(j), tp), rho) |
  .19         j ∈ inds bs(i).pats] | i ∈ inds bs]
```

annotations

.8 The pre-condition specifies that no duplicate bindings are allowed.

end annotations

12.3.7 Lexical Specification

12.3.7.1 General

This clause is intentionally left empty.

12.3.7.2 Characters

This clause is intentionally left empty.

12.3.7.3 Symbols

functions

```
157.0  Id2Id : [OAS'Id] × Env → [CAS'Id]
  .1   Id2Id (id, ρ) ≜
  .2     if id ∈ dom ρ
  .3     then ρ(id)
  .4     else cases id :
  .5           mk-OAS'ValueId(id) → mk-CAS'ValueId(id),
  .6           mk-OAS'PreId(id)   → mk-CAS'PreId(id),
  .7           mk-OAS'PostId(id)  → mk-CAS'PostId(id),
  .8           mk-OAS'InvId(id)   → mk-CAS'InvId(id),
  .9           mk-OAS'InitId(id)  → mk-CAS'InitId(id),
  .10          mk-OAS'MkId(id)    → mk-CAS'MkId(id),
  .11          mk-OAS'IsId(id)    → mk-CAS'IsId(id),
  .12          others             → nil
  .13        end
```

annotations

.2-12 If the identifier to be converted into the CAS is already present in the environment ρ, the CAS relation to the identifier is returned. If, on the other hand, the identifier is tagged, it is picked up by one of the case alternatives listed. This tagging ensures that there is no overlap of identifiers in the CAS'

end annotations

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

end *OAS2CAS*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

13 The Static Semantic Domains

This clause first introduces the notion of types and environments used in the definition of the Static Semantics. Then follows an introduction of well-formedness classifications and finally a number of relations between types are defined including a subtype and a disjointness relation.

13.1 Type Representations

The type representations defined below constitute an extended notion of types compared to the abstract syntactic class *Type*. With the extension, it is possible to express types of statements and operations in addition to the types of values which are expressible using VDM-SL syntax. The type CONT is used as the result type of operations and statements which do not return a value but continue normal sequencing. The types *RetTypeR* and *ExitTypeR* are used as the result type of statements which either return control to the calling operation or exit. The type VOID is used for indicating that no value is present in connection with return and exit statements and the type *OpTypeR* is used for typing of operations.

Moreover, certain value types which are also not expressible using VDM-SL syntax have been added. The UNITTYPE has been introduced as the type of the constant nil. Optional types $[T]$ are represented by union types: T | UNITTYPE. Types for the empty set, sequence and map has also been introduced. The type ANY is used in connection with value definitions without explicit type assertions. It denotes the type of all possible values.

```
1.0 TypeR = BasicTypeR | InvTypeR | QuoteTypeR |
 .1         CompositeTypeR | UnionTypeR | ProductTypeR |
 .2         SetTypeR | SeqTypeR | MapTypeR | FunctionTypeR |
 .3         TypeName | TypeVar |
 .4         RetTypeR | ExitTypeR | OpTypeR;
2.0 BasicTypeR = NATONE | NAT | INT | RAT | REAL |
 .1              BOOLEAN | CHAR | TOKEN |
 .2              EMPTYSET | EMPTYSEQ | EMPTYMAP |
 .3              UNITTYPE | ANY | CONT | VOID
```

Note that, in the abstract syntax, invariants are attached to type definitions rather than type expressions. Note also that type names and variables are as defined in the abstract syntax. The above definition of type representations allows for many types which are not expressible in VDM-SL syntax. Many of these types are also not needed in connection with the current definition of the static semantics. As an example, consider a type for sequences of operations. It is allowed by the above definition and it would be handled with no problems by the static semantics if it could be expressed in the VDM-SL syntax. For this reason, invariants excluding these natural extensions have not been written.

```
3.0 InvTypeR :: shape : TypeR
 .1             invariant : Invariant
```

Note that invariants have the same form as in the abstract syntax. Also note that the basic (non-composite) type representations are identical to the basic types of the abstract syntax.

```
4.0 QuoteTypeR :: lit : QuoteLit ;
5.0 CompositeTypeR :: id : Id
 .1                   fields : FieldR*
```

Notice that the dynamic semantic domains for composite types abstract from field names (selectors). In the static semantic type representations, this information is kept in order to allow for a detailed type analysis.

```
6.0 FieldR :: id : [Id]
 .1           type : TypeR ;
7.0 UnionTypeR :: summands : TypeR-set
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.1 inv $mk\text{-}UnionTypeR(s) \triangleq \mathsf{card}\ s > 0$

Unlike in the abstract syntax, union types are allowed to have just one summand. This makes it possible to cut down the number of special cases which must be considered in certain formulae.

8.0 $ProductTypeR :: factors : TypeR^*$

.1 inv $mk\text{-}ProductTypeR(fs) \triangleq \mathsf{len}\ fs \geq 2;$

9.0 $SetTypeR :: elemtp : TypeR$;

10.0 $SeqTypeR = Seq0TypeR \mid Seq1TypeR;$

11.0 $Seq0TypeR :: elemtp : TypeR$;

12.0 $Seq1TypeR :: elemtp : TypeR$;

13.0 $MapTypeR = GeneralMapTypeR \mid InjectiveMapTypeR;$

14.0 $GeneralMapTypeR :: mapdom : TypeR$

.1 $maprng : TypeR$;

15.0 $InjectiveMapTypeR :: mapdom : TypeR$

.1 $maprng : TypeR$;

16.0 $FunctionTypeR = PartialFnTypeR \mid TotalFnTypeR;$

17.0 $PartialFnTypeR :: fndom : DiscretionaryTypeR$

.1 $fnrng : TypeR$;

18.0 $TotalFnTypeR :: fndom : DiscretionaryTypeR$

.1 $fnrng : TypeR$

In the Outer Abstract Syntax, function definitions may include an assertion about the function defined being total. In general type expressions, however, only partial function types can be expressed. In the Static Semantics, total function types have been introduced to enable expression of the fact that a function is in fact total.

19.0 $DiscretionaryTypeR = TypeR$

Note that UNITTYPE when used as a Discretionary Type will be regarded as representing the domain type of functions with no arguments.

20.0 $RetTypeR :: val : TypeR$;

21.0 $ExitTypeR :: val : TypeR$;

22.0 $OpTypeR :: opstate : [TypeName]$

.1 $opdom : TypeR$

.2 $oprng : StmtTypeR$

13.1.1 Special Subclasses of Type Representations

23.0 $StmtTypeR = TypeR$

.1 inv $str \triangleq$ cases str :

.2 (CONT), $mk\text{-}RetTypeR(\text{-}), mk\text{-}ExitTypeR(\text{-}) \rightarrow \mathsf{true},$

.3 $mk\text{-}UnionTypeR(ts) \rightarrow \forall\, t \in ts \cdot inv\text{-}StmtTypeR(t),$

.4 others $\rightarrow$ false

.5 end;

24.0 $StateTypeR = TypeR$

.1 inv $t \triangleq is\text{-}CompositeTypeR(t) \vee$

.2 $is\text{-}InvTypeR(t) \wedge is\text{-}CompositeTypeR(t.shape)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

The type of the state (if present) is a composite type, possibly with an invariant, and with a field for each state variable.

13.2 Environments

The four components of an environment are, in turn: (1) the name of the state—if there is any— and this is also the name of the type of the state; (2) the type map, i.e., all type names and associated type expressions; (3) the visible names and their associated static semantic information; and (4) the visible (polymorphic) type variables.

25.0 $Env :: statename : [TypeName]$
.1 $typemap : TypeMap$
.2 $visibleenv : VisibleEnv$
.3 $typevars : TypeVar\text{-set}$
.4 inv $mk\text{-}Env(sn, tm, ve, -) \triangleq (sn \neq \mathsf{nil} \Rightarrow sn \in \mathsf{dom}\ tm \wedge Is[StateTypeR, TypeR](tm(sn))) \wedge$
.5 $\forall n \in \mathsf{dom}\ ve \cdot is\text{-}DomR(ve(n)) \Rightarrow n \in \mathsf{dom}\ tm$

If present, the state name may be looked up in the type map to get the type of the state.

26.0 $TypeMap = TypeName \xrightarrow{m} TypeR;$
27.0 $VisName = Name \mid OldName;$
28.0 $VisibleEnv = VisName \xrightarrow{m} StaticSemanticRep$
.1 inv $ve \triangleq \forall n \in \mathsf{dom}\ ve \cdot is\text{-}OldName(n) \Rightarrow is\text{-}VarR(ve(n))$

The visible environment is similar to the ENV in the dynamic semantics. However, rather than mapping the names to their real denotations, they are mapped to the static semantics representations of these denotations.

29.0 $StaticSemanticRep = ValR \mid DomR \mid PolyValR \mid VarR;$
30.0 $ValR :: type : TypeR$
.1 $wfclass : \Pi\ ;$
31.0 $DomR ::$ LOOKINTYPEMAP ;
32.0 $PolyValR :: typepars : TypeVar^{+}$
.1 $valr : ValR\ ;$
33.0 $VarR :: type : TypeR$

The empty environment is introduced as a constant:

34.0 $EmptyEnv : Env = mk\text{-}Env(\mathsf{nil}, \{\mapsto\}, \{\mapsto\}, \{\})$

13.2.1 Accessing Environments

35.0 $StateVarNames : Env \xrightarrow{t} Name\text{-set}$
.1 $StateVarNames\,(env) \triangleq$
.2 dom $StateVisEnv(env);$

36.0 $StateVisEnv : Env \xrightarrow{t} VisibleEnv$
.1 $StateVisEnv\,(env) \triangleq$
.2 if $env.statename = \mathsf{nil}$ then $\{\mapsto\}$
.3 else cases $env.typemap(env.statename)$:
.4 $mk\text{-}InvTypeR(mk\text{-}CompositeTypeR(\text{-}, fs), \text{-}), mk\text{-}CompositeTypeR(\text{-}, fs) \rightarrow$
.5 $\{mk\text{-}Name(id) \mapsto mk\text{-}VarR(type) \mid mk\text{-}FieldR(id, type) \in \mathsf{elems}\ fs \cdot id \neq \mathsf{nil}\}$
.6 end;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

37.0 $IsBoundTo : VisName \xrightarrow{t} StaticSemanticRep \rightarrow (Env \mid VisibleEnv) \rightarrow \mathbb{B}$

.1 $IsBoundTo\,(n)(ssr)(env) \triangleq$
.2 let $ve =$ if $Is[VisibleEnv, Env \mid VisibleEnv](env)$ then env else $env.visibleenv$ in
.3 $n \in$ dom $ve \wedge$
.4 $ssr = ve(n)$;

38.0 $IsBoundToValTypeR : \Pi \xrightarrow{t} VisName \rightarrow TypeR \rightarrow (Env \mid VisibleEnv) \rightarrow \mathbb{B}$

.1 $IsBoundToValTypeR\,(\pi)(n)(t)(env) \triangleq$
.2 $IsBoundTo(n)(mk\text{-}ValR(t, \pi))(env) \vee$
.3 $IsBoundTo(n)(mk\text{-}VarR(t))(env) \vee$
.4 $is\text{-}POS(\pi) \wedge IsBoundTo(n)(mk\text{-}ValR(t, NegWfClass(\pi)))(env)$;

39.0 $IsBoundToDomTypeR : Name \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsBoundToDomTypeR\,(n)(t)(env) \triangleq$
.2 $IsVisibleTypeName(n)(env) \wedge t = env.typemap(n)$;

40.0 $IsVisibleTypeVar : TypeVar \xrightarrow{t} Env \rightarrow \mathbb{B}$

.1 $IsVisibleTypeVar\,(tv)(env) \triangleq$
.2 $tv \in env.typevars$;

41.0 $IsVisibleTypeName : Name \xrightarrow{t} Env \rightarrow \mathbb{B}$

.1 $IsVisibleTypeName\,(n)(env) \triangleq$
.2 $IsBoundTo(n)(mk\text{-}DomR(\text{LOOKINTYPEMAP}))(env)$

13.2.2 Updating Environments

42.0 $UpdateVisEnv : VisibleEnv \xrightarrow{t} VisibleEnv \rightarrow VisibleEnv$

.1 $UpdateVisEnv\,(newdefs)(oldenv) \triangleq$
.2 $oldenv \dagger newdefs$;

43.0 $UpdateEnv : VisibleEnv \mid TypeVar\text{-set} \xrightarrow{t} Env \rightarrow Env$

.1 $UpdateEnv\,(new)(oldenv) \triangleq$
.2 if $Is[VisibleEnv, VisibleEnv \mid TypeVar](new)$
.3 then $\mu\,(oldenv, visibleenv \mapsto UpdateVisEnv(new)(oldenv.visibleenv))$
.4 else $\mu\,(oldenv, typevars \mapsto oldenv.typevars \cup new)$;

44.0 $RemoveFromEnv : Name\text{-set} \xrightarrow{t} Env \rightarrow Env$

.1 $RemoveFromEnv\,(names)(oldenv) \triangleq$
.2 $\mu\,(oldenv, visibleenv \mapsto names \lhd oldenv.visibleenv)$

13.3 Well-formedness Classifications

The well-formedness of specifications (and expressions, statements, etc.) is defined as a parameterised boolean function. The parameter is used for distinguishing between different kinds of well-formedness checks. The main distinction is between possible and definite well-formedness. Moreover, there is a distinction between a basic well-formedness analysis and different combinations of additional, more advanced analyses.

45.0 $\Pi = POS \mid DEF$;

46.0 $POS :: checkset : Check\text{-set}$;

47.0 $DEF :: checkset : Check\text{-set}$;

48.0 $Check =$ LOCALCONTEXT | TOTALFUNC | UNIONCLOSE

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Note that LOCALCONTEXT and TOTALFUNC only have a direct influence on the way that possible well-formedness is checked. However, since the checking of definite well-formedness in connection with patterns and partial functions depends on the checking of possible well-formedness, definite well-formedness may be affected indirectly by the inclusion of these checks. Likewise, UNIONCLOSE only has a direct influence on the checking of definite well-formedness.

```
49.0 NegWfClass : Π →t Π
  .1 NegWfClass (π) ≜
  .2   cases π :
  .3     mk-POS(cs) → mk-DEF(cs),
  .4     mk-DEF(cs) → mk-POS(cs)
  .5   end
```

13.4 Type Relations

The definitions of various type relationships in this clause are based on a notion of *TypeRelation* which is simply a finite set of type representation pairs:

```
50.0 TypeRelation = (TypeR × TypeR)-set
```

13.4.1 Subtypes

The problem of determining whether one type representation denotes a subset of the values denoted by another type representation is fundamental for type checking of VDM-SL specifications. Due to the expressive notion of types, the problem is undecidable in general. Therefore, two decidable variants of the subtype predicate are defined below: definite subtype which is sound and possible subtype which is complete. If the latter does not hold for two types then it may safely be concluded that they are not in subtype relationship to each other (due to the completeness). The subtype predicate is parameterised with a well-formedness classification in order to distinguish the two variants.

In some cases, a restricted form of the subtype predicate is needed. The restricted predicate holds only when the one type is a so-called *essential subtype* of the other. Except for a difference concerning invariant types, the essential and the full subtype predicates have similar definitions. Therefore, they are both expressed in terms of a single, parameterised predicate with a parameter for distinguishing between ESSENTIAL and FULL in addition to the parameter for distinguishing between possible and definite subtype checking.

```
51.0 RelationKind = FULL | ESSENTIAL

52.0 IsFullSubtype : Π →t TypeR × TypeR → Env → 𝔹
  .1 IsFullSubtype (π)(t1, t2)(env) ≜
  .2   Is-x-Subtype(π)(FULL)(t1, t2)(env);

53.0 IsEssentialSubtype : Π →t TypeR × TypeR → Env → 𝔹
  .1 IsEssentialSubtype (π)(t1, t2)(env) ≜
  .2   Is-x-Subtype(π)(ESSENTIAL)(t1, t2)(env)

54.0 Is-x-Subtype : Π →t RelationKind → TypeR × TypeR → Env → 𝔹
  .1 Is-x-Subtype (π)(kind)(t1, t2)(env) ≜
  .2   CheckProof(π)(kind)(DistributeTypeR(t1)(env), t2)({})({})(env)
```

A type representation t_1 is a subtype of another t_2 if it is possible to find a proof which establishes this. Before doing this, the smaller type is, however, expanded by distributing certain type constructs over unions when possible. Note that the types may be defined by recursion. Therefore, the approach taken is always to try establishing the subtype relationship by a combination of fixed point induction on the type names in t_1 and unfolding of the type names in t_2.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

What needs to be shown is that for any fixed point approximation to t_1 there exists an unfolding of t_2 which contains the approximation as a subtype.

There are two problems in this. First, a straightforward attempt at fixed point induction may fail due to too weak an induction hypothesis. Secondly, the existence of a sufficiently large unfolding of t_2 cannot be tested by simply unfolding names in t_2 until either the subtype relation holds or it can be refused due to structural differences with t_1. In some cases, such unfoldings could continue indefinitely. The auxiliary function *CheckProof* defined below handles the first problem by collecting a conjunction of induction hypotheses (the parameter *pos*) which will be sufficient for completing an induction proof. The second problem is avoided by only allowing an unfolding of t_2 if an identical goal is not already being pursued by such an unfolding (an additional parameter *neg* is introduced for accumulating these goals). Initially, there are no induction hypotheses or goals being established by unfoldings of t_2.

The explanation above focuses on the soundness of the (definite subtype) predicate. Considering completeness of the (possible subtype) predicate, the interesting property is the opposite: t_1 is *not* a subtype of t_2. It turns out that the above approach can also be used to establish this property by fixed point induction. In this case, however, the above mentioned parameters *pos* and *neg* exchange roles: the latter is used for collecting induction hypotheses regarding the negative subtype property while the former collects positive assumptions which are used to prevent nontermination by stopping unnecessary unfoldings.

55.0 $CheckProof$:
.1 $\quad \Pi \xrightarrow{t} RelationKind \to TypeR \times TypeR \to TypeRelation \to TypeRelation \to Env \to \mathbb{B}$
.2 $CheckProof\,(\pi)(kind)(t_1,t_2)(pos)(neg)(env) \triangleq$
.3 $\quad CheckProofBasic(\pi)(t_1,t_2)(env) \lor$
.4 $\quad CheckProofStructural(\pi)(kind)(t_1,t_2)(pos)(neg)(env) \lor$
.5 $\quad CheckProofName(\pi)(kind)(t_1,t_2)(pos)(neg)(env) \lor$
.6 $\quad CheckProofInv(\pi)(kind)(t_1,t_2)(pos)(neg)(env) \lor$
.7 $\quad CheckProofUnion(\pi)(kind)(t_1,t_2)(pos)(neg)(env) \lor$
.8 $\quad CheckProofDistUnion(\pi)(t_1,t_2)(env) \lor$
.9 $\quad CheckProofTypeVar(\pi)(t_1,t_2) \lor$
.10 $\quad Is\text{-}x\text{-}EmptyType(\pi)(kind)(t_1)(env)$

The above predicate verifies the correctness of a subtype proof (or disproof) given a set of positive and a set of negative subtype assumptions. Its definition is split into a number of cases corresponding to different choices of the two types.

56.0 $CheckProofBasic : \Pi \xrightarrow{t} TypeR \times TypeR \to Env \to \mathbb{B}$
.1 $CheckProofBasic\,(\pi)(t_1,t_2)(env) \triangleq$
.2 $\quad$ cases $mk\text{-}(t_1,t_2)$:
.3 $\quad\quad mk\text{-}((\text{RAT}),\text{-}) \to t_2 = \text{REAL},$
.4 $\quad\quad mk\text{-}((\text{INT}),\text{-}) \to t_2 \in \{\text{RAT},\text{REAL}\},$
.5 $\quad\quad mk\text{-}((\text{NAT}),\text{-}) \to t_2 \in \{\text{INT},\text{RAT},\text{REAL}\},$
.6 $\quad\quad mk\text{-}((\text{NATONE}),\text{-}) \to t_2 \in \{\text{NAT},\text{INT},\text{RAT},\text{REAL}\},$
.7 $\quad\quad mk\text{-}(\text{-},(\text{ANY})) \to$ true,
.8 $\quad\quad mk\text{-}((\text{EMPTYSET}),\text{-}) \to is\text{-}SetTypeR(t_2),$
.9 $\quad\quad mk\text{-}((\text{EMPTYSEQ}),\text{-}) \to is\text{-}Seq0TypeR(t_2),$
.10 $\quad\quad mk\text{-}((\text{EMPTYMAP}),\text{-}) \to is\text{-}GeneralMapTypeR(t_2) \lor is\text{-}InjectiveMapTypeR(t_2),$
.11 $\quad\quad mk\text{-}(\text{-},(\text{EMPTYSET})) \to is\text{-}POS(\pi) \land$
.12 $\quad\quad\quad\quad is\text{-}SetTypeR(t_1) \land IsEmptyType(\pi)(t_1.elemtp)(env),$
.13 $\quad\quad mk\text{-}(\text{-},(\text{EMPTYSEQ})) \to is\text{-}POS(\pi) \land$
.14 $\quad\quad\quad\quad is\text{-}Seq0TypeR(t_1) \land IsEmptyType(\pi)(t_1.elemtp)(env),$
.15 $\quad\quad mk\text{-}(\text{-},(\text{EMPTYMAP})) \to$
.16 $\quad\quad\quad is\text{-}POS(\pi) \land (is\text{-}GeneralMapTypeR(t_1) \lor is\text{-}InjectiveMapTypeR(t_1)) \land$
.17 $\quad\quad\quad (IsEmptyType(\pi)(t_1.mapdom)(env) \lor IsEmptyType(\pi)(t_1.maprng)(env)),$
.18 $\quad\quad$ others $\to t_1 = t_2$
.19 $\quad$ end

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Here, the basic subtype relationships involving type constants are established. That any type is a subtype of itself is also expressed in this definition.

```
57.0 CheckProofStructural :
  .1     Π ~t~> RelationKind → TypeR × TypeR → TypeRelation → TypeRelation → Env → 𝔹
  .2 CheckProofStructural (π)(kind)(t1, t2)(pos)(neg)(env) ≜
  .3   let check = λ t1 : TypeR, t2 : TypeR ·
  .4                 CheckProof(π)(kind)(t1, t2)(pos)(neg)(env) in
  .5   cases mk-(t1, t2) :
  .6     mk-(mk-SetTypeR(et1), mk-SetTypeR(et2)) → check(et1, et2),
  .7     mk-(mk-Seq0TypeR(et1), -) →
  .8         check(mk-UnionTypeR({EMPTYSEQ, mk-Seq1TypeR(et1)}), t2),
  .9     mk-(-, mk-Seq0TypeR(et2)) →
  .10        check(t1, mk-UnionTypeR({EMPTYSEQ, mk-Seq1TypeR(et2)})),
  .11    mk-(mk-Seq1TypeR(et1), mk-Seq0TypeR(et2)) → check(et1, et2),
  .12    mk-(mk-Seq1TypeR(et1), mk-Seq1TypeR(et2)) → check(et1, et2),
  .13    mk-(mk-ProductTypeR(ts1), mk-ProductTypeR(ts2)) →
  .14        len ts1 = len ts2 ∧ ∀ i ∈ inds ts1 · check(ts1(i), ts2(i)),
  .15    mk-(mk-CompositeTypeR(n1, fs1), mk-CompositeTypeR(n2, fs2)) →
  .16        n1 = n2 ∧ len fs1 = len fs2 ∧
  .17        ∀ i ∈ inds fs1 · fs1(i).id = fs2(i).id ∧ check(fs1(i).type, fs2(i).type),
  .18    mk-(mk-PartialFnTypeR(at1, rt1), mk-PartialFnTypeR(at2, rt2)) →
  .19        check(rt1, rt2) ∧ check(at1, at2) ∧ (check(at2, at1) ∨ is-POS(π)),
  .20    mk-(mk-TotalFnTypeR(at1, rt1), mk-PartialFnTypeR(at2, rt2)) →
  .21        check(rt1, rt2) ∧ check(at1, at2) ∧ (check(at2, at1) ∨ is-POS(π)),
  .22    mk-(mk-TotalFnTypeR(at1, rt1), mk-TotalFnTypeR(at2, rt2)) →
  .23        check(rt1, rt2) ∧ check(at1, at2) ∧ check(at2, at1),
  .24    mk-(mk-GeneralMapTypeR(at1, rt1), mk-GeneralMapTypeR(at2, rt2)) →
  .25        check(at1, at2) ∧ check(rt1, rt2),
  .26    mk-(mk-GeneralMapTypeR(at1, rt1), mk-InjectiveMapTypeR(at2, rt2)) →
  .27        check(at1, at2) ∧ check(rt1, rt2) ∧
  .28        IsOneValueType(π)(at1)(env),
  .29    mk-(mk-InjectiveMapTypeR(at1, rt1), mk-GeneralMapTypeR(at2, rt2)) →
  .30        check(at1, at2) ∧ check(rt1, rt2),
  .31    mk-(mk-InjectiveMapTypeR(at1, rt1), mk-InjectiveMapTypeR(at2, rt2)) →
  .32        check(at1, at2) ∧ check(rt1, rt2),
  .33    mk-(mk-QuoteTypeR(q1), mk-QuoteTypeR(q2)) → q1 = q2,
  .34    mk-(mk-RetTypeR(et1), mk-RetTypeR(et2)) → check(et1, et2),
  .35    mk-(mk-ExitTypeR(et1), mk-ExitTypeR(et2)) → check(et1, et2),
  .36    mk-(mk-OpTypeR(st1, at1, rt1), mk-OpTypeR(st2, at2, rt2)) →
  .37        check(rt1, rt2) ∧ check(at1, at2) ∧ check(at2, at1) ∧
  .38        check(st1, st2) ∧ check(st2, st1),
  .39    others → false
  .40  end
```

The above predicate verifies the correctness of a subtype proof regarding structural (composite) types.

Note the difference between subtyping of partial and total functions. This is due to the fact that a VDM-SL function in the Dynamic Semantics is represented by a total (mathematical) function on the base argument type (i.e. disregarding any invariants). Total VDM-SL functions are then represented by functions which map values in the argument type (fulfilling invariants, if any) to values in the result type; other values are mapped to $\perp$. Partial VDM-SL functions are represented by functions which, in addition, may map any argument value to $\perp$.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

58.0 *CheckProofName* :
.1 $\Pi \xrightarrow{t} RelationKind \to TypeR \times TypeR \to TypeRelation \to TypeRelation \to Env \to \mathbb{B}$
.2 $CheckProofName\,(\pi)(kind)(t_1, t_2)(pos)(neg)(env) \triangleq$
.3 $is\text{-}Name(t_1) \wedge$
.4 $(mk\text{-}(t_1, t_2) \in pos \vee$
.5 $\exists\, t : TypeR \cdot IsBoundToDomTypeR(t_1)(t)(env) \wedge$
.6 $CheckProof(\pi)(kind)(t, t_2)(pos \cup \{mk\text{-}(t_1, t_2)\})(neg)(env))$
.7 $\vee$
.8 $is\text{-}Name(t_2) \wedge$
.9 $\exists\, t : TypeR \cdot IsBoundToDomTypeR(t_2)(t)(env) \wedge$
.10 $mk\text{-}(t_1, t_2) \notin neg \wedge$
.11 $CheckProof(\pi)(kind)(t_1, t)(pos)(neg \cup \{mk\text{-}(t_1, t_2)\})(env)$

Here, the subtype relationships involving type names are established. When t_1 is the name of a recursively defined type, this approach may be seen as a way of planning a fixed point induction proof of the subtype relationship. When t_2 is the name of a recursively defined type, it is unfolded unless such an unfolding has already been done for the same goal. Similarly when rejecting possible subtype relationships, however with converse roles of t_1 and t_2.

59.0 *CheckProofInv* :
.1 $\Pi \xrightarrow{t} RelationKind \to TypeR \times TypeR \to TypeRelation \to TypeRelation \to Env \to \mathbb{B}$
.2 $CheckProofInv\,(\pi)(kind)(t_1, t_2)(pos)(neg)(env) \triangleq$
.3 cases $mk\text{-}(t_1, t_2)$:
.4 $mk\text{-}(\text{-}, mk\text{-}InvTypeR(et_2, \text{-})) \to$
.5 $CheckProof(\pi)(kind)(t_1, et_2)(pos)(neg)(env) \wedge is\text{-}POS(\pi)$,
.6 $mk\text{-}(mk\text{-}InvTypeR(et_1, \text{-}), \text{-}) \to$
.7 $CheckProof(\pi)(kind)(et_1, t_2)(pos)(neg)(env) \vee$
.8 $is\text{-}POS(\pi) \wedge kind = \text{FULL}$,
.9 others → false
.10 end

Here, the subtype relationships involving invariant types are established. The only difference between the full and the essential subtype relation appears when considering whether an invariant type $mk\text{-}InvTypeR(et_1, \text{-})$ possibly is a subtype of another type t_2 or not. Since the invariant potentially could eliminate all elements, leaving an empty type as result, the full, possible subtype relation must include this pair no matter what type t_2 is. The pair is, however, only included in the essential subtype relation if et_1 is a subtype of t_2. This repairs a weakness of the full subtype relation exemplified by the fact that the type T defined by

$T = \text{char} \mid \mathbb{N}$
inv $x \triangleq$ true

is accepted as a possible subtype of both CHAR and NAT even though these types are disjoint. The essential, possible subtype must, however, be used with care since it is only complete for a restricted set of type pairs.

60.0 *CheckProofUnion* :
.1 $\Pi \xrightarrow{t} RelationKind \to TypeR \times TypeR \to TypeRelation \to TypeRelation \to Env \to \mathbb{B}$
.2 $CheckProofUnion\,(\pi)(kind)(t_1, t_2)(pos)(neg)(env) \triangleq$
.3 $is\text{-}UnionTypeR(t_1) \wedge$
.4 $\forall\, t \in t_1.summands \cdot CheckProof(\pi)(kind)(t, t_2)(pos)(neg)(env)$
.5 $\vee$
.6 $is\text{-}UnionTypeR(t_2) \wedge$
.7 $\exists\, t \in t_2.summands \cdot CheckProof(\pi)(kind)(t_1, t)(pos)(neg)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

This function accepts the subtype relationships involving unions which can be established without considering the distributivity of other operators over union. Here, the case of the left-hand side (t_1) being a union can be established if all summands are subtypes of the right-hand side. The case of the right-hand side being a union can be established if the left-hand side is a subtype of one of the summands.

61.0 $CheckProofDistUnion : \Pi \xrightarrow{t} TypeR \times TypeR \to Env \to \mathbb{B}$
.1 $CheckProofDistUnion\,(\pi)(t_1, t_2)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi) \wedge$
.3 $\quad (is\text{-}ProductTypeR(t_1) \vee is\text{-}PartialFnTypeR(t_1) \vee is\text{-}TotalFnTypeR(t_1) \vee$
.4 $\quad\ is\text{-}GeneralMapTypeR(t_1) \vee is\text{-}InjectiveMapTypeR(t_1))$
.5 $\quad \wedge$
.6 $\quad IsEquivToUnionTypeR(\pi)(t_1)(env) \wedge$
.7 $\quad is\text{-}UnionTypeR(t_2) \wedge$
.8 $\quad \exists\, s_1, s_2 \in t_2.summands \cdot$
.9 $\qquad s_1 \neq s_2 \wedge IsOverlapping(\pi)(s_1, t_1)(env) \wedge IsOverlapping(\pi)(s_2, t_1)(env)$

Here, special attention is paid to the fact that some type operators may distribute over union types. It should be noted that the predicate is defined solely for the purpose of obtaining completeness of the possible subtype relation. Definite subtype relationships which could be established on account of distributivity will therefore not be recognised. If the first argument is a product type and it is possible that it may be rewritten as a union only on account of distributivity and, in addition, the second argument is a union which might be rewritten as such a product (and possibly some residual summands) then the subtype relationship may hold. Whether the second argument may possibly be rewritten into a product on account of distributivity is checked by looking among the summands of t_2 for at least two summands which overlap the product type. Function and map types which in special cases may distribute over union types (see the definition below of *IsEquivToUnionTypeR*) are treated similarly.

62.0 $CheckProofTypeVar : \Pi \xrightarrow{t} TypeR \times TypeR \to \mathbb{B}$
.1 $CheckProofTypeVar\,(\pi)(t_1, t_2) \triangleq$
.2 $\quad is\text{-}POS(\pi) \wedge (is\text{-}TypeVarId(t_1) \vee is\text{-}TypeVarId(t_2))$

If one of the types is a type variable, absolutely nothing can be concluded about the subtype relationship: it may possibly hold but we cannot safely conclude that it definitely holds.

63.0 $IsEquivToUnionTypeR : \Pi \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.1 $IsEquivToUnionTypeR\,(\pi)(t)(env) \triangleq$
.2 $\quad \neg\, IsEmptyType(NegWfClass(\pi))(t)(env) \wedge$
.3 $\quad$ cases t :
.4 $\qquad mk\text{-}Name(\text{-}) \to IsEquivToUnionTypeR(\pi)(env.typemap(t))(env),$
.5 $\qquad mk\text{-}UnionTypeR(\text{-}), mk\text{-}Seq0TypeR(\text{-}) \to$ true,
.6 $\qquad mk\text{-}ProductTypeR(ts) \to \exists\, i \in$ inds $ts \cdot IsEquivToUnionTypeR(\pi)(ts(i))(env),$
.7 $\qquad mk\text{-}PartialFnTypeR(at, rt), mk\text{-}TotalFnTypeR(at, rt) \to$
.8 $\qquad\quad is\text{-}POS(\pi) \wedge IsFiniteTypeR(\pi)(at)(env) \wedge IsEquivToUnionTypeR(\pi)(rt)(env),$
.9 $\qquad mk\text{-}GeneralMapTypeR(at, rt) \to$
.10 $\qquad\quad is\text{-}POS(\pi) \wedge IsFiniteTypeR(\pi)(at)(env) \wedge IsEquivToUnionTypeR(\pi)(rt)(env),$
.11 $\qquad mk\text{-}InjectiveMapTypeR(at, rt) \to$
.12 $\qquad\quad is\text{-}POS(\pi) \wedge$
.13 $\qquad\quad (IsFiniteTypeR(\pi)(at)(env) \wedge IsEquivToUnionTypeR(\pi)(rt)(env) \vee$
.14 $\qquad\quad\ IsFiniteTypeR(\pi)(rt)(env) \wedge IsEquivToUnionTypeR(\pi)(at)(env)),$
.15 $\qquad mk\text{-}InvTypeR(t_1, \text{-}) \to IsEquivToUnionTypeR(\pi)(t_1)(env),$
.16 $\qquad mk\text{-}TypeVarId(\text{-}) \to is\text{-}POS(\pi),$
.17 $\qquad$ others $\to$ false
.18 $\quad$ end

This function is used for checking whether a product, function or map type t can be rewritten as a union of other

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

types solely on account of distributivity over union. Note, however, that in this context, a *Seq*0 type is considered a union of EMPTYSEQ and a *Seq*1 type.

13.4.2 Overlapping Subtypes, Disjoint Types and Overlapping Types

In most cases, a restricted form of the subtype predicate is useful. The restricted predicate holds only when the one type is a subtype of the other and the two types have a non-empty overlap. In other words, the "small" type must be non-empty. The subtype relation used in most the well-formedness definitions is that of overlapping, essential subtypes. Therefore, this is given a short name:

64.0 $IsSubtypeR : \Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsSubtypeR\,(\pi)(t_1, t_2)(env) \triangleq$

.2 $\quad Is\text{-}x\text{-}OverlappingSubtype(\pi)(\text{ESSENTIAL})(t_1, t_2)(env);$

65.0 $Is\text{-}x\text{-}OverlappingSubtype : \Pi \xrightarrow{t} RelationKind \rightarrow TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $Is\text{-}x\text{-}OverlappingSubtype\,(\pi)(kind)(t_1, t_2)(env) \triangleq$

.2 $\quad Is\text{-}x\text{-}Subtype(\pi)(kind)(t_1, t_2)(env) \wedge IsOverlapping(\pi)(t_1, t_2)(env)$

Overlap of recursively defined types does not lend itself to a definition with reference to fixed point induction because the base case cannot be established: the empty type is not overlapping any type. Instead, overlap may be defined as the negation of disjointness since fixed point induction works fine as a reference for defining the disjointness predicate.

66.0 $IsOverlapping : \Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsOverlapping\,(\pi)(t_1, t_2)(env) \triangleq$

.2 $\quad \neg\, IsDisjoint(NegWfClass(\pi))(t_1, t_2)(env);$

67.0 $IsDisjoint : \Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsDisjoint\,(\pi)(t_1, t_2)(env) \triangleq$

.2 $\quad \exists\, drel : TypeRelation \cdot IsDisjointnessRel(\pi)(drel)(env) \wedge mk\text{-}(t_1, t_2) \in drel$

One type representation is disjoint from another if it is possible to find a proof which establishes this. The proof is represented by a disjointness relation which simply includes all the subgoals necessary for completing the proof.

68.0 $IsDisjointnessRel : \Pi \xrightarrow{t} TypeRelation \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsDisjointnessRel\,(\pi)(drel)(env) \triangleq$

.2 $\quad \forall\, mk\text{-}(t_1, t_2) \in drel \cdot CheckPairDisjoint(\pi)(t_1, t_2)(drel)(env)$

When checking a disjointness proof, each subgoal must be verified. This is done by the auxiliary function *CheckPairDisjoint*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
69.0 CheckPairDisjoint : Π -t-> TypeR × TypeR → TypeRelation → Env → 𝔹
 .1 CheckPairDisjoint (π)(t1, t2)(drel)(env) ≜
 .2   cases mk-(t1, t2) :
 .3     mk-(mk-SetTypeR(et1), mk-SetTypeR(et2)) → false,
 .4     mk-(mk-Seq0TypeR(et1), mk-Seq0TypeR(et2)) → false,
 .5     mk-(mk-Seq1TypeR(et1), mk-Seq0TypeR(et2)) → mk-(et1, et2) ∈ drel,
 .6     mk-(mk-Seq0TypeR(et1), mk-Seq1TypeR(et2)) → mk-(et1, et2) ∈ drel,
 .7     mk-(mk-Seq1TypeR(et1), mk-Seq1TypeR(et2)) → mk-(et1, et2) ∈ drel,
 .8     mk-(mk-ProductTypeR(ts1), mk-ProductTypeR(ts2)) →
 .9         len ts1 ≠ len ts2 ∨ ∃ i ∈ inds ts1 · mk-(ts1(i), ts2(i)) ∈ drel,
.10     mk-(mk-CompositeTypeR(n1, fs1), mk-CompositeTypeR(n2, fs2)) →
.11         n1 ≠ n2 ∨ len fs1 ≠ len fs2 ∨
.12         ∃ i ∈ inds fs1 ·
.13             fs1(i).id ≠ fs2(i).id ∨ mk-(fs1(i).type, fs2(i).type) ∈ drel,
.14     mk-(mk-GeneralMapTypeR(at1, rt1), mk-GeneralMapTypeR(at2, rt2)) → false,
.15     mk-(mk-GeneralMapTypeR(at1, rt1), mk-InjectiveMapTypeR(at2, rt2)) → false,
.16     mk-(mk-InjectiveMapTypeR(at1, rt1), mk-GeneralMapTypeR(at2, rt2)) → false,
.17     mk-(mk-InjectiveMapTypeR(at1, rt1), mk-InjectiveMapTypeR(at2, rt2)) → false,
.18     mk-(mk-QuoteTypeR(q1), mk-QuoteTypeR(q2)) → q1 ≠ q2,
.19     mk-(mk-RetTypeR(et1), mk-RetTypeR(et2)) → mk-(et1, et2) ∈ drel,
.20     mk-(mk-ExitTypeR(et1), mk-ExitTypeR(et2)) → mk-(et1, et2) ∈ drel,
.21     mk-(mk-OpTypeR(st1, at1, rt1), mk-OpTypeR(st2, at2, rt2)) →
.22         (mk-(rt1, rt2) ∈ drel ∨ mk-(st1, st2) ∈ drel) ∧
.23         ¬ EmptyOpOverlap(NegWfClass(π))(t1, t2)(env),
.24     others → if Is[FunctionTypeR, TypeR](t1) ∧ Is[FunctionTypeR, TypeR](t2)
.25             then CheckPairFunctionDisjoint(π)(t1, t2)(drel)(env)
.26             else CheckPairNonMatchDisjoint(π)(t1, t2)(drel)(env)
.27 end
```

The above predicate verifies the correctness of a subgoal in the proof of a disjointness relationship. The two argument types, t_1 and t_2, are viewed as a disjointness relationship which should be the conclusion of some disjointness proof rule. This pair of types is matched against the conclusions of all rules and in case a match is found it is verified that the hypotheses of the rule (if any) are also included in the proof. Note that two set types, two map types, and two possibly empty sequence types are not disjoint because they share the empty set, map and sequence, respectively.

In case one of the two types is a name or a basic type, a union type or an invariant type, a further analysis is necessary in order to determine whether they are disjoint. This is done by the auxiliary function *CheckPairNonMatchDisjoint*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

70.0 $CheckPairNonMatchDisjoint : \Pi \xrightarrow{t} TypeR \times TypeR \to TypeRelation \to Env \to \mathbb{B}$

.1 $CheckPairNonMatchDisjoint\,(\pi)(t_1, t_2)(drel)(env) \triangleq$

.2 cases $mk\text{-}(t_1, t_2)$:

.3 $mk\text{-}(mk\text{-}Name(i), t), mk\text{-}(t, mk\text{-}Name(i)) \to$

.4 $\exists\, t' : TypeR \cdot IsBoundToDomTypeR(mk\text{-}Name(i))(t')(env) \wedge mk\text{-}(t, t') \in drel,$

.5 $mk\text{-}(mk\text{-}UnionTypeR(ts), t), mk\text{-}(t, mk\text{-}UnionTypeR(ts)) \to$

.6 $\forall\, t' \in ts \cdot mk\text{-}(t, t') \in drel,$

.7 $mk\text{-}(mk\text{-}InvTypeR(t', \text{-}), t), mk\text{-}(t, mk\text{-}InvTypeR(t', \text{-})) \to$

.8 $mk\text{-}(t, t') \in drel \vee is\text{-}POS(\pi),$

.9 $mk\text{-}(\text{-}, (\text{ANY})), mk\text{-}((\text{ANY}), \text{-}) \to \mathsf{false},$

.10 $mk\text{-}(\text{-}, (\text{VOID})), mk\text{-}((\text{VOID}), \text{-}) \to \mathsf{true},$

.11 $mk\text{-}((\text{EMPTYSET}), t), mk\text{-}(t, (\text{EMPTYSET})) \to$

.12 $\neg\, is\text{-}SetTypeR(t),$

.13 $mk\text{-}((\text{EMPTYMAP}), t), mk\text{-}(t, (\text{EMPTYMAP})) \to$

.14 $\neg\, (is\text{-}GeneralMapTypeR(t) \vee is\text{-}InjectiveMapTypeR(t)),$

.15 $mk\text{-}((\text{EMPTYSEQ}), t), mk\text{-}(t, (\text{EMPTYSEQ})) \to$

.16 $\neg\, is\text{-}Seq0TypeR(t),$

.17 $mk\text{-}(\text{-}, mk\text{-}TypeVarId(\text{-})), mk\text{-}(mk\text{-}TypeVarId(\text{-}), \text{-}) \to is\text{-}POS(\pi),$

.18 others $\to t_1 \neq t_2 \wedge$

.19 $\neg\, \{t_1, t_2\} \subseteq \{\text{NATONE}, \text{NAT}, \text{INT}, \text{RAT}, \text{REAL}\}$

.20 end

Here, the disjointness relationships involving two syntactically different type expressions are established. When t_1 or t_2 is the name of a recursively defined type, the unfolded type is simply required to be in the disjointness relationship. This approach may be seen as a way of planning a fixed point induction proof of the disjointness relationship. In such an induction, the base case concerns the empty type and it may be observed that disjointness actually holds if either t_1 or t_2 is empty.

Note that here, the only difference between the definite and possible disjointness relations appears in connection with invariant types.

Regarding type constants, special care is taken to avoid pairs which are overlapping. If none of the preceeding cases are encountered, a syntactical difference between t_1 and t_2 implies their disjointness.

71.0 $CheckPairFunctionDisjoint$:

.1 $\Pi \xrightarrow{t} FunctionTypeR \times FunctionTypeR \to TypeRelation \to Env \to \mathbb{B}$

.2 $CheckPairFunctionDisjoint\,(\pi)(t_1, t_2)(drel)(env) \triangleq$

.3 $\exists\, at_1, at_2, rt_1, rt_2 : TypeR \cdot$

.4 $IsFunctionTypeR(t_1)(at_1, rt_1) \wedge$

.5 $IsFunctionTypeR(t_2)(at_2, rt_2) \wedge$

.6 $\neg\, EmptyFnOverlap(NegWfClass(\pi))(t_1, t_2)(env) \wedge$

.7 $(mk\text{-}(rt_1, rt_2) \in drel \vee$

.8 $is\text{-}TotalFnTypeR(t_1) \wedge is\text{-}TotalFnTypeR(t_2) \wedge mk\text{-}(at_1, at_2) \in drel \vee$

.9 $is\text{-}POS(\pi) \wedge at_1 \neq at_2)$

For two function types to be disjoint, they must not both contain the empty function (the function with the empty graph). It can then be concluded that two function types are disjoint if their result types are disjoint. This also holds if they are total and their argument types are disjoint. In addition, possible disjointness of function types is accepted as long as the argument types are (syntactically) different.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
72.0 EmptyFnOverlap : Π -t-> TypeR × TypeR → Env → 𝔹
  .1 EmptyFnOverlap (π)(t1, t2)(env) ≜
  .2   IsTypeOfEmptyFn(π)(t1)(env) ∧
  .3   IsTypeOfEmptyFn(π)(t2)(env)
```

This predicate ensures that two function types do not both include the empty function which would otherwise result in overlap between the types.

```
73.0 IsTypeOfEmptyFn : Π -t-> TypeR → Env → 𝔹
  .1 IsTypeOfEmptyFn (π)(type)(env) ≜
  .2   cases type :
  .3     mk-PartialFnTypeR(at, -), mk-TotalFnTypeR(at, -) →
  .4         Is-x-EmptyType(π)(ESSENTIAL)(at)(env),
  .5     mk-TypeVarId(-) → is-POS(π),
  .6     others → false
  .7   end
```

This predicate tests whether a type is a function type containing just the empty function, i.e. the function with the empty function graph.

```
74.0 EmptyOpOverlap : Π -t-> TypeR × TypeR → Env → 𝔹
  .1 EmptyOpOverlap (π)(t1, t2)(env) ≜
  .2   IsTypeOfEmptyOp(π)(t1)(env) ∧
  .3   IsTypeOfEmptyOp(π)(t2)(env)
```

This predicate ensures that two operation types do not both include the empty operation which would otherwise result in overlap between the types.

```
75.0 IsTypeOfEmptyOp : Π -t-> TypeR → Env → 𝔹
  .1 IsTypeOfEmptyOp (π)(type)(env) ≜
  .2   cases type :
  .3     mk-OpTypeR(stt, at, rt) →
  .4         let at1 = mk-ProductTypeR([stt, at]),
  .5             rt1 = mk-ProductTypeR([stt, rt]) in
  .6         IsTypeOfEmptyFn(π)(mk-PartialFnTypeR(at1, rt1))(env),
  .7     others → false
  .8   end
```

This predicate tests whether a type is an operation type containing just the empty operation.

13.4.3 Auxiliary Type Relations and Functions

```
76.0 IsMapTypeR : TypeR -t-> TypeR × TypeR → 𝔹
  .1 IsMapTypeR (mt)(dt, rt) ≜
  .2   mt = mk-GeneralMapTypeR(dt, rt) ∨
  .3   mt = mk-InjectiveMapTypeR(dt, rt) ∨ mt = EMPTYMAP;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

77.0 $IsFunctionTypeR : TypeR \xrightarrow{t} TypeR \times TypeR \to \mathbb{B}$

.1 $IsFunctionTypeR\,(ft)(dt, rt) \triangleq$
.2 $\quad ft = mk\text{-}PartialFnTypeR(dt, rt) \vee$
.3 $\quad ft = mk\text{-}TotalFnTypeR(dt, rt);$

78.0 $IsSeqTypeR : TypeR \xrightarrow{t} TypeR \to \mathbb{B}$

.1 $IsSeqTypeR\,(seqt)(et) \triangleq$
.2 $\quad seqt = mk\text{-}Seq0TypeR(et) \vee$
.3 $\quad seqt = mk\text{-}Seq1TypeR(et) \vee seqt = \text{EMPTYSEQ};$

79.0 $LengthofSeq : \Pi \xrightarrow{t} TypeR \times \mathbb{N} \to Env \to \mathbb{B}$

.1 $LengthofSeq\,(\pi)(seqtype, seqlength)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi)$

Checks that the sequence type *seqtype* only represents sequence values with a length equal to *seqlength*. Note however that a more fine-grained notion of types is necessary in order to represent information about the length of sequences.

80.0 $CardofSet : \Pi \xrightarrow{t} TypeR \times \mathbb{N} \to Env \to \mathbb{B}$

.1 $CardofSet\,(\pi)(settype, setcard)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi)$

This function checks that *settype* only represents set values with a cardinality less than or equal to *setcard*. Note however that a more fine-grained notion of types is necessary in order to represent information about the cardinality of sets.

81.0 $IsNonEmptySet : \Pi \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.1 $IsNonEmptySet\,(\pi)(type)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi) \wedge \neg\, type = \text{EMPTYSET}$

This function checks that *settype* only represents nonempty set values.

82.0 $IsEquivalent : \Pi \xrightarrow{t} TypeR \times TypeR \to Env \to \mathbb{B}$

.1 $IsEquivalent\,(\pi)(t_1, t_2)(env) \triangleq$
.2 $\quad IsFullSubtype(\pi)(t_1, t_2)(env) \wedge IsFullSubtype(\pi)(t_2, t_1)(env)$

Here type equivalence is defined as the mutual, full subtype relationship.

83.0 $IsEssEquivTypeR : \Pi \to TypeR \times TypeR \to Env \to \mathbb{B}$

.1 $IsEssEquivTypeR\,(\pi)(t_1, t_2)(env) \triangleq$
.2 $\quad IsEssentialSubtype(\pi)(t_1, t_2)(env) \wedge$
.3 $\quad IsFullSubtype(\pi)(t_2, t_1)(env);$

84.0 $IsNonEmptyEssEquivTypeR : \Pi \to TypeR \times TypeR \to Env \to \mathbb{B}$

.1 $IsNonEmptyEssEquivTypeR\,(\pi)(t_1, t_2)(env) \triangleq$
.2 $\quad IsSubtypeR(\pi)(t_1, t_2)(env) \wedge$
.3 $\quad IsFullSubtype(\pi)(t_2, t_1)(env)$

In a typical use of these predicates, the type representation t_2 has been extracted from a type explicitly written by the user. The predicate then holds for all equivalent type representations t_1 which are also essential subtypes of t_2. The latter predicate holds, however, only if t_1 in addition is overlapping t_2 and thus non-empty.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

85.0 $IsIntersectionType : \Pi \xrightarrow{t} TypeR \times TypeR \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsIntersectionType\,(\pi)(t_1, t_2)(t)(env) \triangleq$

.2 cases π :

.3 $\quad mk\text{-}POS(\text{-}) \rightarrow IsFullSubtype(\pi)(t, t_1)(env) \wedge IsFullSubtype(\pi)(t, t_2)(env),$

.4 $\quad mk\text{-}DEF(\text{-}) \rightarrow t = t_1 \vee t = t_2$

.5 end

$IsIntersectionType(\pi)(t_1, t_2)(t)(env)$ is used to make sound or complete judgements regarding the equality of the set of values denoted by t and the intersection of the sets of values denoted by t_1 and t_2.

86.0 $SameValue : \Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $SameValue\,(\pi)(t_1, t_2)(env) \triangleq$

.2 cases π :

.3 $\quad mk\text{-}POS(\text{-}) \rightarrow t_1 = t_2,$

.4 $\quad mk\text{-}DEF(\text{-}) \rightarrow IsEquivalent(\pi)(t_1, t_2)(env) \wedge IsOneValueType(\pi)(t_1)(env)$

.5 end

SameValue checks that two values belonging to the two types t_1 respectively t_2 are one and the same value. This is possibly so if the two types are overlapping and definitely so if the two types are equivalent and represent singleton sets.

87.0 $IsEmptyType : \Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsEmptyType\,(\pi)(t)(env) \triangleq$

.2 $\quad Is\text{-}x\text{-}EmptyType(\pi)(\text{FULL})(t)(env);$

88.0 $Is\text{-}x\text{-}EmptyType : \Pi \xrightarrow{t} RelationKind \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $Is\text{-}x\text{-}EmptyType\,(\pi)(kind)(t)(env) \triangleq$

.2 $\quad IsEmptyAux(\pi)(kind)(t)(env)(\{\});$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
89.0 IsEmptyAux : Π →t RelationKind → TypeR → Env → TypeR-set → 𝔹
 .1 IsEmptyAux (π)(kind)(t)(env)(empty-types) ≜
 .2   t ∈ empty-types ∨
 .3   let empty-types1 = empty-types ∪ {t} in
 .4   cases t :
 .5     mk-Name(-) → IsEmptyAux(π)(kind)(env.typemap(t))(env)(empty-types1),
 .6     mk-Seq1 TypeR(t1) → IsEmptyAux(π)(kind)(t1)(env)(empty-types1),
 .7     mk-UnionTypeR(ts) → ∀ t1 ∈ ts · IsEmptyAux(π)(kind)(t1)(env)(empty-types1),
 .8     mk-ProductTypeR(ts) →
 .9         ∃ t1 ∈ elems ts · IsEmptyAux(π)(kind)(t1)(env)(empty-types1),
.10     mk-CompositeTypeR(-, fs) →
.11         ∃ mk-FieldR(-, et) ∈ elems fs · IsEmptyAux(π)(kind)(et)(env)(empty-types1),
.12     mk-InvTypeR(et, -) →
.13         IsEmptyAux(π)(kind)(et)(env)(empty-types1) ∨ is-POS(π) ∧ kind = FULL,
.14     mk-TotalFnTypeR(at, rt) →
.15         Is-x-EmptyType(π)(kind)(rt)(env) ∧
.16         ¬ Is-x-EmptyType(NegWfClass(π))(kind)(at)(env),
.17     mk-PartialFnTypeR(-, -) →
.18         false,
.19     mk-OpTypeR(stt, at, rt) →
.20         false,
.21     mk-RetTypeR(et), mk-ExitTypeR(et) → IsEmptyAux(π)(kind)(et)(env)(empty-types1),
.22     mk-TypeVarId(-) → is-POS(π),
.23     others → false
.24   end
```

Emptiness of a type may be caused by the least fixed point semantics of recursive type definitions. The bottom element in this connection is the empty type and a number of type constructors are strict with respect to this bottom element. This has motivated the special selection of type constructors considered above. Another way of making a type empty is by introducing an invariant which restricts the set of values in the type to the empty set. However, the latter possibility is only taken into account if doing a FULL emptiness check. When checking for ESSENTIAL emptyiess, only emptiness caused by the least fixed point semantics of recursive type definitions is considered.

It may be noted that recursion through function and operation types is not allowed/not introduced so there is no need for considering emptiness caused by such recursions.

```
90.0 IsOneValueType : Π →t TypeR → Env → 𝔹
 .1 IsOneValueType (π)(type)(env) ≜
 .2   IsOneValueTypeAux(π)(type)(env)({});
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

91.0 $IsOneValueTypeAux : \Pi \xrightarrow{t} TypeR \to Env \to TypeR\text{-set} \to \mathbb{B}$

.1 $IsOneValueTypeAux\ (\pi)(type)(env)(visitedtypes) \triangleq$
.2 $type \notin visitedtypes \wedge$
.3 let $visitedtypes_1 = visitedtypes \cup \{type\}$ in
.4 cases $type$:
.5 $mk\text{-}Name(\text{-}) \to IsOneValueTypeAux(\pi)(env.typemap(type))(env)(visitedtypes_1)$,
.6 $mk\text{-}InvTypeR(et, \text{-}) \to \neg\, IsEmptyType(NegWfClass(\pi))(et)(env) \wedge is\text{-}POS(\pi)$,
.7 $mk\text{-}QuoteTypeR(\text{-}) \to$ true,
.8 $mk\text{-}CompositeTypeR(\text{-}, fields) \to$
.9 $\forall f \in$ elems $fields \cdot IsOneValueTypeAux(\pi)(f.type)(env)(visitedtypes_1)$,
.10 $mk\text{-}UnionTypeR(ts) \to$
.11 $\exists t \in ts \cdot$
.12 $IsOneValueTypeAux(\pi)(t)(env)(visitedtypes_1) \wedge$
.13 $\forall t' \in ts \cdot$
.14 $IsEquivalent(\pi)(t, t')(env) \vee IsEmptyType(\pi)(t')(env)$,
.15 $mk\text{-}ProductTypeR(factors) \to$
.16 $\forall t \in$ elems $factors \cdot IsOneValueTypeAux(\pi)(t)(env)(visitedtypes_1)$,
.17 $mk\text{-}SetTypeR(et), mk\text{-}Seq0Type(et) \to IsEmptyType(\pi)(et)(env)$,
.18 $mk\text{-}GeneralMapTypeR(at, rt), mk\text{-}InjectiveMapTypeR(at, rt) \to$
.19 $IsEmptyType(\pi)(at)(env) \vee IsEmptyType(\pi)(rt)(env)$,
.20 $mk\text{-}PartialFnTypeR(at, rt) \to$
.21 $IsEmptyType(\pi)(at)(env) \vee IsEmptyType(\pi)(rt)(env)$,
.22 $mk\text{-}TotalFnTypeR(at, rt) \to$
.23 $IsEmptyType(\pi)(at)(env) \vee$
.24 $is\text{-}POS(\pi) \wedge IsEmptyType(\pi)(rt)(env) \vee$
.25 $IsOneValueTypeAux(\pi)(at)(env)(visitedtypes_1) \wedge$
.26 $IsOneValueTypeAux(\pi)(rt)(env)(visitedtypes_1)$,
.27 $mk\text{-}RetTypeR(et), mk\text{-}ExitTypeR(et) \to IsOneValueTypeAux(\pi)(et)(env)(visitedtypes_1)$,
.28 (EMPTYSET), (EMPTYSEQ), (EMPTYMAP), (UNITTYPE), (CONT), (VOID) $\to$ true,
.29 $mk\text{-}TypeVarId(\text{-}) \to is\text{-}POS(\pi)$,
.30 others $\to$ false
.31 end

The function *IsOneValueType* checks whether a type represents a singleton set. Note, that line 91.9 includes a composite type with no fields.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
92.0 IsFiniteTypeR : Π →t TypeR → Env → 𝔹
  .1 IsFiniteTypeR (π)(t)(env) ≜
  .2   IsEmptyType(π)(t)(env) ∨ IsOneValueType(π)(t)(env) ∨
  .3   cases t :
  .4     mk-Name(-) →
  .5         let nt = env.typemap(t) in
  .6         cases π :
  .7           mk-DEF(-) → DoesNotBuildOn(π)(nt, t)(env) ∧ IsFiniteTypeR(π)(nt)(env),
  .8           mk-POS(-) → BuildsOn(π)(nt, t)(env) ∨ IsFiniteTypeR(π)(nt)(env)
  .9         end,
  .10    mk-InvTypeR(et, -) → is-POS(π) ∨ IsFiniteTypeR(π)(et)(env),
  .11    mk-CompositeTypeR(-, fields) →
  .12        ∀ f ∈ elems fields · IsFiniteTypeR(π)(f.type)(env),
  .13    mk-UnionTypeR(ts) →
  .14        ∀ t ∈ ts · IsFiniteTypeR(π)(t)(env),
  .15    mk-ProductTypeR(factors) →
  .16        ∀ t ∈ elems factors · IsFiniteTypeR(π)(t)(env),
  .17    mk-SetTypeR(et) → IsFiniteTypeR(π)(et)(env),
  .18    mk-Seq0TypeR(et) → IsEmptyType(π)(et)(env),
  .19    mk-GeneralMapTypeR(at, rt), mk-InjectiveMapTypeR(at, rt) →
  .20        IsFiniteTypeR(π)(at)(env) ∧ IsFiniteTypeR(π)(rt)(env),
  .21    mk-PartialFnTypeR(at, rt), mk-TotalFnTypeR(at, rt) →
  .22        IsFiniteTypeR(π)(at)(env) ∧ IsFiniteTypeR(π)(rt)(env),
  .23    mk-OpTypeR(stt, at, rt) →
  .24        IsFiniteTypeR(π)(stt)(env) ∧
  .25        IsFiniteTypeR(π)(at)(env) ∧ IsFiniteTypeR(π)(rt)(env),
  .26    mk-RetTypeR(et), mk-ExitTypeR(et) → IsFiniteTypeR(π)(et)(env),
  .27    mk-QuoteTypeR(-) → true,
  .28    (BOOLEAN) → true,
  .29    others → false
  .30  end;

93.0 IsUnionComp : TypeR × TypeR →t Env → 𝔹
  .1 IsUnionComp (t1, t2)(env) ≜
  .2   let check = λ t1 : TypeR, t2 : TypeR ·
  .3                 IsUnionComp(t1, t2)(env) in
  .4   cases mk-(t1, t2) :
  .5     mk-(mk-InvTypeR(et1, i), mk-InvTypeR(et2, i)) → check(et1, et2),
  .6     mk-(mk-CompositeTypeR(n, fs1), mk-CompositeTypeR(n, fs2)) →
  .7         len fs1 = len fs2 ∧
  .8         ∀ i ∈ inds fs1 · fs1(i).id = fs2(i).id ∧ check(fs1(i).type, fs2(i).type),
  .9     mk-(-, mk-UnionTypeR(ts2)) → ∃ et2 ∈ ts2 · check(t1, et2),
  .10    mk-(mk-ProductTypeR(ts1), mk-ProductTypeR(ts2)) →
  .11        len ts1 = len ts2 ∧ ∀ i ∈ inds ts1 · check(ts1(i), ts2(i)),
  .12    mk-(-, mk-Seq0TypeR(et2)) →
  .13        check(t1, mk-UnionTypeR({EMPTYSEQ, mk-Seq1TypeR(et2)})),
  .14    mk-(mk-RetTypeR(et1), mk-RetTypeR(et2)) → check(et1, et2),
  .15    mk-(mk-ExitTypeR(et1), mk-ExitTypeR(et2)) → check(et1, et2),
  .16    others → t1 = t2
  .17  end
```

This predicate describes for a type t_2, a finite set of so-called union components. The set of all the values in all of these union components equals the set of values in t_2. The union components can be seen as the result of splitting union types into components and distributing certain type constructs over union types if possible. In case there are

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

no unions to split, t_2 is its own union component.

```
94.0 IsUnionComponent : TypeR × TypeR →(t) Env → 𝔹
  .1 IsUnionComponent (t1, t2)(env) ≜
  .2   IsUnionComp(TypeSubst(env.typemap)(t1), t2)(env)
```

Like the above predicate except that all type names are expanded once before considering the union components.

```
95.0 DistributeTypeR : TypeR →(t) Env → TypeR
  .1 DistributeTypeR (t)(env) ≜
  .2   if ∃ t1, t2 : TypeR ·
  .3           t1 ≠ t2 ∧ IsUnionComponent(t1, t)(env) ∧ IsUnionComponent(t2, t)(env)
  .4   then let summands : TypeR-set be st
  .5                 ∀ s : TypeR · s ∈ summands ⇔ IsUnionComponent(s, t)(env) in
  .6        mk-UnionTypeR(summands)
  .7   else t
```

This function "normalises" a type t by forming a union type of all the union components of t.

```
96.0 Shape : TypeR →(t) TypeR
  .1 Shape (t) ≜
  .2   cases t :
  .3     mk-InvTypeR(et, -) → et,
  .4     others → t
  .5   end
```

13.5 Extended Abstract Syntax

Direct definition of higher-order (curried) functions is in many respects similar to function definition by lambda expressions. The similarities are, in the Static Semantics, emphasised by introduction of an abstraction and generalisation of the lambda expressions known from the Outer Abstract Syntax. This new Static Semantic representation of lambda expressions is called *LambdaR*. The Static Semantics of (usual) lambda expressions and direct function definitions is then defined by expressing them in terms of *LambdaR* expressions. The *LambdaR* expressions constitute an abstraction since only one formal parameter is allowed. They do, however, also constitute a generalisation in the sense that it now can be asserted that the denotation of the lambda expression should be a total function. Moreover, pre-conditions may now also be specified.

```
97.0 LambdaR :: id : Id
  .1            type : TypeR
  .2            body : Expr | LambdaR
  .3            fnpre : [Expr]
  .4            total : 𝔹
```

14 The Static Semantics

In this clause the Static Semantics of VDM-SL specifications is defined in terms of so-called well-formedness predicates. First, whole specifications are considered then the state, type, value, function and operation definitions which constitute a specification and, finally, expressions, state designators, statements, patterns and bindings. Finally, a number of auxiliary functions used in the definition are defined.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.1 Documents

According to the outer abstract syntax, a specification written in VDM-SL constitutes a *document* which in turn is a sequence of definition blocks. The Dynamic Semantics of a specification, is the set of interpretations satisfying the specification. An interpretation is a mapping from names to the values, types, functions, operations or state components which they denote. This is called an ENV_{PURE} in the Dynamic Semantics.

The purpose of the Static Semantics is to identify some of the specifications which definitely are satisfied by at least one interpretation and also to identify some of the specifications which cannot possibly be satisfied by any interpretation. If the Static Semantic function *wf-Spec* yields true when applied to a well-formedness classification π of type *DEF* and a sequence of definition blocks constituting a specification then the specification is definitely satisfied by at least one interpretation. If, however, the Static Semantic function *wf-Spec* yields false when applied to a well-formedness classification π of type *POS* and a specification then the specification cannot possibly be satisfied by any interpretation.

98.0 $wf\text{-}Spec : \Pi \xrightarrow{t} DefinitionBlock^* \to \mathbb{B}$

.1 $wf\text{-}Spec(\pi)(dbs) \triangleq$

.2 let $typedefs = \mathsf{conc}\,[dbs(i).typedefs \mid i \in \mathsf{inds}\ dbs \cdot is\text{-}TypeDefinitions(dbs(i))]$,

.3 $statedefs = [dbs(i) \mid i \in \mathsf{inds}\ dbs \cdot is\text{-}StateDef(dbs(i))]$,

.4 $valdefs = \mathsf{conc}\,[dbs(i).valdefs \mid i \in \mathsf{inds}\ dbs \cdot is\text{-}ValueDefinitions(dbs(i))]$,

.5 $fndefs = \mathsf{conc}\,[dbs(i).funcdefs \mid i \in \mathsf{inds}\ dbs \cdot is\text{-}FunctionDefinitions(dbs(i))]$,

.6 $opdefs = \mathsf{conc}\,[dbs(i).opdefs \mid i \in \mathsf{inds}\ dbs \cdot is\text{-}OperationDefinitions(dbs(i))]$,

.7 $fnppdefs = ExtractFnPrePostDefs(fndefs)$,

.8 $used\text{-}ids = UsedIds(typedefs, statedefs, valdefs, fndefs, opdefs)$,

.9 $opppdefs = ExtractOpPrePostDefs(opdefs, statedefs, used\text{-}ids)$,

.10 $invdefs = ExtractInvDefs(typedefs \frown statedefs)$,

.11 $initdefs = ExtractInitDefs(statedefs)$,

.12 $valfnopdefs = valdefs \frown fndefs \frown opdefs \frown$

.13 $fnppdefs \frown opppdefs \frown invdefs \frown initdefs$ in

.14 $\exists\, env : Env, venv : VisibleEnv \cdot$

.15 $wf\text{-}TypeStateDefsInInitialEnv(\pi)(typedefs, statedefs)(env) \wedge$

.16 $wf\text{-}ValFnOpDefs(\pi)(valfnopdefs)(venv)(env) \wedge$

.17 $NoNameClash(typedefs \frown valfnopdefs) \wedge$

.18 $NoUnsatisfiableTypes(\pi)(valfnopdefs \frown typedefs)(env) \wedge$

.19 $NoLooseInvariants(\pi)(invdefs, valfnopdefs)(env)$

The verification of the well-formedness of a sequence of definition blocks is performed by dividing the definitions into separate classes. The definitions of pre-, post-, inv-, and init- functions are then derived from the function, operation, type, and state definitions. The existence of an interpretation satisfying all the definitions is then verified by considering two classes of definitions separately: (1) type and state definitions; and (2) value, function and operation definitions. This is motivated by the fact that the former may be verified (almost) independently of the latter (since inv- functions are verified as if they were defined simultaneously with the other functions). The verification of the value, function and operation definitions may then be done under the assumption that the type definitions have already been verified. The only potential problem is that of mutual recursion between type definitions and function definitions (via the invariant in a type definition). This is avoided by adding an explicit analysis (98.18) which ensures that specifications containing such mutually recursive definitions will not be considered definitely well-formed.

The different sequences of definitions considered above belong to the following specialised syntax classes:

99.0 $ValFnOpTpDefs = ValFnOpTpDef^*$;

100.0 $ValFnOpTpDef = ValFnOpDef \mid TypeDef$;

101.0 $ValFnOpDefs = ValFnOpDef^*$;

102.0 $ValFnOpDef = ValFnDef \mid OperationDef$;

103.0 $ValFnDefs = ValFnDef^*$;

104.0 $ValFnDef = ValueDef \mid FunctionDef$;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

105.0 $TypeDefs = TypeDef^*$

14.1.1 Auxiliary Well-formedness Requirements

106.0 $NoNameClash : ValFnOpTpDefs \xrightarrow{t} \mathbb{B}$
.1 $NoNameClash\,(ds) \triangleq$
.2 $\quad \forall\, i, j \in \mathsf{inds}\ ds \cdot$
.3 $\qquad i \neq j \Rightarrow$
.4 $\qquad LefthandSide(ds(i)) \neq LefthandSide(ds(j)) \wedge$
.5 $\qquad (DefNamesInDef(ds(i)) \cap DefNamesInDef(ds(j)) \neq \{\}$
.6 $\qquad\quad \Rightarrow\ is\text{-}ValueDef(ds(i)) \wedge is\text{-}ValueDef(ds(j)))$

Note that, according to the Dynamic Semantics, identical names are allowed in the patterns of two different value definitions (106.5-.6).

107.0 $LefthandSide : ValFnOpTpDef \xrightarrow{t} Pattern \mid Id$
.1 $LefthandSide\,(d) \triangleq$
.2 $\quad \mathsf{if}\ is\text{-}ValueDef(d)\ \mathsf{then}\ d.pat\ \mathsf{else}\ d.id;$

108.0 $NoUnsatisfiableTypes : \Pi \xrightarrow{t} ValFnOpTpDefs \to Env \to \mathbb{B}$
.1 $NoUnsatisfiableTypes\,(\pi)(ds)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi) \vee$
.3 $\quad \neg\, \exists\, fd : ValFnDef, td : TypeDef \cdot$
.4 $\qquad \{mk\text{-}(fd, td), mk\text{-}(td, fd)\} \subseteq ValFnOpTpTransDefDep(ds)$

An unsatisfiable type definition may be written by introducing an invariant which refers recursively to the type being defined. If there are no such recursions through type definitions, this problem can definitely not occur. Even if there is recursion, it is possible that the recursion does not cause unsatisfiability.

109.0 $NoLooseInvariants : \Pi \xrightarrow{t} FunctionDef^* \times ValFnOpDefs \to Env \to \mathbb{B}$
.1 $NoLooseInvariants\,(\pi)(invfns, ds)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi) \vee$
.3 $\quad \forall\, invfn \in \mathsf{elems}\ invfns \cdot$
.4 $\qquad NoLoosenessInDepValFnDefs(invfn)(ds)(env)$

Underspecified invariants on types are not allowed according to the Dynamic Semantics. It may be ensured that an invariant is definitely not underspecified by disallowing all constructs which might cause looseness. However, even if such constructs appear in an invariant definition or in other definitions which an invariant definition depends on, it may still be possible that it does not actually cause looseness.

110.0 $NoLoosenessInDepValFnDefs : ValFnDef \xrightarrow{t} ValFnOpDefs \to Env \to \mathbb{B}$
.1 $NoLoosenessInDepValFnDefs\,(d_1)(ds)(env) \triangleq$
.2 $\quad NoLoosenessInValFnDef(d_1) \wedge$
.3 $\quad \forall\, d_2 \in \mathsf{elems}\ ds \cdot$
.4 $\qquad mk\text{-}(d_1, d_2) \in ValFnOpTpTransDefDep(ds) \wedge$
.5 $\qquad (is\text{-}ValueDef(d_2) \vee IsFunctionDef(d_2)) \Rightarrow$
.6 $\qquad NoLoosenessInValFnDef(d_2);$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

111.0 $NoLoosenessInValFnDef : ValFnDef \xrightarrow{t} \mathbb{B}$

.1 $NoLoosenessInValFnDef\,(d) \triangleq$
.2 $(\forall\, s \in SynComps[Expr](d) \cdot NoLoosenessInExpr(s)) \wedge$
.3 $\forall\, s \in SynComps[Pattern](d) \cdot NoLoosenessInPat(s) \wedge$
.4 $\forall\, s \in SynComps[ExplFnDef](d) \cdot$
.5 $\quad \neg\, HasNonTrivialPreCond(mk\text{-}POS(\{\}))(s) \wedge$
.6 $SynComps[ImplFnDef](d) = \{\};$

112.0 $NoLoosenessInExpr : Expr \xrightarrow{t} \mathbb{B}$

.1 $NoLoosenessInExpr\,(e) \triangleq$
.2 $\neg\, is\text{-}LetBeSTExpr(e);$

113.0 $NoLoosenessInPat : Pattern \xrightarrow{t} \mathbb{B}$

.1 $NoLoosenessInPat\,(p) \triangleq$
.2 $\neg\, is\text{-}SetUnionPattern(p) \wedge \neg\, is\text{-}SetEnumPattern(p) \wedge \neg\, is\text{-}SeqConcPattern(p)$

14.2 Definitions

14.2.1 Type and State Definitions

The well-formedness of type and state definitions is expressed with respect to an initial (top level) environment of the whole specification excluding functions and operations. It is ensured that this initial environment contains a type map including entries for all the type definitions and also for the type of the state, if a state definition is present. The type map is also checked for well-formedness but it should be noted that this does not include well-formedness checks of the invariants. The invariants are checked indirectly by considering the corresponding 'inv-...' functions which are extracted and checked together with the other top level function definitions. When checking the function and operation definitions, the environment used is one with respect to which the state and type definitions are well-formed.

114.0 $wf\text{-}TypeStateDefsInInitialEnv : \Pi \xrightarrow{t} TypeDefs \times StateDef^* \to Env \to \mathbb{B}$

.1 $wf\text{-}TypeStateDefsInInitialEnv\,(\pi)(tds, sds)(env) \triangleq$
.2 $\mathsf{len}\ sds \leq 1 \wedge$
.3 $\mathsf{dom}\ ExtractTypeMap(tds) \cap \mathsf{dom}\ StatcTypeMap(sds) = \{\} \wedge$
.4 $wf\text{-}TypeMap(\pi)(env) \wedge$
.5 $env.typemap = ExtractTypeMap(tds) \biguplus StateTypeMap(sds) \wedge$
.6 $env.visibleenv = \{n \mapsto mk\text{-}DomR(\text{LOOKINTYPEMAP}) \mid n : Name \cdot n \in \mathsf{dom}\ env.typemap\} \wedge$
.7 $env.statename = StateName(sds) \wedge$
.8 $env.typevars = \{\} \wedge$
.9 $SatisfiableStateInitialisation(\pi)(sds)$

115.0 $StateName : StateDef^* \xrightarrow{t} [Name]$

.1 $StateName\,(sds) \triangleq$
.2 $\mathsf{if}\ sds = []\ \mathsf{then}\ \mathsf{nil}\ \mathsf{else}\ mk\text{-}Name((\mathsf{hd}\ sds).stid);$

116.0 $StateTypeMap : StateDef^* \xrightarrow{t} TypeMap$

.1 $StateTypeMap\,(sds) \triangleq$
.2 $\mathsf{if}\ sds = []\ \mathsf{then}\ \{\mapsto\}$
.3 $\mathsf{else\ let}\ mk\text{-}StateDef(stid, fields, stinv, \text{-}) = \mathsf{hd}\ sds\ \mathsf{in}$
.4 $\quad ExtractTypeMap([mk\text{-}TaggedTypeDef(stid, fields, stinv)]);$

117.0 $SatisfiableStateInitialisation : \Pi \xrightarrow{t} StateDef^* \to \mathbb{B}$

.1 $SatisfiableStateInitialisation\,(\pi)(sds) \triangleq$
.2 $is\text{-}POS(\pi) \vee sds = []$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.2.1.1 Types and Type maps

In this clause, the well-formedness of type definitions and type expressions is defined. The definition is indirect: type definitions are first transformed to a so-called type map mapping type names to the static semantic representation of the types which they denote. The type map is then checked for well-formedness. Likewise, type expressions are transformed to type representations which are then verified.

118.0 $wf\text{-}TypeMap : \Pi \xrightarrow{t} Env \to \mathbb{B}$

.1 $wf\text{-}TypeMap\,(\pi)(env) \triangleq$
.2 $\quad \forall\, t \in \mathsf{rng}\ env.typemap \cdot$
.3 $\qquad wf\text{-}TypeR(\pi)(t)(env);$

119.0 $wf\text{-}Type : \Pi \xrightarrow{t} Type \mid OpType \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}Type\,(\pi)(type)(typerep)(env) \triangleq$
.2 $\quad typerep = ExtractTypeR(type)(env) \wedge wf\text{-}TypeR(\pi)(typerep)(env);$

120.0 $wf\text{-}AssertedType : \Pi \xrightarrow{t} [Type] \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}AssertedType\,(\pi)(ot)(t)(env) \triangleq$
.2 $\quad \mathsf{if}\ ot = \mathsf{nil}$
.3 $\quad \mathsf{then}\ t = \text{ANY}$
.4 $\quad \mathsf{else}\ wf\text{-}Type(\pi)(ot)(t)(env);$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
121.0 wf-TypeR : Π →t TypeR → Env → 𝔹
   .1 wf-TypeR (π)(type)(env) ≙
   .2   cases type :
   .3     mk-InvTypeR(t, -) →
   .4         wf-TypeR(π)(t)(env),
   .5     mk-CompositeTypeR(id, fs) →
   .6         (∃! t : CompositeTypeRIsTypeRwithTagInEnv(t, id)(env)) ∧
   .7         (∀ mk-FieldR(-, t) ∈ elems fs · wf-TypeR(π)(t)(env)) ∧
   .8         ∀ i1, i2 ∈ inds fs ·
   .9             i1 ≠ i2 ∧ fs(i1).id ≠ nil ⇒ fs(i1).id ≠ fs(i2).id,
  .10     mk-UnionTypeR(ts) →
  .11         ∀ t ∈ ts · wf-TypeR(π)(t)(env),
  .12     mk-ProductTypeR(ts) →
  .13         ∀ t ∈ elems ts · wf-TypeR(π)(t)(env),
  .14     mk-SetTypeR(t) →
  .15         IsFlatTypeR(π)(t)(env) ∧ wf-TypeR(π)(t)(env),
  .16     mk-Seq0TypeR(t) →
  .17         wf-TypeR(π)(t)(env),
  .18     mk-Seq1TypeR(t) →
  .19         wf-TypeR(π)(t)(env),
  .20     mk-GeneralMapTypeR(dt, rt) →
  .21         IsFlatTypeR(π)(dt)(env) ∧
  .22         wf-TypeR(π)(dt)(env) ∧ wf-TypeR(π)(rt)(env),
  .23     mk-InjectiveMapTypeR(dt, rt) →
  .24         IsFlatTypeR(π)(dt)(env) ∧
  .25         wf-TypeR(π)(dt)(env) ∧ wf-TypeR(π)(rt)(env),
  .26     mk-PartialFnTypeR(dt, rt) →
  .27         DoesNotBuildOn(π)(type, type)(env) ∧
  .28         wf-TypeR(π)(dt)(env) ∧ wf-TypeR(π)(rt)(env),
  .29     mk-TotalFnTypeR(dt, rt) →
  .30         wf-TypeR(π)(mk-PartialFnTypeR(dt, rt))(env),
  .31     mk-OpTypeR(-, dt, rt) →
  .32         IsFlatTypeR(π)(dt)(env) ∧ IsFlatTypeR(π)(rt)(env) ∧
  .33         wf-TypeR(π)(dt)(env) ∧ wf-TypeR(π)(rt)(env),
  .34     mk-RetTypeR(t) → wf-TypeR(π)(t)(env),
  .35     mk-ExitTypeR(t) → wf-TypeR(π)(t)(env),
  .36     mk-Name(-) →
  .37         IsVisibleTypeName(type)(env),
  .38     mk-TypeVarId(-) →
  .39         IsVisibleTypeVar(type)(env),
  .40     others → true
  .41   end
```

Note that well-formedness of a type expression is expressed relatively to a whole environment – not just the type map part. This is due to the fact that visibility of (polymorphic) type variables must be verified (121.39). Notice also that well-formedness of invariants is checked elsewhere by checking the corresponding "inv-..." function definitions. In line 121.27, it is checked that function types do not build (recursively) on themselves because this would give rise to non-continuity of the corresponding type functional which is disallowed in the Dynamic Semantics.

```
122.0 IsFlatTypeR : Π →t TypeR → Env → 𝔹
   .1 IsFlatTypeR (π)(t)(env) ≙
   .2   ∀ ft : FunctionTypeR · ft ≠ t ∧ DoesNotBuildOn(π)(t, ft)(env);
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

123.0 $BuildsOn : \Pi \xrightarrow{t} TypeR \times TypeR \to Env \to \mathbb{B}$
.1 $BuildsOn\,(\pi)(t_1, t_2)(env) \triangleq$
.2 $\quad mk\text{-}(t_1, t_2) \in TransTypeRDep(\pi)(t_1)(env);$

124.0 $DoesNotBuildOn : \Pi \xrightarrow{t} TypeR \times TypeR \to Env \to \mathbb{B}$
.1 $DoesNotBuildOn\,(\pi)(t_1, t_2)(env) \triangleq$
.2 $\quad \neg\, BuildsOn(NegWfClass(\pi))(t_1, t_2)(env);$

125.0 $IsTypeRwithTagInEnv : TypeR \times Id \xrightarrow{t} Env \to \mathbb{B}$
.1 $IsTypeRwithTagInEnv\,(t, id)(env) \triangleq$
.2 $\quad IsTagOfType(id, t)(\mathsf{rng}\ env.typemap);$

126.0 $IsTagOfType : Id \times TypeR \xrightarrow{t} TypeR\text{-}\mathsf{set} \to \mathbb{B}$
.1 $IsTagOfType\,(id, t)(ts) \triangleq$
.2 $\quad is\text{-}CompositeTypeR(t) \wedge$
.3 $\quad \exists\, t1 \in ts \cdot$
.4 $\qquad IsTypeRSubExpr(t, t1) \wedge t.id = id;$

127.0 $IsTypeRSubExpr : TypeR \times TypeR \xrightarrow{t} \mathbb{B}$
.1 $IsTypeRSubExpr\,(se, e) \triangleq$
.2 $\quad mk\text{-}(se, e) \in TransReflSynSubComp[TypeR](e)$

14.2.1.2 Extraction of Type Representations

128.0 $ExtractTypeMap : TypeDef^* \xrightarrow{t} TypeMap$
.1 $ExtractTypeMap\,(tds) \triangleq$
.2 $\quad \mathsf{if}\ tds = []$
.3 $\quad \mathsf{then}\ \{\mapsto\}$
.4 $\quad \mathsf{else\ let}\ tdef = \mathsf{hd}\ tds,$
.5 $\qquad name = mk\text{-}Name(tdef.id),$
.6 $\qquad tinv = tdef.typeinv,$
.7 $\qquad tp = \mathsf{if}\ is\text{-}TaggedTypeDef(tdef)$
.8 $\qquad\quad \mathsf{then}\ mk\text{-}CompositeType(tdef.id, tdef.fields)\ \mathsf{else}\ tdef.shape,$
.9 $\qquad tpr = ExtractTypeR(tp)(EmptyEnv)\ \mathsf{in}$
.10 $\quad \{name \mapsto \mathsf{if}\ tinv = \mathsf{nil}\ \mathsf{then}\ tpr\ \mathsf{else}\ mk\text{-}InvTypeR(tpr, tinv)\} \dagger$
.11 $\quad ExtractTypeMap(\mathsf{tl}\ tds);$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
129.0 ExtractTypeR : Type | OpType -t-> Env → TypeR
  .1 ExtractTypeR (tp)(env) ≜
  .2   cases tp :
  .3     mk-BracketedType(t) → ExtractTypeR(t)(env),
  .4     mk-QuoteType(lit) → mk-QuoteTypeR(lit),
  .5     mk-CompositeType(id, fs) → mk-CompositeTypeR(id, ExtractFieldRs(fs)(env)),
  .6     mk-UnionType(ts) → mk-UnionTypeR(elems ExtractTypeRs(ts)(env)),
  .7     mk-ProductType(ts) → mk-ProductTypeR(ExtractTypeRs(ts)(env)),
  .8     mk-OptionalType(t) → mk-UnionTypeR({ExtractTypeR(t)(env), UNITTYPE}),
  .9     mk-SetType(t) → mk-SetTypeR(ExtractTypeR(t)(env)),
  .10    mk-Seq0Type(t) → mk-Seq0TypeR(ExtractTypeR(t)(env)),
  .11    mk-Seq1Type(t) → mk-Seq1TypeR(ExtractTypeR(t)(env)),
  .12    mk-GeneralMapType(dt, rt) →
  .13        mk-GeneralMapTypeR(ExtractTypeR(dt)(env), ExtractTypeR(rt)(env)),
  .14    mk-InjectiveMapType(dt, rt) →
  .15        mk-InjectiveMapTypeR(ExtractTypeR(dt)(env), ExtractTypeR(rt)(env)),
  .16    mk-PartialFnType(dt, rt) →
  .17        mk-PartialFnTypeR(ExtractTypeR(dt)(env), ExtractTypeR(rt)(env)),
  .18    mk-TotalFnType(dt, rt) →
  .19        mk-TotalFnTypeR(ExtractTypeR(dt)(env), ExtractTypeR(rt)(env)),
  .20    mk-OpType(dt, rt) →
  .21        mk-OpTypeR(env.statename, ExtractTypeR(dt)(env), ExtractTypeR(rt)(env)),
  .22    others → tp
  .23  end;

130.0 ExtractTypeRs : Type* -t-> Env → TypeR*
  .1 ExtractTypeRs (ts)(env) ≜
  .2   [ExtractTypeR(ts(i))(env) | i ∈ inds ts];

131.0 ExtractFieldRs : Field* -t-> Env → FieldR*
  .1 ExtractFieldRs (fs)(env) ≜
  .2   [mk-FieldR(fs(i).sel, ExtractTypeR(fs(i).type)(env)) | i ∈ inds fs]
```

14.2.2 Value, Function, and Operation Definitions

14.2.2.1 Simultaneous Definitions

Simultaneous definitions are verified in an environment containing interpretations of the names defined in the context. They are verified with respect to another environment containing interpretations of the names being defined. There are several problems related to the verification of simultaneous definitions in VDM-SL. First of all, explicit type information need not be included in value definitions (which may be mutually recursive). Secondly, the notion of type representations used in the Static Semantics denotes only "real" values, i.e. excluding $\perp$. So in order to conclude that a recursively defined entity definitely has some type, termination must be ensured.

The Static Semantics does not attempt to solve the mutually recursive type equations which might be extracted from the mutually recursive value definitions. Instead the definitions are first partitioned into groups. Two definitions are in the same group if and only if they are mutually recursive. Each definition which is not mutually recursive with any other definition forms a singleton group. The groups are then sorted topologically according to dependencies: One group A depends on another B iff at least one definition in A depends on an entity defined by a definition in B. The definition groups are then verified one by one, in order, starting with the group(s) which do not depend on any other group. After verification of a group, the names defined and their types are then added to the environment which is used when verifying the following groups.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

132.0 $wf\text{-}ValFnOpDefs : \Pi \xrightarrow{t} ValFnOpDefs \to VisibleEnv \to Env \to \mathbb{B}$

.1 $wf\text{-}ValFnOpDefs\,(\pi)(ds)(denv)(env) \triangleq$
.2 let $defgrps : ValFnOpDef\text{-set}^* = OrderedDefGroups(ds, env)$ in
.3 $wf\text{-}OrderedDefGroups(\pi)(defgrps)(denv)(env)$

14.2.2.2 Ordered Definition Groups and Definition Sequences

133.0 $NonTpDef = ValFnOpDef \mid DefBind;$

134.0 $NonTpDefs = NonTpDef^*$

135.0 $wf\text{-}OrderedDefGroups : \Pi \xrightarrow{t} ValFnOpDef\text{-set}^* \to VisibleEnv \to Env \to \mathbb{B}$

.1 $wf\text{-}OrderedDefGroups\,(\pi)(dgs)(denv)(env) \triangleq$
.2 cases dgs :
.3 $([]) \to denv = \{\mapsto\},$
.4 $preceding \frown [last] \to$
.5 $\exists\, denv_1, denv_2 : VisibleEnv \cdot$
.6 $wf\text{-}OrderedDefGroups(\pi)(preceding)(denv_1)(env) \wedge$
.7 $wf\text{-}DefGroup(\pi)(last)(denv_2)(UpdateEnv(denv_1)(env)) \wedge$
.8 $IsVisEnvMerge(\pi)(denv_1, denv_2)(denv)(env)$
.9 end

Before verifying a group, the environment is augmented by "safe" typings of the entities being defined. Otherwise, the bodies of the definitions could not be type checked as they would include names (the recursive references) for which no type was given. The safe typings are basically the types stated by the user as part of the definition, however restricted to ensure soundness of definite well-formedness and completed to ensure completeness of possible well-formedness.

136.0 $wf\text{-}DefGroup : \Pi \xrightarrow{t} ValFnOpDef\text{-set} \to VisibleEnv \to Env \to \mathbb{B}$

.1 $wf\text{-}DefGroup\,(\pi)(dg)(denv)(env) \triangleq$
.2 $\exists\, dl : ValFnOpDefs, deps, essential, order : DefDepRel \cdot$
.3 $\exists\, defnames, useful : Name\text{-set}, saenv : VisibleEnv \cdot$
.4 $IsOrderingOf(dg, dl)(order) \wedge$
.5 $deps = ValFnOpTpDefDep(dl) \wedge$
.6 $IsSafeAssertedVisEnv(\pi)(dl)(saenv)(env) \wedge$
.7 $defnames = DefNamesInDefs(dl) \wedge$
.8 $useful = \{n \mid n \in \mathsf{dom}\ saenv \cdot$
.9 $\exists\, t : TypeR \cdot IsBoundToValTypeR(\pi)(n)(t)(saenv) \wedge t \neq \text{ANY}\} \wedge$
.10 $essential =$
.11 $\{mk\text{-}(d_1, d_2) \mid mk\text{-}(d_1, d_2) \in deps \cdot \neg\, DefNamesInDef(d_2) \subseteq useful\} \wedge$
.12 $order = FindDepOrdering(essential) \wedge$
.13 $wf\text{-}OrderedDefs(\pi)(dl, \mathsf{true})(denv)(UpdateEnv(saenv)(RemoveFromEnv(defnames)(env)))$

The definitions of the group are now considered in an order based on "essential" dependencies between the definitions. A dependency is essential iff there is no "useful" information in the environment regarding the type of the entity depended on. The type ANY is not useful because it says nothing about the entity except that it is not bottom. All other information regarding the type of an entity is considered useful. From the essential dependencies, a partial order is defined by disregarding enough of the dependencies to ensure acyclicity. Based on the resulting partial order, the definitions in the group are sorted topologically and verified one by one, in order, starting with the definitions which do not depend on any other definition. After verification of a definition, the names defined and their types are then added to the environment which is used when verifying the following definitions.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

137.0 $wf\text{-}OrderedDefs : \Pi \xrightarrow{t} NonTpDefs \times \mathbb{B} \to VisibleEnv \to Env \to \mathbb{B}$

.1 $wf\text{-}OrderedDefs\,(\pi)(ds, envmerge)(denv)(env) \triangleq$
.2 cases ds :
.3 $([]) \to denv = \{\mapsto\}$,
.4 $preceding \frown [last] \to$
.5 $\exists\, denv_1, denv_2 : VisibleEnv \cdot$
.6 $wf\text{-}OrderedDefs(\pi)(preceding, envmerge)(denv_1)(env) \wedge$
.7 $wf\text{-}NonTpDef(\pi)(last)(denv_2)(UpdateEnv(denv_1)(env)) \wedge$
.8 if $envmerge$
.9 then $IsVisEnvMerge(\pi)(denv_1, denv_2)(denv)(env)$
.10 else $denv = UpdateVisEnv(denv_2)(denv_1)$
.11 end

Note that the same name may be defined by different simultaneous value definitions in a let expression or statement as long as it is bound to the same (semantic) value (137.9).

138.0 $IsVisEnvMerge : \Pi \to VisibleEnv \times VisibleEnv \to VisibleEnv \to Env \to \mathbb{B}$

.1 $IsVisEnvMerge\,(\pi)(venv_1, venv_2)(venv)(env) \triangleq$
.2 $\exists\, valenv, valenv_1, valenv_2, rest_1, rest_2 : VisibleEnv \cdot$
.3 $venv_1 = valenv_1 \cup rest_1 \wedge venv_2 = valenv_2 \cup rest_2 \wedge$
.4 $\forall\, ssr \in \mathsf{rng}\ valenv_1 \cup \mathsf{rng}\ valenv_2 \cdot is\text{-}ValR(ssr) \wedge$
.5 $IsVenvMerge(\pi)(valenv_1, valenv_2)(valenv)(env) \wedge$
.6 $venv = valenv \cup rest_1 \cup rest_2$

14.2.2.3 Definitions

139.0 $wf\text{-}NonTpDef : \Pi \xrightarrow{t} NonTpDef \to VisibleEnv \to Env \to \mathbb{B}$

.1 $wf\text{-}NonTpDef\,(\pi)(d)(denv)(env) \triangleq$
.2 $is\text{-}ValueDef(d) \wedge wf\text{-}ValueDef(\pi)(d)(denv)(env) \vee$
.3 $is\text{-}ExplFnDef(d) \wedge wf\text{-}ExplFnDef(\pi)(d)(denv)(env) \vee$
.4 $is\text{-}ImplFnDef(d) \wedge wf\text{-}ImplFnDef(\pi)(d)(denv)(env) \vee$
.5 $is\text{-}ExplOprtDef(d) \wedge wf\text{-}ExplOprtDef(\pi)(d)(denv)(env) \vee$
.6 $is\text{-}ImplOprtDef(d) \wedge wf\text{-}ImplOprtDef(\pi)(d)(denv)(env) \vee$
.7 $is\text{-}DefBind(d) \wedge wf\text{-}DefBind(\pi)(d)(denv)(env)$;

140.0 $wf\text{-}ValueDef : \Pi \xrightarrow{t} ValueDef \to VisibleEnv \to Env \to \mathbb{B}$

.1 $wf\text{-}ValueDef\,(\pi)(mk\text{-}ValueDef(pat, tp, val))(denv)(env) \triangleq$
.2 let $pb = $ if $tp = $ nil then pat else $mk\text{-}TypeBind(pat, tp)$ in
.3 $\exists\, t : TypeR \cdot$
.4 $wf\text{-}Expr(\pi)(val)(t)(env) \wedge$
.5 $wf\text{-}PatternBind(\pi)(pb)(mk\text{-}TVE(t, denv))(env)$;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
141.0 wf-ExplFnDef : Π →t ExplFnDef → VisibleEnv → Env → 𝔹
  .1 wf-ExplFnDef (π)(d)(denv)(env) ≜
  .2   d.id = d.idrepeated ∧
  .3   ∃ n : Name, ssr : StaticSemanticRep, tvs : TypeVar*, at : FunctionTypeR ·
  .4       IsAssertedFromDef(π)(d)({n ↦ ssr})(env) ∧
  .5       IsFnRep(tvs, at)(ssr) ∧
  .6       ∃ lam : LambdaR, vt : FunctionTypeR, ssr2 : StaticSemanticRep ·
  .7           IsCurriedFn(lam)(d.parms, at, d.fnpre, d.body)(env) ∧
  .8           wf-LambdaR(π)(lam)(vt)(UpdateEnv(elems tvs)(env)) ∧
  .9           IsSubtype(π)(vt, at)(env) ∧
  .10          IsFnRep(tvs, vt)(ssr2) ∧
  .11          denv = {n ↦ ssr2};

142.0 IsFnRep : TypeVar* × FunctionTypeR →t StaticSemanticRep → 𝔹
  .1 IsFnRep (tvs, t)(ssr) ≜
  .2   ∃ π : Π ·
  .3       tvs = [] ∧ ssr = mk-ValR(t, π) ∨
  .4       tvs ≠ [] ∧ ssr = mk-PolyValR(tvs, mk-ValR(t, π));

143.0 wf-ImplFnDef : Π →t ImplFnDef → VisibleEnv → Env → 𝔹
  .1 wf-ImplFnDef (π)(ifd)(denv)(env) ≜
  .2   let mk-ImplFnDef(id, tpparms, partps, residtype, fnpre, fnpost) = ifd in
  .3   ∃ n : Name, ft : TotalFnTypeR ·
  .4       ¬ IsEmptyType(NegWfClass(π))(ft)(env) ∧
  .5       denv = {n ↦ mk-ValR(ft, π)} ∧ IsAssertedFromDef(π)(ifd)(denv)(env) ∧
  .6       ∃ x : Name, fnpre', fnpost' : Expr ·
  .7           x.name ≠ residtype.id ∧
  .8           IsLetExpandedParTypes(fnpre')(partps, x, fnpre)(env) ∧
  .9           IsLetExpandedParTypes(fnpost')(partps, x, fnpost)(env) ∧
  .10          let venv_pre = {x ↦ mk-ValR(ft.fndom, π)},
  .11              env_pre = UpdateEnv(venv_pre)(env),
  .12              venv_post = {mk-Name(residtype.id) ↦ mk-ValR(ft.fnrng, π)},
  .13              env_post = UpdateEnv(venv_post)(env_pre) in
  .14          PreNeverTrue(π)(fnpre')(env_pre) ∨
  .15          cases π :
  .16            mk-DEF(-) → false,
  .17            mk-POS(-) → PreAlwaysTrue(NegWfClass(π))(fnpre')(env_pre) ⇒
  .18                        let locvars = dom venv_pre ∪ dom venv_post,
  .19                            wfe = (InScope[Expr](π)(locvars)) ∘
  .20                                  (InType[Expr](π)(wf-Expr(π))) in
  .21                        wfe(fnpost')(BOOLEAN)(env_post)
  .22          end;

144.0 wf-DefBind : Π →t DefBind → VisibleEnv → Env → 𝔹
  .1 wf-DefBind (π)(mk-DefBind(pb, expr))(denv)(env) ≜
  .2   ∃ t : TypeR ·
  .3       wf-Expr(π)(expr)(t)(env) ∧
  .4       wf-PatternBind(π)(pb)(mk-TVE(t, denv))(env)
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Well-formedness of Operation Definitions

```
145.0 wf-ExplOprtDef : Π →t ExplOprtDef → VisibleEnv → Env → 𝔹
  .1  wf-ExplOprtDef (π)(eod)(denv)(env) ≜
  .2    let mk-ExplOprtDef(id, optype, idrepeated, params, body, oppre) = eod,
  .3        env' = UpdateEnv(StateVisEnv(env))(env),
  .4        p = Pats2Pat(params) in
  .5    id = idrepeated ∧
  .6    ∃ n : Name, ot : OpTypeR ·
  .7        denv = {n ↦ mk-ValR(ot, π)} ∧ IsAssertedFromDef(π)(eod)(denv)(env') ∧
  .8        ∃ x : Name, oppre' : Expr, body' : Stmt ·
  .9            IsLetExpansion(oppre')(p, nil, x, oppre)(env') ∧
  .10           IsLetExpansion(body')(p, nil, x, body)(env') ∧
  .11           let venv = {x ↦ mk-ValR(ot.opdom, π)},
  .12               newenv = UpdateEnv(venv)(env') in
  .13           PreNeverTrue(π)(oppre')(newenv) ∨
  .14           cases π :
  .15             mk-DEF(cs) →
  .16                 wf-Stmt(mk-DEF(cs))(body)(ot.oprng)(newenv) ∨
  .17                 ¬ ∃ body_t : StmtTypeR ·
  .18                     let π_POS = mk-POS(cs),
  .19                         wfs = InScope[Stmt](π_POS)(dom venv)(wf-Stmt(π_POS)) in
  .20                     wfs(body')(body_t)(newenv),
  .21             mk-POS(cs) →
  .22                 PreAlwaysTrue(NegWfClass(π))(oppre')(newenv) ⇒
  .23                 ¬ ∃ body_t : StmtTypeR ·
  .24                     wf-Stmt(mk-DEF(cs))(body')(body_t)(newenv) ∧
  .25                     IsDisjoint(mk-DEF(cs))(body_t, ot.oprng)(env')
  .26           end
```

An explicit operation definition is well-formed if the pre-condition is never true (145.13). Otherwise the well-formedness is dependent of the check class. An explicit operation definition is definitely well-formed if the body yields a result of the operation's range (statement) type (145.16), or if the operation is the everywhere undefined operation (145.17-.20). An explicit operation definition is possibly well-formed if the body does not yield a result which definitely is disjoint from the operation's range (statement) type (145.22-.25).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
146.0 wf-ImplOprtDef : Π -t-> ImplOprtDef → VisibleEnv → Env → 𝔹
  .1 wf-ImplOprtDef (π)(iod)(denv)(env) ≙
  .2   let mk-ImplOprtDef(id, params, residtype, opext, oppre, oppost, excps) = iod,
  .3       env' = UpdateEnv(StateVisEnv(env))(env),
  .4       oppre' = PreOExceptions2Pre(oppre, excps),
  .5       oppost' = PostOExceptions2Post(oppost, excps) in
  .6   ∃ n : Name, ot : OpTypeR ·
  .7       denv = {n ↦ mk-ValR(ot, π)} ∧ IsAssertedFromDef(π)(iod)(denv)(env') ∧
  .8       wf-OExternals(π)(opext)(env') ∧
  .9       ∃ x : Name, oppre'', oppost'' : Expr ·
  .10          (residtype ≠ nil ⇒ x.name ≠ residtype.id) ∧
  .11          IsLetExpandedParTypes(oppre'')(params, x, oppre')(env') ∧
  .12          IsLetExpandedParTypes(oppost'')(params, x, oppost')(env') ∧
  .13          let venv1 = {x ↦ mk-ValR(ot.opdom, π)},
  .14              venv2 = if residtype = nil
  .15                      then { ↦}
  .16                      else {mk-Name(residtype.id) ↦ mk-ValR(ot.oprng, π)},
  .17              env_pre = UpdateEnv(venv1)(env'),
  .18              venv3 = {mk-OldName(nn.name) ↦ ssr |
  .19                           nn ∈ StateVarNames(env'), ssr : StaticSemanticRep ·
  .20                               IsBoundTo(nn)(ssr)(env')},
  .21              env_post = UpdateEnv(venv3 ⩌ venv2)(env_pre) in
  .22          PreNeverTrue(π)(oppre'')(env_post) ∨
  .23          cases π :
  .24            mk-DEF(-) → false,
  .25            mk-POS(-) →
  .26                PreAlwaysTrue(NegWfClass(π))(oppre')(env_pre) ⇒
  .27                let locvars = dom venv1 ∪ dom venv2 ∪ dom venv3,
  .28                    wfs = (InScope[Expr](π)(locvars)) ∘ (InType[Expr](wf-Expr(π))) in
  .29                wfs(oppost'')(BOOLEAN)(env_post)
  .30          end
```

An implicit operation definition is well-formed if the pre-condition is never true (146.22). If it cannot be rejected that the pre-condition is true for all arguments, the operation definition is not definitely well-formed since the specification may be unsatisfiable (146.24). An implicit operation definition is rejected if the post-condition never has a boolean value. This is a safe way to state that it never is true (146.26-.29). Note that the error definitions are checked together with the pre- and post-condition (146.4–146.5).

```
147.0 wf-OExternals : Π -t-> OExternals → Env → 𝔹
  .1 wf-OExternals (π)(exts)(env) ≙
  .2   ∀ varinf ∈ elems exts · wf-VarInf(π)(varinf)(env);
```

```
148.0 wf-VarInf : Π -t-> VarInf → Env → 𝔹
  .1 wf-VarInf (π)(mk-VarInf(mode, vars, type))(env) ≙
  .2   mode = READ ⇒
  .3   ∀ v ∈ elems vars · v ∈ dom env.visibleenv
```

14.2.2.4. Definition Groups and their Ordering

A sequence of definitions can be partitioned into groups according to the following rules. Two definitions are in the same group if and only if they are mutually recursive (mutually dependent). Each definition which is not mutually recursive with any other definition forms a singleton group. The groups can then be sorted topologically according to

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

induced dependencies: A group A depends on another B iff at least one definition in A depends on an entity defined by a definition in B.

149.0 $OrderedDefGroups\,(ds : ValFnOpDefs, env : Env)\ odg : ValFnOpDef\text{-set}^*$
.1 post let $deprel = ValFnOpTpTransDefDep(ds)$,
.2 $simrel = SimultaneousDefRel(\text{elems } ds, deprel)$,
.3 $groupset = Partition[ValFnOpDef](\text{elems } ds, simrel)$,
.4 $order = InducedPartitionOrder[ValFnOpDef](groupset, deprel)$ in
.5 len $odg = \text{card } groupset \wedge \text{elems } odg = groupset \wedge$
.6 $IsPartiallyOrdered[ValFnOpTpDef\text{-set}](odg)(order)$;

150.0 $SimultaneousDefRel : ValFnOpDef\text{-set} \times DefDepRel \xrightarrow{t} DefDepRel$
.1 $SimultaneousDefRel\,(dset, drel) \triangleq$
.2 $\{mk\text{-}(d, d) \mid d \in dset\} \cup$
.3 $\{mk\text{-}(d_1, d_2) \mid d_1, d_2 \in dset \cdot \{mk\text{-}(d_1, d_2), mk\text{-}(d_2, d_1)\} \subseteq drel\}$;

151.0 $InducedPartitionOrder[@T]\,(ss : @T\text{-set-set}, rel : (@T \times @T)\text{-set})\ srel : (@T\text{-set} \times @T\text{-set})\text{-set}$
.1 post $\forall s_1, s_2 \in ss \cdot$
.2 $mk\text{-}(s_1, s_2) \in srel \ \Leftrightarrow$
.3 $\exists e_1 \in s_1, e_2 \in s_2 \cdot mk\text{-}(e_1, e_2) \in rel$;

152.0 $Partition[@T]\,(s : @T\text{-set}, rel : (@T \times @T)\text{-set})\ ss : @T\text{-set-set}$
.1 post $\bigcup ss = s \wedge \{\} \notin ss \wedge$
.2 $\forall s_1, s_2 \in ss \cdot$
.3 $s_1 \cap s_2 = \{\} \vee s_1 = s_2 \wedge$
.4 $\forall e_1, e_2 \in s \cdot$
.5 $mk\text{-}(e_1, e_2) \in rel \ \Leftrightarrow$
.6 $\exists s \in ss \cdot \{e_1, e_2\} \subseteq s$

14.2.2.5 Ordering of Definitions

Given a relation recording the dependencies between definitions, a partial ordering of the definitions can be defined. The problem here is to avoid the circularities (symmetries) caused by mutual dependencies. By disregarding some of the dependencies in the dependency relation, circularities in the transitive closure of the dependencies may be eliminated. The choice of which of the original dependencies to disregard is left open (by under-specification) as long as none of the disregarded dependencies could be added without causing circularity. Having eliminated the possibility of circularities in the transitive closure, the result is a partial order which may be used to order the definitions.

153.0 $FindDepOrdering\,(rel : DefDepRel)\ order : DefDepRel$
.1 post $\exists\, subrel : DefDepRel \cdot$
.2 $subrel \subseteq rel \wedge order = TransClosure[ValFnOpTpDef](subrel) \wedge$
.3 $IsAsymmetric[ValFnOpTpDef](order) \wedge$
.4 $\forall\, pair \in rel \cdot$
.5 $pair \notin subrel \ \Rightarrow$
.6 $\neg\, IsAsymmetric[ValFnOpTpDef](TransClosure[ValFnOpTpDef](rel \cup \{pair\}))$;

154.0 $IsAsymmetric[@T] : (@T \times @T)\text{-set} \xrightarrow{t} \mathbb{B}$
.1 $IsAsymmetric\,(rel) \triangleq$
.2 $\forall x, y : @T \cdot$
.3 $mk\text{-}(x, y) \in rel \ \Rightarrow \ mk\text{-}(y, x) \notin rel$;

155.0 $IsOrderingOf : ValFnOpDef\text{-set} \times ValFnOpDefs \xrightarrow{t} DefDepRel \to \mathbb{B}$
.1 $IsOrderingOf\,(dg, dl)(order) \triangleq$
.2 elems $dl = dg \wedge$ len $dl =$ card $dg \wedge$
.3 $IsPartiallyOrdered[ValFnOpTpDef](dl)(order)$;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

156.0 $IsPartiallyOrdered[@T] : @T^* \xrightarrow{t} (@T \times @T)\text{-set} \rightarrow \mathbb{B}$

.1 $IsPartiallyOrdered\,(s)(order) \triangleq$
.2 $\quad \forall\, i_1, i_2 \in \text{inds } s \cdot$
.3 $\quad\quad mk\text{-}(s(i_2), s(i_1)) \in order \;\Rightarrow\; i_1 \leq i_2$

14.2.2.6 Asserted Visible Environments

The Static Semantics of a group of simultaneous definitions is based on the explicitly declared (asserted) types which are included in the definitions. In connection with definite well-formedness it is, however, potentially unsound to include assumptions about (non-function) value types and total function types. Value types are therefore not included and total function types are weakened to partial ones. Also, it is potentially unsound to include any assumptions about definite types at all, unless all the definitions in the group belong to one class of definitions which in the Dynamic Semantics is given a least fixed point semantics. Moreover, for possible well-formedness, it would be incomplete if an entity being defined was not included in the environment. Therefore, value definitions without explicitly stated types are treated as if the type ANY had been stated.

157.0 $IsSafeAssertedVisEnv : \Pi \xrightarrow{t} ValFnOpDefs \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsSafeAssertedVisEnv\,(\pi)(ds)(saenv)(env) \triangleq$
.2 $\quad$ if $(\forall\, d \in \text{elems } ds \cdot$
.3 $\quad\quad is\text{-}ValueDef(d) \vee is\text{-}ExplFnDef(d) \wedge d.tpparms = [])\ \vee$
.4 $\quad \forall\, d \in \text{elems } ds \cdot$
.5 $\quad\quad is\text{-}ExplFnDef(d) \wedge d.tpparms \neq [] \vee$
.6 $\quad is\text{-}POS(\pi)$
.7 $\quad$ then $IsSafeAssertedFromDefs(\pi)(ds)(saenv)(env)$
.8 $\quad$ else $IsSafeAssertedFromDefs(NegWfClass(\pi))(ds)(saenv)(env)$;

158.0 $IsSafeAssertedFromDefs : \Pi \xrightarrow{t} ValFnOpDefs \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsSafeAssertedFromDefs\,(\pi)(ds)(saenv)(env) \triangleq$
.2 $\quad$ if $ds = []$ then $saenv = \{\mapsto\}$
.3 $\quad$ else $\exists\, saenv_1, saenv_2 : VisibleEnv \cdot$
.4 $\quad\quad IsSafeAssertedFromDef(\pi)(\text{hd } ds)(saenv_1)(env) \wedge$
.5 $\quad\quad IsSafeAssertedFromDefs(\pi)(\text{tl } ds)(saenv_2)(env) \wedge$
.6 $\quad\quad IsVisEnvMerge(\pi)(saenv_1, saenv_2)(saenv)(env)$;

159.0 $IsSafeAssertedFromDef : \Pi \xrightarrow{t} ValFnOpDef \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$

.1 $IsSafeAssertedFromDef\,(\pi)(d)(saenv)(env) \triangleq$
.2 $\quad \exists\, aenv : VisibleEnv \cdot$
.3 $\quad\quad IsAssertedFromDef(\pi)(d)(aenv)(env) \wedge$
.4 $\quad\quad saenv = \{n \mapsto SafeEnvEntry(\pi)(aenv(n))(is\text{-}ValueDef(d)) \mid n \in \text{dom } aenv\}$;

160.0 $SafeEnvEntry : \Pi \xrightarrow{t} StaticSemanticRep \rightarrow \mathbb{B} \rightarrow StaticSemanticRep$

.1 $SafeEnvEntry\,(\pi)(ssr)(origins\text{-}from\text{-}valuedef) \triangleq$
.2 $\quad$ if $is\text{-}POS(\pi)$ then ssr
.3 $\quad$ else cases ssr :
.4 $\quad\quad mk\text{-}ValR(t, \text{-}) \rightarrow$
.5 $\quad\quad\quad$ if $Is[FunctionTypeR, TypeR](t) \wedge \neg\, origins\text{-}from\text{-}valuedef$
.6 $\quad\quad\quad$ then $mk\text{-}ValR(MakePartialFnTypeR(t), \pi)$
.7 $\quad\quad\quad$ else $mk\text{-}ValR(t, NegWfClass(\pi))$,
.8 $\quad\quad mk\text{-}PolyValR(tppars, valr) \rightarrow$
.9 $\quad\quad\quad mk\text{-}PolyValR(tppars, SafeEnvEntry(\pi)(valr)(origins\text{-}from\text{-}valuedef))$,
.10 $\quad\quad$ others $\rightarrow ssr$
.11 $\quad$ end;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

161.0 $MakePartialFnTypeR : FunctionTypeR \xrightarrow{t} PartialFnTypeR$

.1 $MakePartialFnTypeR\,(ft) \triangleq$
.2 $\quad mk\text{-}PartialFnTypeR(ft.fndom, ft.fnrng);$

162.0 $IsAssertedVisEnv : \Pi \xrightarrow{t} ValFnOpDefs \to VisibleEnv \to Env \to \mathbb{B}$

.1 $IsAssertedVisEnv\,(\pi)(ds)(denv)(env) \triangleq$
.2 $\quad ds = [] \wedge denv = \{\mapsto\} \vee$
.3 $\quad \exists\, denv_1, denv_2 : VisibleEnv \cdot$
.4 $\qquad IsAssertedVisEnv(\pi)(\mathsf{tl}\ ds)(denv_1)(env) \wedge$
.5 $\qquad IsAssertedFromDef(\pi)(\mathsf{hd}\ ds)(denv_2)(env) \wedge$
.6 $\qquad IsVisEnvMerge(\pi)(denv_1, denv_2)(denv)(env);$

163.0 $IsAssertedFromDef : \Pi \xrightarrow{t} ValFnOpDef \to VisibleEnv \to Env \to \mathbb{B}$

.1 $IsAssertedFromDef\,(\pi)(d)(denv)(env) \triangleq$
.2 $\quad \mathsf{cases}\ d :$
.3 $\qquad mk\text{-}ValueDef(pat, otype, \text{-}) \to$
.4 $\qquad\quad \exists\, t, et : TypeR \cdot$
.5 $\qquad\qquad wf\text{-}AssertedType(\pi)(otype)(t)(env) \wedge$
.6 $\qquad\qquad IsNonEmptyEssEquivTypeR(\pi)(et, t)(env) \wedge$
.7 $\qquad\qquad HasMatchingSubtype(\pi)(pat)(et)(denv)(env),$
.8 $\qquad mk\text{-}ExplFnDef(i, otvs, type, \text{-}, \text{-}, \text{-}) \to$
.9 $\qquad\quad \exists\, t, et : FunctionTypeR, vr : ValR \cdot$
.10 $\qquad\qquad wf\text{-}Type(\pi)(type)(t)(env) \wedge$
.11 $\qquad\qquad IsNonEmptyEssEquivTypeR(\pi)(et, t)(env) \wedge$
.12 $\qquad\qquad denv = MakeFnEntry(\pi)(i, HasNonTrivialPreCond(\pi)(d), otvs, et),$
.13 $\qquad mk\text{-}ImplFnDef(i, otvs, ps, mk\text{-}IdType(\text{-}, rtype), \text{-}, \text{-}) \to$
.14 $\qquad\quad \exists\, at, rt, ft, et : TypeR, vr : ValR \cdot$
.15 $\qquad\qquad wf\text{-}ParameterTypes(\pi)(ps)(at)(env) \wedge$
.16 $\qquad\qquad wf\text{-}Type(\pi)(rtype)(rt)(env) \wedge$
.17 $\qquad\qquad ft = mk\text{-}TotalFnTypeR(at, rt) \wedge$
.18 $\qquad\qquad IsNonEmptyEssEquivTypeR(\pi)(et, ft)(env) \wedge$
.19 $\qquad\qquad denv = MakeFnEntry(\pi)(i, HasNonTrivialPreCond(\pi)(d), otvs, et),$
.20 $\qquad mk\text{-}ExplOprtDef(i, otype, \text{-}, \text{-}, \text{-}, \text{-}) \to$
.21 $\qquad\quad \exists\, ot, et : OpTypeR \cdot$
.22 $\qquad\qquad wf\text{-}Type(\pi)(otype)(ot)(env) \wedge$
.23 $\qquad\qquad IsNonEmptyEssEquivTypeR(\pi)(et, ot)(env) \wedge$
.24 $\qquad\qquad denv = \{mk\text{-}Name(i) \mapsto mk\text{-}ValR(et, \pi)\},$
.25 $\qquad mk\text{-}ImplOprtDef(i, ps, ridtp, \text{-}, \text{-}, \text{-}, \text{-}) \to$
.26 $\qquad\quad \exists\, at, rt, ot, et : TypeR \cdot$
.27 $\qquad\qquad wf\text{-}ParameterTypes(\pi)(ps)(at)(env) \wedge$
.28 $\qquad\qquad (wf\text{-}Type(\pi)(ridtp.type)(rt)(env) \vee ridtp = \mathsf{nil} \wedge rt = \text{UNITTYPE}) \wedge$
.29 $\qquad\qquad ot = mk\text{-}OpTypeR(env.statename, at, rt) \wedge$
.30 $\qquad\qquad IsNonEmptyEssEquivTypeR(\pi)(et, ot)(env) \wedge$
.31 $\qquad\qquad denv = \{mk\text{-}Name(i) \mapsto mk\text{-}ValR(et, \pi)\}$
.32 $\quad \mathsf{end};$

164.0 $MakeFnEntry : \Pi \xrightarrow{t} Id \times \mathbb{B} \times TypeVar^* \times FunctionTypeR \to VisibleEnv$

.1 $MakeFnEntry\,(\pi)(i, haspre, otvs, t) \triangleq$
.2 $\quad \mathsf{let}\ vr = mk\text{-}ValR(\mathsf{if}\ haspre\ \mathsf{then}\ MakePartialFnTypeR(t)\ \mathsf{else}\ t, \pi),$
.3 $\qquad ssr = \mathsf{if}\ otvs = []\ \mathsf{then}\ vr\ \mathsf{else}\ mk\text{-}PolyValR(otvs, vr)\ \mathsf{in}$
.4 $\quad \{mk\text{-}Name(i) \mapsto ssr\};$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

165.0 $wf\text{-}ParameterTypes : \Pi \xrightarrow{t} ParameterTypes \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}ParameterTypes\,(\pi)(ps)(t)(env) \triangleq$
.2 let $atype = ParameterTypes2Type(ps)$ in
.3 $wf\text{-}Type(\pi)(atype)(t)(env)$;

166.0 $HasMatchingSubtype : \Pi \xrightarrow{t} Pattern \to TypeR \to VisibleEnv \to Env \to \mathbb{B}$

.1 $HasMatchingSubtype\,(\pi)(pat)(tp)(ve)(env) \triangleq$
.2 $\exists\, stp : TypeR \cdot$
.3 $IsSubtypeR(\pi)(stp, tp)(env) \wedge$
.4 $wf\text{-}Pattern(\pi)(pat)(mk\text{-}TVE(stp, ve))(env)$

The purpose of this predicate is to find types for the identifiers in a pattern given that the pattern matches some subtype of an asserted type. The types found are registered in a visible environment.

14.2.2.7 Pre-Conditions

167.0 $HasNonTrivialPreCond : \Pi \xrightarrow{t} FunctionDef \to \mathbb{B}$

.1 $HasNonTrivialPreCond\,(\pi)(d) \triangleq$
.2 $\neg\, is\text{-}DEF(\pi) \wedge$
.3 cases d :
.4 $mk\text{-}ExplFnDef(\text{-}, \text{-}, \text{-}, \text{-}, \text{-}, body, fnpre), mk\text{-}ImplFnDef(\text{-}, \text{-}, \text{-}, \text{-}, fnpre, \text{-}) \to$
.5 $fnpre \neq$ nil
.6 end;

168.0 $PreNeverTrue : \Pi \xrightarrow{t} [Expr] \to Env \to \mathbb{B}$

.1 $PreNeverTrue\,(\pi)(opre)(env) \triangleq$
.2 $opre \neq$ nil $\wedge$
.3 cases π :
.4 $mk\text{-}POS(\text{-}) \to$ true,
.5 $mk\text{-}DEF(cs) \to \neg\, InType[Expr](mk\text{-}POS(cs))(wf\text{-}Expr(mk\text{-}POS(cs)))(opre)(\text{BOOLEAN})(env)$
.6 end;

169.0 $PreAlwaysTrue : \Pi \xrightarrow{t} [Expr] \to Env \to \mathbb{B}$

.1 $PreAlwaysTrue\,(\pi)(opre)(env) \triangleq$
.2 $opre =$ nil $\vee\ is\text{-}POS(\pi)$

14.3 Expressions

In this clause the well-formedness predicates for expressions and the different expression constructs are defined. The definition of the basic well-formedness predicate for an expression construct ($XExpr$) has the form:

$wf\text{-}XExpr : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$
$wf\text{-}XExpr\,(\pi)(e)(t)(env) \triangleq$
$body$

Here π is the well-formedness classification parameter used to distinguish between different kinds of well-formedness checks, the main distinction being between definite and possible well-formedness.

The well-formedness predicates for the different expression constructs will be defined in one of two ways: either directly in terms of for instance the subtype predicate and the type and expression parameters or indirectly in terms of so-called "expression characteristic predicates". Characteristic predicates are only used for strict compound expressions which do not introduce new identifiers.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.1 Expression Characteristic Predicates

A characteristic predicate for an expression defines the relationship between the types of the subexpressions and the type of the whole expression:

170.0 $ExprCP = TypeR^* \to TypeR \to Env \to \mathbb{B}$

On the basis of such a characteristic predicate, the well-formedness of an expression with given subexpressions is defined as follows:

```
171.0 wf-SCompExpr : Π -t-> ExprCP → Expr* → TypeR → Env → 𝔹
  .1  wf-SCompExpr (π)(CP)(exprs)(restype)(env) ≙
  .2    ∃ exprtypes : TypeR* ·
  .3        len exprs = len exprtypes
  .4        ∧
  .5        (∀ i ∈ inds exprs · wf-Expr(π)(exprs(i))(exprtypes(i))(env))
  .6        ∧
  .7        cases π :
  .8          mk-DEF({(UNIONCLOSE)} ∪ -) →
  .9              UnionCloseExprCP(CP)(exprtypes)(restype)(env),
  .10         mk-DEF(-) → CP(exprtypes)(restype)(env),
  .11         mk-POS(-) → ComplCP(π)(CP)(exprtypes)(restype)(env)
  .12       end
```

wf-SCompExpr checks that the sequence of component expressions *exprs* from some compound expression is well-formed with respect to the expression characteristic predicate specified for the considered expression and the expression's result type *restype*.

Note that the expression characteristic predicate is assumed to be defined so it just exhibits the basic type property of the considered expression construct. Therefore, it is completed and relaxed in certain ways before it is used (*ComplCP* and *UnionCloseExprCP*).

```
172.0 ComplCP : Π -t-> ExprCP → ExprCP
  .1  ComplCP (π)(CP)(ets)(rt)(env) ≙
  .2    let liftedtoCP : TypeR* -t-> 𝔹
  .3        liftedtoCP (cts) ≙
  .4          len ets = len cts ∧
  .5          ∀ i ∈ inds ets · IsSubtypeR(π)(ets(i), cts(i))(env) ∧
  .6          ∃ cprt : TypeR · CP(cts)(cprt)(env)
  .7        in
  .8    ∃ cts : TypeR* ·
  .9        liftedtoCP(cts) ∧ CP(cts)(rt)(env) ∧
  .10       ∀ octs : TypeR* ·
  .11           liftedtoCP(octs) ⇒
  .12           ∀ i ∈ inds cts ·
  .13               IsSubtypeR(π)(octs(i), cts(i))(env) ⇒ IsSubtypeR(π)(cts(i), octs(i))(env)
```

ComplCP$(\pi)(CP)$ completes the expression characteristic predicate *CP* so it satisfies a completeness property in connection with possible well-formedness: If $CP(pt_1, \ldots, pt_n)(rt)(env)$ is true and the types t_i are subtypes of pt_i then $(ComplCP(CP))(t_1, \ldots, t_n)(rt)(env)$ will also be true. Note that in order not to loosen the characteristic predicate so it may accept arbitrarily great result types the requirement in lines 172.10-.13 is added.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.2 Relaxations and Restriction of Predicates

Both expression characteristic predicates and well-formedness predicates may be defined by transforming other such predicates. The transformations will in most cases be done by use of higher order functions. This clause defines and explains the different transformations.

Expression characteristic predicates are transformed to new predicates by functions of type $ExprCP \rightarrow ExprCP$. All modifications of expression characteristic predicates are relaxations, i.e. the new predicate holds for more combinations of component types and result type.

An expression well-formedness predicate corresponding to some specific check $\pi : \Pi$ has the type WFE, where

173.0 $WFE = Expr \rightarrow WF;$

174.0 $WF = TypeR \rightarrow Env \rightarrow \mathbb{B}$

Some of the transformations defined for expression well-formedness predicates will also be used on the well-formedness predicates for the *statement* constructs. A well-formedness predicate defined solely for statements would have a type $Stmt \rightarrow WF$.

A function which only transforms expression well-formedness predicates has the type $\Pi \rightarrow WFE \rightarrow WFE$. A function which is used to transform both expression and statement well-formedness predicates has the "type"

$$\Pi \rightarrow (@C \rightarrow WF) \rightarrow @C \rightarrow WF$$

where $@C$ may be replaced by one of the Construct types *Expr* and *Stmt*.

14.3.2.1 The Subsumption Rule

The static semantics for the different expression constructs are defined by so-called basic well-formedness predicates, which define the basic type properties (see Formula 182). For most constructs these predicates will only be true for a finite set of types. In order to be able to check that an expression yields values within some set of values required by the given context another expression well-formedness predicate is introduced: *wf-Expr*(see formula 183) is defined by transforming the basic expression well-formedness predicate according to the subsumption rule (se Formula 175).This means that if $wf\text{-}Expr(\pi)(e)(t)(env)$ is true for some type t then it is also true for all types equivalente to t. Furthermore, for definite well-formedness (i.e. $is\text{-}DEF(\pi)$) if $wf\text{-}Expr(\pi)(e)(t)(env)$ is true for some type t it is also true for all types t' such that $IsSubtypeR(\pi)(t,t')(env)$ *and* for possible well-formednes (i.e. $is\text{-}POS(\pi)$) if $wf\text{-}Expr(\pi)(e)(t)(env)$ is true for some type t it is also true for all types t' such that $IsSubtypeR(\pi)(t',t)(env)$.

175.0 $SubsumeExpr : \Pi \xrightarrow{t} WFE \rightarrow WFE$

.1 $SubsumeExpr\,(\pi)(wfe)(expr)(rtype)(env) \triangleq$

.2 $\quad \exists\, et : TypeR \cdot$

.3 $\quad\quad wfe(expr)(et)(env) \wedge Subsume(\pi)(et, rtype)(env);$

176.0 $Subsume : \Pi \rightarrow TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $Subsume\,(\pi)(source\text{-}t, expected\text{-}t)(env) \triangleq$

.2 cases π :

.3 $\quad mk\text{-}DEF(\text{-}) \rightarrow IsSubtypeR(\pi)(source\text{-}t, expected\text{-}t)(env),$

.4 $\quad mk\text{-}POS(\text{-}) \rightarrow IsSubtypeR(\pi)(expected\text{-}t, source\text{-}t)(env)$

.5 end

14.3.2.2 The InType Rule

When a subcomponent expression in the given construct is required to yield a value in a specific type *ct* the *InType* rule is applied in addition to the subsumption rule. Let *wfe* be some expression well-formedness predicate. The

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

transformed predicate $InType(\pi)(wfe)$ will for $is\text{-}DEF(\pi)$ behave exactly as the untransformed predicate wfe. However, for possible well-formedness, i.e. $is\text{-}POS(\pi)$, $InType(\pi)(wfe)(e)(ct)(env)$ is true if some subtype est of ct exist such that $wfe(e)(est)(env)$ is true.

Let wfe be some well-formedness predicate which includes the subsumption rule. Rejection can now be based on the following simple rule. If

$$\neg\, InType(\pi)(wfe)(e)(ct)(env)$$

where $is\text{-}POS(\pi)$, then it may be *rejected* that the expression e evaluates to a value in the type ct.

```
177.0 InType[@C] : Π →t (@C → WF) → @C → WF
   .1 InType (π)(wfc)(c)(expectedt)(env) ≜
   .2   ∃ rt : TypeR ·
   .3         wfc(c)(rt)(env) ∧
   .4         cases π :
   .5           mk-DEF(-) → rt = expectedt,
   .6           mk-POS(-) → IsSubtypeR(π)(rt, expectedt)(env)
   .7         end
```

14.3.2.3 Union Close

This clause defines a relaxation of expression characteristic predicates which is useful for the definite well-formedness check of expressions whose subexpressions are well-formed with respect to union types. The whole expression may then also be well-formed with respect to a union type:

```
178.0 UnionCloseExprCP : ExprCP →t ExprCP
   .1 UnionCloseExprCP (cp)(ats)(rt)(env) ≜
   .2   ∀ aats : TypeR* ·
   .3         len aats = len ats ∧
   .4         ∀ i ∈ inds aats ·
   .5               IsUnionComponent(aats(i), ats(i))(env)
   .6           ⇒
   .7         ∃ art : TypeR ·
   .8               IsUnionComponent(art, rt)(env) ∧ cp(aats)(art)(env)
```

14.3.2.4 Sub-Environment Based Checks

The higher order functions described in this clause transform well-formedness predicates for possible well-formedness to more restricted predicates which perform a case analyses on the types of identifiers which have union types in the given environment. If an identifier has a union type then it must denote a value in one of the summands of the union type. Therefore, it may be required that well-formedness can be obtained for some environment where the identifier has one of the summand types (*IsSubEnvLoc*):

```
179.0 InScope[@C] : Π →t Name-set → (@C → WF) → @C → WF
   .1 InScope (π)(locnames)(wfc)(c)(type)(env) ≜
   .2   cases π :
   .3     mk-POS({(LOCALCONTEXT)} ∪ -) →
   .4           ∃ subenv : Env ·
   .5                 IsSubEnvLoc(π)(locnames)(subenv, env) ∧ wfc(c)(type)(subenv),
   .6     others → wfc(c)(type)(env)
   .7   end
```

The next function concerns check of *total functions* (without preconditions). In general, for a function defined to be

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

total (and without preconditions) the body must evaluate to a value in the declared result type for all bindings of formal parameters to values in their declared type. This requirement is in the static semantics approximated in the following way: for the function body to be possibly well-formed it must be possibly well-formed in *all sub-environments* with respect to the formal parameter names:

```
180.0 InTotalFnScope : Π →t Name-set → WFE → WFE
  .1  InTotalFnScope (π)(locnames)(wfe)(expr)(type)(env) ≜
  .2    cases π :
  .3      mk-POS({(TOTALFUNC)} ∪ -) →
  .4          ∀ subenv : Env ·
  .5              IsSubEnvLoc(π)(locnames)(subenv, env) ∧ wfe(expr)(type)(subenv),
  .6      others → InScope[Expr](π)(locnames)(wfe)(expr)(type)(env)
  .7    end
```

Sub-Environments

```
181.0 IsSubEnvLoc : Π →t Name-set → Env × Env → 𝔹
  .1  IsSubEnvLoc (π)(locnames)(subenv, env) ≜
  .2    subenv = μ (env, visibleenv ↦ subenv.visibleenv) ∧
  .3    let subVenv = subenv.visibleenv,
  .4        Venv = env.visibleenv in
  .5    dom subVenv = dom Venv ∧
  .6    ∀ name ∈ dom Venv \ locnames · subVenv(name) = Venv(name) ∧
  .7    ∀ name ∈ dom Venv ∩ locnames ·
  .8        cases mk-(subVenv(name), Venv(name)) :
  .9          mk-(mk-ValR(subt, -), mk-ValR(t, -)) → IsUnionComponent(subt, t)(env),
  .10         mk-(mk-VarR(subt), mk-VarR(t)) → IsUnionComponent(subt, t)(env),
  .11         mk-(ssrep1, ssrep2) → ssrep1 = ssrep2
  .12       end
```

IsSubEnvLoc defines a relation between two environments *subenv* and *env*. *subenv* is a subenvironment of *env* with respect to some local names *locnames* if the following conditions hold: the two environments are equal except for the visible environment part (line 181.2); the two visible environment maps have equal domains (line 181.5), and the two maps map identifiers not occuring in *locnames* to the same type (line 181.6). Finally, *subenv*'s visible environment will map identifiers occuring in *locnames* to types which are union components of the corresponding type in *env*'s visible environment.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.3 Well-Formedness of Expressions

182.0 $wf\text{-}ExprBasic : \Pi \xrightarrow{t} WFE$

.1 $wf\text{-}ExprBasic\,(\pi)(e)(t)(env) \triangleq$
.2 cases e :
.3 $mk\text{-}BracketedExpr(\text{-}) \rightarrow wf\text{-}BracketedExpr(\pi)(e)(t)(env)$,
.4 $mk\text{-}LetExpr(\text{-},\text{-}) \rightarrow wf\text{-}LetExpr(\pi)(e)(t)(env)$,
.5 $mk\text{-}LetBeSTExpr(\text{-},\text{-},\text{-}) \rightarrow wf\text{-}LetBeSTExpr(\pi)(e)(t)(env)$,
.6 $mk\text{-}DefExpr(\text{-},\text{-}) \rightarrow wf\text{-}DefExpr(\pi)(e)(t)(env)$,
.7 $mk\text{-}IfExpr(\text{-},\text{-},\text{-},\text{-}) \rightarrow wf\text{-}IfExpr(\pi)(e)(t)(env)$,
.8 $mk\text{-}CasesExpr(\text{-},\text{-},\text{-}) \rightarrow wf\text{-}CasesExpr(\pi)(e)(t)(env)$,
.9 $mk\text{-}PrefixExpr(\text{-},\text{-}) \rightarrow wf\text{-}PrefixExpr(\pi)(e)(t)(env)$,
.10 $mk\text{-}MapInverseExpr(\text{-}) \rightarrow wf\text{-}MapseExpr(\pi)(e)(t)(env)$,
.11 $mk\text{-}BinaryExpr(\text{-},\text{-},\text{-}) \rightarrow wf\text{-}BinaryExpr(\pi)(e)(t)(env)$,
.12 $mk\text{-}AllExpr(\text{-},\text{-}) \rightarrow wf\text{-}AllExpr(\pi)(e)(t)(env)$,
.13 $mk\text{-}ExistsExpr(\text{-},\text{-}) \rightarrow wf\text{-}ExistsExpr(\pi)(e)(t)(env)$,
.14 $mk\text{-}ExistsUniqueExpr(\text{-},\text{-}) \rightarrow wf\text{-}ExistsUniqueExpr(\pi)(e)(t)(env)$,
.15 $mk\text{-}IotaExpr(\text{-},\text{-}) \rightarrow wf\text{-}IotaExpr(\pi)(e)(t)(env)$,
.16 $mk\text{-}SetEnumeration(\text{-}) \rightarrow wf\text{-}SetEnumeration(\pi)(e)(t)(env)$,
.17 $mk\text{-}SetComprehension(\text{-},\text{-},\text{-}) \rightarrow wf\text{-}SetComprehension(\pi)(e)(t)(env)$,
.18 $mk\text{-}SetRange(\text{-},\text{-}) \rightarrow wf\text{-}SetRange(\pi)(e)(t)(env)$,
.19 $mk\text{-}SeqEnumeration(\text{-}) \rightarrow wf\text{-}SeqEnumeration(\pi)(e)(t)(env)$,
.20 $mk\text{-}SeqComprehension(\text{-},\text{-},\text{-}) \rightarrow wf\text{-}SeqComprehension(\pi)(e)(t)(env)$,
.21 $mk\text{-}SubSequence(\text{-},\text{-},\text{-}) \rightarrow wf\text{-}SubSequence(\pi)(e)(t)(env)$,
.22 $mk\text{-}MapEnumeration(\text{-}) \rightarrow wf\text{-}MapEnumeration(\pi)(e)(t)(env)$,
.23 $mk\text{-}MapComprehension(\text{-},\text{-},\text{-}) \rightarrow wf\text{-}MapComprehension(\pi)(e)(t)(env)$,
.24 $mk\text{-}TupleConstructor(\text{-}) \rightarrow wf\text{-}TupleConstructor(\pi)(e)(t)(env)$,
.25 $mk\text{-}RecordConstructor(\text{-},\text{-}) \rightarrow wf\text{-}RecordConstructor(\pi)(e)(t)(env)$,
.26 $mk\text{-}RecordModifier(\text{-},\text{-}) \rightarrow wf\text{-}RecordModifier(\pi)(e)(t)(env)$,
.27 $mk\text{-}Apply(\text{-},\text{-}) \rightarrow wf\text{-}Apply(\pi)(e)(t)(env)$,
.28 $mk\text{-}FieldSelect(\text{-},\text{-}) \rightarrow wf\text{-}FieldSelect(\pi)(e)(t)(env)$,
.29 $mk\text{-}FctTypeInst(\text{-},\text{-}) \rightarrow wf\text{-}FctTypeInst(\pi)(e)(t)(env)$,
.30 $mk\text{-}Lambda(\text{-},\text{-}) \rightarrow wf\text{-}Lambda(\pi)(e)(t)(env)$,
.31 $mk\text{-}IsDefTypeExpr(\text{-},\text{-}) \rightarrow wf\text{-}IsDefTypeExpr(\pi)(e)(t)(env)$,
.32 $mk\text{-}IsBasicTypeExpr(\text{-},\text{-}) \rightarrow wf\text{-}IsBasicTypeExpr(\pi)(e)(t)(env)$,
.33 $mk\text{-}Name(\text{-}) \rightarrow wf\text{-}Name(\pi)(e)(t)(env)$,
.34 $mk\text{-}OldName(\text{-}) \rightarrow wf\text{-}OldName(\pi)(e)(t)(env)$,
.35 others $\rightarrow wf\text{-}Literal(\pi)(e)(t)(env)$
.36 end

wf-ExprBasic and the basic well-formedness predicates for the different expression constructs defines the basic type properties of expressions. There is no subsumption rule included in these basic well-formedness predicates, so for many expression constructs they will only be true for at most one type.

The subsumption rule is applied in the next formula where the general expression well-formedness predicate *wf-Expr* is defined. *wf-Expr* is the well-formedness predicate to be used for checking all subcomponent expressions. Depending on the context of the subcomponent expression the predicate may furthermore be explicitly transformed by other rules (*InType*, *InScope*).

183.0 $wf\text{-}Expr : \Pi \xrightarrow{t} WFE$

.1 $wf\text{-}Expr\,(\pi) \triangleq$
.2 $SubsumeExpr(\pi)(wf\text{-}ExprBasic(\pi))$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.4 Bracketed Expression

184.0 $wf\text{-}BracketedExpr : \Pi \xrightarrow{t} BracketedExpr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}BracketedExpr\,(\pi)(mk\text{-}BracketedExpr(e))(type)(env) \triangleq$
.2 $\quad wf\text{-}Expr(\pi)(e)(type)(env)$

14.3.5 Local Binding Expressions

14.3.5.1 Let Expression

185.0 $wf\text{-}LetExpr : \Pi \xrightarrow{t} LetExpr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}LetExpr\,(\pi)(mk\text{-}LetExpr(localdefs, bod \quad ype)(env) \triangleq$
.2 $\quad \exists\, venv : VisibleEnv \cdot$
.3 $\qquad$ let $wfe = InScope[Expr](\pi)(\mathsf{dom}\ venv)(wf\text{-}Expr(\pi))$,
.4 $\qquad\quad lenv = UpdateEnv(venv)(env)$ in
.5 $\qquad wf\text{-}ValFnOpDefs(\pi)(localdefs)(venv)(env) \wedge$
.6 $\qquad wfe(body)(type)(lenv)$

Notice that in the transformation from outer abstract syntax to core abstract syntax the sequence of local definitions (*localdefs*) is transformed to a map. Consequently, if an identifier is defined twice in *localdefs* the two definitions must be compatible so they can be merged; this is checked in *wf-OrderedDefs* via *wf-ValFnOpDefs* (compare with *wf-DefExpr*).

14.3.5.2 Let Be ST Expression

186.0 $wf\text{-}LetBeSTExpr : \Pi \xrightarrow{t} LetBeSTExpr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}LetBeSTExpr\,(\pi)(mk\text{-}LetBeSTExpr(bind, ocond, body))(type)(env) \triangleq$
.2 $\quad (\exists\, venv : ValEnv, bindattrs : BindAttrs \cdot$
.3 $\qquad wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \wedge$
.4 $\qquad$ let $newenv = UpdateEnv(venv)(env)$,
.5 $\qquad\quad wfe = (InScope[Expr](\pi)(\mathsf{dom}\ venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi))$ in
.6 $\qquad (ocond \neq \mathsf{nil} \Rightarrow$
.7 $\qquad\ \exists\, truetype : TypeR \cdot$
.8 $\qquad\quad wfe(ocond)(truetype)(newenv) \wedge IsTrueType(\pi)(truetype)(env)) \wedge$
.9 $\qquad InScope[Expr](\pi)(\mathsf{dom}\ venv)(wf\text{-}Expr(\pi))(body)(type)(newenv)) \wedge$
.10 $\neg \exists\, venv : ValEnv, bindattrs : BindAttrs \cdot$
.11 $\qquad wf\text{-}Bind(NegWfClass(\pi))(bind)(venv, bindattrs)(env) \wedge$
.12 $\qquad \text{EMPTY} \in bindattrs$

For a LetBeST expression to be well-formed several conditions must be satisfied. The binding *bind* must be well-formed (line 186.3) producing some value environment *venv*. The optional expression *ocond* and the body expression *body* must be well-formed in the original environment extended with the value environment originating from the binding (line 186.4). In order to make the sub-environment based checks possible the *In-Scope* rule has been applied to the basic well-formedness predicate *wf-Expr*, this is so both at the check of the expression *ocond* (line 186.5 and 186.8) and at the check of the expression *body* (line 186.9).

For the produced value environment *venv* the condition *ocond* must in the Dynamic Semantics evaluate to true. This is in the Static Semantics checked in *IsTrueType* (line 186.8). When checking an expression for well-formedness with respect to some specific type (here *truetype*) the *InType* rule must be applied (line 186.5).

Finally, it is checked that the set of value environments generated by the binding is non-empty (line 186.10-.12). For definite well-formedness it is checked that one can reject that EMPTY is in *bindattrs*. For possible well-formedness, if

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

one cannot accept that EMPTY is in *bindattrs*, then one cannot reject that the set of generated value environments is non-empty.

```
187.0 IsTrueType : Π -t-> TypeR → Env → 𝔹
   .1 IsTrueType (π)(t)(env) ≙
   .2   cases π :
   .3     mk-DEF(-) → false,
   .4     mk-POS(-) → t = BOOLEAN
   .5   end
```

This predicate holds only if the argument type contains one single value: true. Clearly, this is possibly the case for the type of Booleans and, on the other hand, it is sound never to say that it is definitely the case.

14.3.5.3 Def Expression

```
188.0 wf-DefExpr : Π -t-> DefExpr → TypeR → Env → 𝔹
   .1 wf-DefExpr (π)(mk-DefExpr(defs, body))(type)(env) ≙
   .2   ∃ venv : VisibleEnv ·
   .3       let wfe = InScope[Expr](π)(dom venv)(wf-Expr(π)),
   .4           lenv = UpdateEnv(venv)(env) in
   .5       wf-OrderedDefs(π)(defs, false)(venv)(env) ∧
   .6       wfe(body)(type)(lenv)
```

In the transformation from outer abstract syntax to core abstract syntax the *DefExpr* is treated in the following way. If the *DefExpr* has more than one *DefBind* in the sequence of *DefBinds* (*defs*), the *DefExpr* is transformed to nested *DefExpr*s, each having one *DefBind*. Consequently, for two *DefBind*'s declaring the same identifier there is no requirement about compatibility (the false in line 188.5).

14.3.6 Conditional Expressions

14.3.6.1 If Expression

```
189.0 wf-IfExpr : Π -t-> IfExpr → TypeR → Env → 𝔹
   .1 wf-IfExpr (π)(mk-IfExpr(test, cons, elsifaltns, altn))(type)(env) ≙
   .2   let altn' : Expr =
   .3           cases elsifaltns :
   .4             ([]) → altn,
   .5             [mk-ElsifExpr(test1, cons1)] ⌢ raltns →
   .6                 mk-IfExpr(test1, cons1, raltns, altn)
   .7           end in
   .8   InType[Expr](π)(wf-Expr(π))(test)(BOOLEAN)(env) ∧
   .9   cases π :
   .10    mk-DEF(-) →
   .11        wf-Expr(π)(cons)(type)(env) ∧ wf-Expr(π)(altn')(type)(env),
   .12    mk-POS(-) →
   .13        wf-Expr(π)(cons)(type)(env) ∨ wf-Expr(π)(altn')(type)(env)
   .14  end
```

In order to simplify the well-formedness check the If expression is treated as a simple if-then-else expression: if *test* then *cons* else *altn'*. This is arranged in line 189.3-.7 by transforming the nested sequence of elseif-alternatives and the final alternative to a single alternative expression. With the type representations available, the Static Semantics

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

is not able to make a separate analysis of the two cases arising from the *test*-expression being respectively true and false. Consequently, the type of the If expression covers both possibilities.

For definite well-formedness *both* the consequence (*cons*) *and* the alternative (*altn'*) have to be definitly well-formed with respect to the result *type* of the If expression. This will work as follows. The two expressions *cons* and *altn'* will be definitly well-formed for some smallest types t_1 respectively t_2. Because the subsumption rule is built into the *wf-Expr*–predicate they will also bc definitly well-formed for some result *type* not less than the union type $t_1 \mid t_2$.

For possible well-formedness the If expression is possibly well-formed for all types for which either *cons* or *altn'* is possibly well-formed. Consequently, because of the completeness of the posssible well-formedness check, if the If expression isn't possibly well-formed for some *type* it may safely be rejected that the If expression in the Dynamic Semantics evaluates to a value in that type.

Notice that for possible well-formedness the two result expressions *cons* and *altn'* are *not* checked using *wf-Expr*. Using *wf-Expr* would result in double application of the subsumption rule, and because the subsumption rule is not transitive for possible well-formedness, that would result in weak conclusio

14.3.6.2 Cases Expression

```
190.0 wf-CasesExpr : Π →t CasesExpr → TypeR → Env → 𝔹
  .1 wf-CasesExpr (π)(e)(type)(env) ≙
  .2   let mk-CasesExpr(sel, altns, (nil)) = RemoveCaseOther(e) in
  .3   ∃ selt : TypeR ·
  .4         wf-Expr(π)(sel)(selt)(env) ∧
  .5         cases π :
  .6           mk-DEF(-) →
  .7                (∀ alt ∈ elems altns ·
  .8                      ∀ pat ∈ elems alt.match ·
  .9                           ∀ matcht : TypeR, venv : ValEnv ·
  .10                               (let π' = mk-POS(π.checkset) in
  .11                                wf-Expr(π')(sel)(matcht)(env) ∧
  .12                                wf-Pattern(π')(pat)(mk-TVE(matcht, venv))(env))
  .13                                ⇒
  .14                               let wfe = InScope[Expr](π)(dom venv)(wf-Expr(π)) in
  .15                               wfe(alt.body)(type)(UpdateEnv(venv)(env)))
  .16                ∧
  .17                CanMatch(π)(mk-CasesExpr(sel, altns, nil))(env),
  .18           mk-POS(-) →
  .19                ∃ alt ∈ elems altns ·
  .20                     ∃ pat ∈ elems alt.match ·
  .21                          ∃ venv : ValEnv ·
  .22                               wf-Pattern(π)(pat)(mk-TVE(selt, venv))(env) ∧
  .23                               let wfe = InScope[Expr](π)(dom venv)(wf-Expr(π)) in
  .24                               wfe(alt.body)(type)(UpdateEnv(venv)(env))
  .25         end
```

In order to treat all case alternatives in a uniform way a possible 'other' alternative is removed and placed as the last ordinary alternative (line 190.2 *RemoveCaseOther*). For a case expression to be well-formed the selector expression and the case alternatives must be well-formed. We treat definite and possible well-formedness of the case alternatives seperately.

For definite well-formedness it is required that all alternative expressions which may be evaluated in the Dynamic Semantics are definitly well-formed. The easiest way for the Static Semantics to fulfil this requirement is to require that *all* alternative expressions are definitely well-formed. However, the Static Semantics uses a better approach. If for some alternative it can be proven that it will never be executed in the Dynamic Semantics, then the corresponding

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

expression will not be checked for definite well-formedness. For all patterns in all alternatives (line 190.7-.15), if for some matchtype (*matcht*) and value environment (*venv*) (line 190.9-.15:) the selector expression is *possibly* well-formed with respect to that type and the pattern *possibly* matches a value of that type producing a valueenvironment *venv*, then it is not rejectable that the selector expression may evaluate to a value which matches the pattern. Consequently (line 190.14-.15) the Static Semantics checks that the corresponding alternative expression is *definitly* well-formed with respect to the given case expression *type*. The check is made in an environment extended with the value environment (*venv*) originating from the pattern match. Notice, that in order to make the sub-environment based checks possible the *In-Scope*–rule has been used to the basic well-formedness predicate (*wf-Expr*). Finally for definite well-formedness the Static Semantics guarantees that there always will be a case alternative which will match the selector expression value (line 190.17).

For possible well-formedness: if the selector expression is possibly well-formed for some type (*selt*) (line 190.4) and (190.19-.24) if for some pattern in some alternative (line 190.22-.24) the pattern possibly matches a value of that type and the corresponding alternative expression is possibly well-formed, then it is not rejectable that the case expression may evaluate correctly in the Dynamic Semantics. As for definite well-formedness the alternative expression is checked in the extended environment using the *In-Scope*–rule.

```
191.0 CanMatch : Π -t-> CasesExpr → Env → 𝔹
   .1 CanMatch (π)(mk-CasesExpr(sel, altns, (nil)))(env) ≜
   .2   let Is-matchtypes : TypeR-set → 𝔹
   .3       Is-matchtypes (mtypes) ≜
   .4         ∀ mtype ∈ mtypes ·
   .5               ∃ alt ∈ elems altns ·
   .6                    ∃ pat ∈ elems alt.match, venv : ValEnv ·
   .7                         wf-Pattern(π)(pat)(mk-TVE(mtype, venv))(env)
   .8       in
   .9   ∃ mtypes : TypeR-set, seltype : TypeR ·
  .10       Is-matchtypes(mtypes) ∧
  .11       wf-Expr(π)(sel)(seltype)(env) ∧
  .12       IsSubtypeR(π)(seltype, mk-UnionTypeR(mtypes))(env)
  .13pre  is-DEF(π)
```

Notice, that *CanMatch* is always called with a π for which *is-DEF*(π). *CanMatch* checks that for every value to which the selector expression may evaluate, there definitly exists an alternative with a pattern which matches the value. In line 191.3, *is-matchtypes*(*mtypes*) is true if every type t in *mtypes* has the property that every value in the type definitly will match some pattern of some alternative.

```
192.0 RemoveCaseOther : CasesExpr -t-> CasesExpr
   .1 RemoveCaseOther (mk-CasesExpr(sel, altns, other)) ≜
   .2   cases other :
   .3     (nil) → mk-CasesExpr(sel, altns, other),
   .4     others → let lastalt = mk-CaseAltn([mk-PatternId(nil)], other) in
   .5                 mk-CasesExpr(sel, altns ↷ [lastalt], nil)
   .6   end
```

14.3.7 Unary Expressions

All unary expressions are checked by using characteristic predicates.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

193.0 $wf\text{-}PrefixExpr : \Pi \xrightarrow{t} PrefixExpr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}PrefixExpr\,(\pi)(mk\text{-}PrefixExpr(pref, expr))(type)(env) \triangleq$
.2 cases $pref$:
.3 (NUMPLUS) → $wf\text{-}NUMPLUS(\pi)(expr)(type)(env)$,
.4 (NUMMINUS) → $wf\text{-}NUMMINUS(\pi)(expr)(type)(env)$,
.5 (NUMABS) → $wf\text{-}NUMABS(\pi)(expr)(type)(env)$,
.6 (FLOOR) → $wf\text{-}FLOOR(\pi)(expr)(type)(env)$,
.7 (NOT) → $wf\text{-}NOT(\pi)(expr)(type)(env)$,
.8 (SETCARD) → $wf\text{-}SETCARD(\pi)(expr)(type)(env)$,
.9 (SETPOWER) → $wf\text{-}SETPOWER(\pi)(expr)(type)(env)$,
.10 (SETDISTRUNION) → $wf\text{-}SETDISTRUNION(\pi)(expr)(type)(env)$,
.11 (SETDISTRINTERSECT) → $wf\text{-}SETDISTRINTERSECT(\pi)(expr)(type)(env)$,
.12 (SEQHEAD) → $wf\text{-}SEQHEAD(\pi)(expr)(type)(env)$,
.13 (SEQTAIL) → $wf\text{-}SEQTAIL(\pi)(expr)(type)(env)$,
.14 (SEQLEN) → $wf\text{-}SEQLEN(\pi)(expr)(type)(env)$,
.15 (SEQELEMS) → $wf\text{-}SEQELEMS(\pi)(expr)(type)(env)$,
.16 (SEQINDICES) → $wf\text{-}SEQINDICES(\pi)(expr)(type)(env)$,
.17 (SEQDISTRCONC) → $wf\text{-}SEQDISTRCONC(\pi)(expr)(type)(env)$,
.18 (MAPDOM) → $wf\text{-}MAPDOM(\pi)(expr)(type)(env)$,
.19 (MAPRNG) → $wf\text{-}MAPRNG(\pi)(expr)(type)(env)$,
.20 (MAPDISTRMERGE) → $wf\text{-}MAPDISTRMERGE(\pi)(expr)(type)(env)$
.21 end

14.3.7.1 Numeric Operations

194.0 $wf\text{-}NUMPLUS : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}NUMPLUS\,(\pi)(e)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([e_t])(rt)(env) \triangleq$
.4 $IsSubtypeR(\pi)(e_t, \text{REAL})(env) \wedge rt = e_t$
.5 in
.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

195.0 $wf\text{-}NUMMINUS : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}NUMMINUS\,(\pi)(e)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([e_t])(rt)(env) \triangleq$
.4 $IsSubtypeR(\pi)(e_t, \text{REAL})(env) \wedge$
.5 $rt = mk\text{-}UnionTypeR(\{e_t, \text{INT}\})$
.6 in
.7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

The requirement in line 195.5 says that the result type rt is the greatest of the operand type and INTEGER. In the type system the least type to represent the value of unary minus applied to a natural number is INTEGER.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

196.0 $wf\text{-}NUMABS : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}NUMABS\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $IsSubtypeR(\pi)(e_t, \text{REAL})(env) \wedge$

.5 cases e_t :

.6 $(\text{INT}) \to rt = \text{NAT}$,

.7 others $\to rt = e_t$

.8 end

.9 in

.10 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

197.0 $wf\text{-}FLOOR : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}FLOOR\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $IsIntersectionType(\pi)(\text{INT}, e_t)(rt)(env)$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

14.3.7.2 Logical Operation

198.0 $wf\text{-}NOT : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}NOT\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $e_t = \text{BOOLEAN} \wedge rt = \text{BOOLEAN}$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.7.3 Set Operations

199.0 $wf\text{-}SETCARD : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SETCARD\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(e_t) \wedge rt = \text{NAT}$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

200.0 $wf\text{-}SETPOWER : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SETPOWER\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(e_t) \wedge rt = mk\text{-}SetTypeR(e_t)$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

201.0 $wf\text{-}SETDISTRUNION : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SETDISTRUNION\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $\exists\, t : TypeR \cdot$

.5 $e_t = mk\text{-}SetTypeR(mk\text{-}SetTypeR(t)) \wedge rt = mk\text{-}SetTypeR(t)$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

202.0 $wf\text{-}SETDISTRINTERSECT : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SETDISTRINTERSECT\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(rt) \wedge$

.5 $IsNonEmptySet(\pi)(e_t)(env) \wedge$

.6 $Shape(e_t) = mk\text{-}SetTypeR(rt)$

.7 in

.8 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.7.4 Sequence Operations

203.0 $wf\text{-}SEQHEAD : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SEQHEAD\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $\exists\, t : TypeR \cdot e_t = mk\text{-}Seq1\,TypeR(t) \wedge rt = t$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

204.0 $wf\text{-}SEQTAIL : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SEQTAIL\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $\exists\, t : TypeR \cdot e_t = mk\text{-}Seq1\,TypeR(t) \wedge rt = mk\text{-}Seq0\,TypeR(t)$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

205.0 $wf\text{-}SEQLEN : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SEQLEN\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $is\text{-}Seq0\,TypeR(e_t) \wedge rt = \text{NAT} \vee$

.5 $is\text{-}Seq1\,TypeR(e_t) \wedge rt = \text{NATONE}$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

206.0 $wf\text{-}SEQELEMS : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SEQELEMS\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $\exists\, t : TypeR \cdot$

.5 $IsSeqTypeR(e_t)(t) \wedge rt = mk\text{-}SetTypeR(t)$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

207.0 $wf\text{-}SEQINDICES : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SEQINDICES\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $(\exists\, elemt : TypeR \cdot IsSeqTypeR(e_t)(elemt)) \wedge rt = mk\text{-}SetTypeR(\text{NATONE})$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

208.0 $wf\text{-}SEQDISTRCONC : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SEQDISTRCONC\,(\pi)(e)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([e_t])(rt)(env) \triangleq$

.4 $\exists\, seqt, t : TypeR \cdot$

.5 $IsSeqTypeR(e_t)(seqt) \wedge IsSeqTypeR(seqt)(t) \wedge$

.6 if $is\text{-}Seq1\,TypeR(e_t) \wedge is\text{-}Seq1\,TypeR(seqt)$

.7 then $rt = mk\text{-}Seq1\,TypeR(t)$ else $rt = mk\text{-}Seq0\,TypeR(t)$

.8 in

.9 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.7.5 Map Operations

209.0 $wf\text{-}MAPDOM : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}MAPDOM\,(\pi)(e)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([e_t])(rt)(env) \triangleq$
.4 $\exists\, t_1, t_2 : TypeR \cdot$
.5 $IsMapTypeR(e_t)(t_1, t_2) \wedge rt = mk\text{-}SetTypeR(t_1)$
.6 in
.7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

210.0 $wf\text{-}MAPRNG : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}MAPRNG\,(\pi)(e)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([e_t])(rt)(env) \triangleq$
.4 $\exists\, t_1, t_2 : TypeR \cdot$
.5 $IsMapTypeR(e_t)(t_1, t_2) \wedge rt = mk\text{-}SetTypeR(t_2)$
.6 in
.7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

211.0 $wf\text{-}MAPDISTRMERGE : \Pi \xrightarrow{t} Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}MAPDISTRMERGE\,(\pi)(e)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([e_t])(rt)(env) \triangleq$
.4 $\exists\, mapt, t_1, t_2 : TypeR \cdot$
.5 $e_t = mk\text{-}SetTypeR(mapt) \wedge IsMapTypeR(mapt)(t_1, t_2) \wedge$
.6 $rt = mk\text{-}GeneralMapTypeR(t_1, t_2) \wedge$
.7 $CompatibleMapSet(\pi)(e_t)(env)$
.8 in
.9 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$;

212.0 $CompatibleMapSet : \Pi \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.1 $CompatibleMapSet\,(\pi)(type)(env) \triangleq$
.2 $\exists\, mapt, t_1, t_2 : TypeR \cdot$
.3 $type = mk\text{-}SetTypeR(mapt) \wedge IsMapTypeR(mapt)(t_1, t_2) \wedge$
.4 $(is\text{-}DEF(\pi) \Rightarrow IsOneValueType(\pi)(t_2)(env))$

CompatibleMapSet checks that *type* only contains values which are sets of compatible maps. For two maps to be compatible, overlapping domain values must be bound to the same range value. As all the map-values in the Static Semantics are represented by the same map type *mapt*, in the definite case the Static Semantics can only assure this by requiring that the range type of *mapt* contains a single value.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.7.6 Map Inverse Expression

213.0 $wf\text{-}MapInverseExpr : \Pi \xrightarrow{t} MapInverseExpr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}MapInverseExpr\,(\pi)(mk\text{-}MapInverseExpr(mape))(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([mapt])(rt)(env) \triangleq$

.4 $\exists\, t_1, t_2 : TypeR \cdot$

.5 $mapt = mk\text{-}InjectiveMapTypeR(t_1, t_2) \wedge$

.6 $rt = mk\text{-}InjectiveMapTypeR(t_2, t_1) \wedge$

.7 $IsFlatTypeR(\pi)(t_2)(env)$

.8 in

.9 $wf\text{-}SCompExpr(\pi)(CP)([mape])(type)(env)$

14.3.8 Binary Expressions

The binary expressions – other than some of the logical operations – are checked by using characteristic predicates.

214.0 $wf\text{-}BinaryExpr : \Pi \xrightarrow{t} BinaryExpr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}BinaryExpr\,(\pi)(mk\text{-}BinaryExpr(le, opr, re))(type)(env) \triangleq$

.2 cases opr :

.3 (EQ) $\to wf\text{-}EQ(\pi)(le, re)(type)(env)$,

.4 (NE) $\to wf\text{-}NE(\pi)(le, re)(type)(env)$,

.5 (OR) $\to wf\text{-}OR(\pi)(le, re)(type)(env)$,

.6 (AND) $\to wf\text{-}AND(\pi)(le, re)(type)(env)$,

.7 (IMPLY) $\to wf\text{-}IMPLY(\pi)(le, re)(type)(env)$,

.8 (EQUIV) $\to wf\text{-}EQUIV(\pi)(le, re)(type)(env)$,

.9 (NUMLT) $\to wf\text{-}NUMLT(\pi)(le, re)(type)(env)$,

.10 (NUMLE) $\to wf\text{-}NUMLE(\pi)(le, re)(type)(env)$,

.11 (NUMGT) $\to wf\text{-}NUMGT(\pi)(le, re)(type)(env)$,

.12 (NUMGE) $\to wf\text{-}NUMGE(\pi)(le, re)(type)(env)$,

.13 (INSET) $\to wf\text{-}INSET(\pi)(le, re)(type)(env)$,

.14 (NOTINSET) $\to wf\text{-}NOTINSET(\pi)(le, re)(type)(env)$,

.15 (SUBSET) $\to wf\text{-}SUBSET(\pi)(le, re)(type)(env)$,

.16 (PROPERSUBSET) $\to wf\text{-}PROPERSUBSET(\pi)(le, re)(type)(env)$,

.17 (NUMPLUS) $\to wf\text{-}NUMBINPLUS(\pi)(le, re)(type)(env)$,

.18 (NUMMINUS) $\to wf\text{-}NUMBINMINUS(\pi)(le, re)(type)(env)$,

.19 (NUMMULT) $\to wf\text{-}NUMMULT(\pi)(le, re)(type)(env)$,

.20 (NUMDIV) $\to wf\text{-}NUMDIV(\pi)(le, re)(type)(env)$,

.21 (NUMREM) $\to wf\text{-}NUMREM(\pi)(le, re)(type)(env)$,

.22 (NUMMOD) $\to wf\text{-}NUMMOD(\pi)(le, re)(type)(env)$,

.23 (INTDIV) $\to wf\text{-}INTDIV(\pi)(le, re)(type)(env)$,

.24 (SETUNION) $\to wf\text{-}SETUNION(\pi)(le, re)(type)(env)$,

.25 (SETDIFFERENCE) $\to wf\text{-}SETDIFFERENCE(\pi)(le, re)(type)(env)$,

.26 (SETINTERSECT) $\to wf\text{-}SETINTERSECT(\pi)(le, re)(type)(env)$,

.27 (SEQCONC) $\to wf\text{-}SEQCONC(\pi)(le, re)(type)(env)$,

.28 (MAPMERGE) $\to wf\text{-}MAPMERGE(\pi)(le, re)(type)(env)$,

.29 (MAPORSEQMOD) $\to wf\text{-}MAPORSEQMOD(\pi)(le, re)(type)(env)$,

.30 (MAPDOMRESTRTO) $\to wf\text{-}MAPDOMRESTRTO(\pi)(le, re)(type)(env)$,

.31 (MAPDOMRESTRBY) $\to wf\text{-}MAPDOMRESTRBY(\pi)(le, re)(type)(env)$,

.32 (MAPRNGRESTRTO) $\to wf\text{-}MAPRNGRESTRTO(\pi)(le, re)(type)(env)$,

.33 (MAPRNGRESTRBY) $\to wf\text{-}MAPRNGRESTRBY(\pi)(le, re)(type)(env)$,

.34 (COMPOSE) $\to wf\text{-}COMPOSE(\pi)(le, re)(type)(env)$,

.35 (ITERATE) $\to wf\text{-}ITERATE(\pi)(le, re)(type)(env)$

.36 end

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.1 Numeric Operations

215.0 $wf\text{-}NUMBINPLUS : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}NUMBINPLUS\,(\pi)(le, re)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([le_t, re_t])(rt)(env) \triangleq$
.4 $(\forall\, t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, \text{REAL})(env)) \wedge$
.5 $rt = mk\text{-}UnionTypeR(\{le_t, re_t\})$
.6 in
.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

The requirement in line 215.5 is equivalent to the requirement that the result type rt is the greatest of the two operand types.

216.0 $wf\text{-}NUMBINMINUS : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}NUMBINMINUS\,(\pi)(le, re)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([le_t, re_t])(rt)(env) \triangleq$
.4 $(\forall\, t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, \text{REAL})(env)) \wedge$
.5 $rt = mk\text{-}UnionTypeR(\{le_t, re_t, \text{INT}\})$
.6 in
.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

In line 216.5, the result type rt is the greatest of the two operand types and INTEGER.

217.0 $wf\text{-}NUMMULT : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}NUMMULT\,(\pi)(le, re)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([le_t, re_t])(rt)(env) \triangleq$
.4 $(\forall\, t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, \text{REAL})(env)) \wedge$
.5 $rt = mk\text{-}UnionTypeR(\{le_t, re_t\})$
.6 in
.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

The requirement in line 217.5 says that the result type is the greatest of the two operand types.

218.0 $wf\text{-}NUMDIV : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}NUMDIV\,(\pi)(le, re)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([le_t, re_t])(rt)(env) \triangleq$
.4 $(\forall\, t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, \text{REAL})(env)) \wedge$
.5 $rt = mk\text{-}UnionTypeR(\{le_t, re_t, \text{RAT}\}) \wedge$
.6 $(is\text{-}DEF(\pi) \Rightarrow IsSubtypeR(\pi)(re_t, \text{NATONE})(env))$
.7 in
.8 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

The result type of a numeric division is the greatest of the two operand types and the type for rational numbers RAT (line 218.5).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

219.0 $wf\text{-}INTDIV : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}INTDIV\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $rt = \text{INT} \wedge$

.5 $le_t = \text{INT} \wedge$

.6 $re_t =$ if $is\text{-}POS(\pi)$ then INT else NATONE

.7 in

.8 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Notice that the result type doesn't depend on the argument types, so the argument types may be lifted to the required types without overestimating the result type.

220.0 $wf\text{-}NUMREM : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}NUMREM\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $(\forall\, t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, \text{INT})(env)) \wedge$

.5 $rt = mk\text{-}UnionTypeR(\{le_t, re_t, \text{NAT}\}) \wedge$

.6 $(is\text{-}DEF(\pi) \Rightarrow IsSubtypeR(\pi)(re_t, \text{NATONE})(env))$

.7 in

.8 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

221.0 $wf\text{-}NUMMOD : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}NUMMOD\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $le_t = \text{NAT} \wedge re_t = \text{NATONE} \wedge$

.5 $rt = \text{NAT}$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.2 Numeric Comparison Operators

```
222.0 wf-NUMLT : Π ─t→ Expr × Expr → TypeR → Env → 𝔹
  .1  wf-NUMLT (π)(le, re)(type)(env) ≙
  .2    let CP : TypeR* ─t→ TypeR → Env → 𝔹
  .3        CP ([le_t, re_t])(rt)(env) ≙
  .4          (∀ t ∈ {le_t, re_t} · t = REAL) ∧
  .5          rt = BOOLEAN
  .6        in
  .7    wf-SCompExpr(π)(CP)([le, re])(type)(env);

223.0 wf-NUMLE : Π ─t→ Expr × Expr → TypeR → Env → 𝔹
  .1  wf-NUMLE (π)(le, re)(type)(env) ≙
  .2    let CP : TypeR* ─t→ TypeR → Env → 𝔹
  .3        CP ([le_t, re_t])(rt)(env) ≙
  .4          (∀ t ∈ {le_t, re_t} · t = REAL) ∧
  .5          rt = BOOLEAN
  .6        in
  .7    wf-SCompExpr(π)(CP)([le, re])(type)(env);

224.0 wf-NUMGT : Π ─t→ Expr × Expr → TypeR → Env → 𝔹
  .1  wf-NUMGT (π)(le, re)(type)(env) ≙
  .2    let CP : TypeR* ─t→ TypeR → Env → 𝔹
  .3        CP ([le_t, re_t])(rt)(env) ≙
  .4          (∀ t ∈ {le_t, re_t} · t = REAL) ∧
  .5          rt = BOOLEAN
  .6        in
  .7    wf-SCompExpr(π)(CP)([le, re])(type)(env);

225.0 wf-NUMGE : Π ─t→ Expr × Expr → TypeR → Env → 𝔹
  .1  wf-NUMGE (π)(le, re)(type)(env) ≙
  .2    let CP : TypeR* ─t→ TypeR → Env → 𝔹
  .3        CP ([le_t, re_t])(rt)(env) ≙
  .4          (∀ t ∈ {le_t, re_t} · t = REAL) ∧
  .5          rt = BOOLEAN
  .6        in
  .7    wf-SCompExpr(π)(CP)([le, re])(type)(env)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.3 Equality operations

226.0 $wf\text{-}EQ : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
.1 $wf\text{-}EQ\,(\pi)(le, re)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
.3 $CP\,([le_t, re_t])(rt)(env) \triangleq$
.4 $rt = \text{BOOLEAN}$
.5 in
.6 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$;

227.0 $wf\text{-}NE : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
.1 $wf\text{-}NE\,(\pi)(le, re)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
.3 $CP\,([le_t, re_t])(rt)(env) \triangleq$
.4 $rt = \text{BOOLEAN}$
.5 in
.6 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

14.3.8.4 Logical Operations

228.0 $wf\text{-}OR : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
.1 $wf\text{-}OR\,(\pi)(le, re)(type)(env) \triangleq$
.2 $wf\text{-}NonStrictLOGOPRT(\pi)(le, re)(type)(env)$;

229.0 $wf\text{-}AND : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
.1 $wf\text{-}AND\,(\pi)(le, re)(type)(env) \triangleq$
.2 $wf\text{-}NonStrictLOGOPRT(\pi)(le, re)(type)(env)$;

230.0 $wf\text{-}IMPLY : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
.1 $wf\text{-}IMPLY\,(\pi)(le, re)(type)(env) \triangleq$
.2 $wf\text{-}NonStrictLOGOPRT(\pi)(le, re)(type)(env)$

The OR, AND and IMPLY operations are nonstrict, i.e. depending on the value of one operand the other is allowed to evaluate to $\perp$. However, with the granularity of the given type system the Static Semantics cannot distinguish between the values true and false. Consequently for definite well-formedness (line 231.5-.6) it is required that *both* operands are definite well-formed with respect to the type BOOLEAN. Notice, because the well-formedness requirement is with respect to a specific type, the *InType* rule is applied.

For possible well-formedness (line 231.8-.12), one of the operands must be possibly well-formed with respect to the type BOOLEAN (line 231.9), i.e. it is not rejectable that the operand may evaluate to true or false. Depending on the actual value the *other* operand is allowed to evaluate to $\perp$, but if it does *not* evaluate to $\perp$ then the operand value must be true or false (this requirement is approximated in line 231.10-.12).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

231.0 $wf\text{-}NonStrictLOGOPRT : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}NonStrictLOGOPRT\,(\pi)(le, re)(type)(env) \triangleq$
.2 (let $\pi_D = mk\text{-}DEF(\pi.checkset)$ in
.3 cases π :
.4 $mk\text{-}DEF(\text{-}) \to$
.5 $\forall\, e \in \{le, re\} \cdot$
.6 $InType[Expr](\pi)(wf\text{-}Expr(\pi))(e)(\text{BOOLEAN})(env)$,
.7 $mk\text{-}POS(\text{-}) \to$
.8 $\exists\, mk\text{-}(e_1, e_2) \in \{mk\text{-}(le, re), mk\text{-}(re, le)\}$
.9 $InType[Expr](\pi)(wf\text{-}Expr(\pi))(e_1)(\text{BOOLEAN})(env) \wedge$
.10 $\forall\, t : TypeR \cdot$
.11 $wf\text{-}Expr(\pi_D)(e_2)(t)(env) \Rightarrow$
.12 $IsOverlapping(\pi)(t, \text{BOOLEAN})(env)$
.13 end)
.14 $\wedge$
.15 $type = \text{BOOLEAN}$;

232.0 $wf\text{-}EQUIV : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}EQUIV\,(\pi)(le, re)(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([le_t, re_t])(rt)(env) \triangleq$
.4 $le_t = \text{BOOLEAN} \wedge re_t = \text{BOOLEAN} \wedge rt = \text{BOOLEAN}$
.5 in
.6 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.5 Logical Set Operations

233.0 $wf\text{-}INSET : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}INSET\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(re_t) \wedge rt = \text{BOOLEAN}$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

234.0 $wf\text{-}NOTINSET : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}NOTINSET\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(re_t) \wedge rt = \text{BOOLEAN}$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

235.0 $wf\text{-}SUBSET : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}SUBSET\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(le_t) \wedge is\text{-}SetTypeR(re_t) \wedge$

.5 $rt = \text{BOOLEAN}$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

236.0 $wf\text{-}PROPERSUBSET : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{I}$

.1 $wf\text{-}PROPERSUBSET\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(le_t) \wedge is\text{-}SetTypeR(re_t) \wedge$

.5 $rt = \text{BOOLEAN}$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.6 Set Operations

237.0 $wf\text{-}SETUNION : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SETUNION\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $\exists\, et_1, et_2 : TypeR \cdot$

.5 $le_t = mk\text{-}SetTypeR(et_1) \wedge re_t = mk\text{-}SetTypeR(et_2) \wedge$

.6 $rt = mk\text{-}SetTypeR(mk\text{-}UnionTypeR(\{et_1, et_2\}))$

.7 in

.8 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

238.0 $wf\text{-}SETDIFFERENCE : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SETDIFFERENCE\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(le_t) \wedge is\text{-}SetTypeR(re_t) \wedge$

.5 $rt = le_t$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

239.0 $wf\text{-}SETINTERSECT : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SETINTERSECT\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $is\text{-}SetTypeR(le_t) \wedge is\text{-}SetTypeR(re_t) \wedge$

.5 $IsIntersectionType(\pi)(le_t, re_t)(rt)(env)$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

14.3.8.7 Sequence Operation

240.0 $wf\text{-}SEQCONC : \Pi \xrightarrow{t} Expr \times Expr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}SEQCONC\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $\exists\, et_1, et_2 : TypeR \cdot$

.5 $IsSeqTypeR(le_t)(et_1) \wedge IsSeqTypeR(re_t)(et_2) \wedge$

.6 let $newelemtp = mk\text{-}UnionTypeR(\{et_1, et_2\})$ in

.7 if $is\text{-}Seq1\,TypeR(le_t) \vee is\text{-}Seq1\,TypeR(re_t)$

.8 then $rt = mk\text{-}Seq1\,TypeR(newelemtp)$

.9 else $rt = mk\text{-}Seq0\,TypeR(newelemtp)$

.10 in

.11 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

--`,,```,,,,````-`-`,,`,,`,`,,`---
Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.8 Sequence and Map Modification Operation

```
241.0 wf-MAPORSEQMOD : Π →t Expr × Expr → TypeR → Env → 𝔹
   .1 wf-MAPORSEQMOD (π)(le, re)(type)(env) ≙
   .2   let CP : TypeR* →t TypeR → Env → 𝔹
   .3       CP ([le_t, re_t])(rt)(env) ≙
   .4         ∃ dt_2, rt_2 : TypeR ·
   .5               IsMapTypeR(re_t)(dt_2, rt_2) ∧
   .6               ((∃ dt_1, rt_1 : TypeR ·
   .7                     IsMapTypeR(le_t)(dt_1, rt_1) ∧
   .8                     let newdt = mk-UnionTypeR({dt_1, dt_2}),
   .9                         newrt = mk-UnionTypeR({rt_1, rt_2}) in
   .10                    rt = mk-GeneralMapTypeR(newdt, newrt))
   .11                ∨
   .12                cases le_t :
   .13                  mk-Seq0TypeR(et) → rt = mk-Seq0TypeR(mk-UnionTypeR({et, rt_2})),
   .14                  mk-Seq1TypeR(et) → rt = mk-Seq1TypeR(mk-UnionTypeR({et, rt_2}))
   .15                end
   .16                ∧
   .17                IsSubtypeR(π)(dt_2, NATONE)(env) ∧
   .18                MapDomInInds(π)(dt_2, le_t)(env))
   .19     in
   .20   wf-SCompExpr(π)(CP)([le, re])(type)(env)
```

Note that this operation can be used both as a map override and as a sequence modification operation. In both cases the right operand must be a map (line 241.5).

If the operator is used as a mapoverride operator (line 241.6-.10) the left operand must be a map, and the resulting map type (rt) has a domain type, which is the union of the two map operands' domain types, and it has a range type, which is the union of t two map operands' range type.

If the operator is used as a sequence modification operator (line 241.12-.18) the left operand must be a sequence and the domain elements of the modification map (the right operand) must be natural numbers. In the Dynamic Semantics the domain of the actual modification map must actually be in the set of indices of the sequence to be modified. In the Static Semantics, this is checked in *MapDomInInds* (in linc .242.2).

```
242.0 MapDomInInds : Π →t TypeR × TypeR → Env → 𝔹
   .1 MapDomInInds (π)(mapdomtype, sequencetype)(env) ≙
   .2   is-POS(π)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.9 Map Operations

```
243.0 wf-MAPMERGE : Π →t Expr × Expr → TypeR → Env → 𝔹
  .1  wf-MAPMERGE (π)(le, re)(type)(env) ≙
  .2    let CP : TypeR* →t TypeR → Env → 𝔹
  .3        CP ([le_t, re_t])(rt)(env) ≙
  .4          ∃ at_1, rt_1, at_2, rt_2 : TypeR ·
  .5                IsMapTypeR(le_t)(at_1, rt_1) ∧ IsMapTypeR(re_t)(at_2, rt_2) ∧
  .6                Compatible2Maps(π)(le_t, re_t)(env)
  .7                ∧
  .8                let newat = mk-UnionTypeR({at_1, at_2}),
  .9                    newrt = mk-UnionTypeR({rt_1, rt_2}) in
  .10               rt = mk-GeneralMapTypeR(newat, newrt)
  .11       in
  .12   wf-SCompExpr(π)(CP)([le, re])(type)(env);

244.0 Compatible2Maps : Π → TypeR × TypeR → Env → 𝔹
  .1  Compatible2Maps (π)(mt_1, mt_2)(env) ≙
  .2    ∃ dt_1, rt_1, dt_2, rt_2 : TypeR ·
  .3        IsMapTypeR(mt_1)(dt_1, rt_1) ∧ IsMapTypeR(mt_2)(dt_2, rt_2) ∧
  .4        (SameValue(π)(rt_1, rt_2)(env) ∨ IsDisjoint(π)(dt_1, dt_2)(env))
```

Compatible2Maps checks that the two types mt_1 and mt_2 represent compatible maps. Two maps are compatible if any common element of their domains is mapped to the same value by the two maps. With the granularity of the type system the Static Semantics must approximate this by requiring *either* that the two map ranges represent exactly the same value *or* that the two domains are disjoint.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

245.0 $wf\text{-}MAPDOMRESTRTO : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}MAPDOMRESTRTO\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $(\exists\, mat, mrt : TypeR \cdot$

.5 $IsMapTypeR(le_t)(mat, mrt) \wedge is\text{-}SetTypeR(re_t)) \wedge rt = le_t$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

246.0 $wf\text{-}MAPDOMRESTRBY : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}MAPDOMRESTRBY\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $(\exists\, mat, mrt : TypeR \cdot$

.5 $IsMapTypeR(le_t)(mat, mrt) \wedge is\text{-}SetTypeR(re_t)) \wedge rt = le_t$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

247.0 $wf\text{-}MAPRNGRESTRTO : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}MAPRNGRESTRTO\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $(\exists\, mat, mrt : TypeR \cdot$

.5 $IsMapTypeR(le_t)(mat, mrt) \wedge is\text{-}SetTypeR(re_t)) \wedge rt = le_t$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

248.0 $wf\text{-}MAPRNGRESTRBY : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}MAPRNGRESTRBY\ (\pi)(le, re)(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

.3 $CP\ ([le_t, re_t])(rt)(env) \triangleq$

.4 $(\exists\, mat, mrt : TypeR \cdot$

.5 $IsMapTypeR(le_t)(mat, mrt) \wedge is\text{-}SetTypeR(re_t)) \wedge rt = le_t$

.6 in

.7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.8.10 Compose and Iterate

```
249.0 wf-COMPOSE : Π -t-> Expr × Expr → TypeR → Env → 𝔹
  .1 wf-COMPOSE (π)(le, re)(type)(env) ≜
  .2   let CP : TypeR* -t-> TypeR → Env → 𝔹
  .3       CP ([le_t, re_t])(rt)(env) ≜
  .4         ∃ at_1, rt_1, at_2, rt_2 : TypeR ·
  .5               IsSubtypeR(π)(rt_2, at_1)(env)
  .6               ∧
  .7               (IsMapTypeR(le_t)(at_1, rt_1) ∧ IsMapTypeR(re_t)(at_2, rt_2) ∧
  .8                if is-InjectiveMapTypeR(le_t) ∧ is-InjectiveMapTypeR(re_t)
  .9                then rt = mk-InjectiveMapTypeR(at_2, rt_1)
  .10               else rt = mk-GeneralMapTypeR(at_2, rt_1)
  .11                ∨
  .12               IsFunctionTypeR(le_t)(at_1, rt_1) ∧ IsFunctionTypeR(re_t)(at_2, rt_2) ∧
  .13               if is-TotalFnTypeR(le_t) ∧ is-TotalFnTypeR(re_t)
  .14               then rt = mk-TotalFnTypeR(at_2, rt_1)
  .15               else rt = mk-PartialFnTypeR(at_2, rt_1))
  .16     in
  .17   wf-SCompExpr(π)(CP)([le, re])(type)(env)
```

Note that composition is an overloaded operator. A composition of two functions is a function, similarly , a composition of two mappings is a mapping. Two total functions compose to a total function, otherwise a composition of functions is a partial function. Two injective mappings compose to an injective one, otherwise their composition is a general mapping .

```
250.0 wf-ITERATE : Π -t-> Expr × Expr → TypeR → Env → 𝔹
  .1 wf-ITERATE (π)(le, re)(type)(env) ≜
  .2   let CP : TypeR* -t-> TypeR → Env → 𝔹
  .3       CP ([le_t, re_t])(rt)(env) ≜
  .4         let π' = NegWfClass(π) in
  .5         (∃ t_1, t_2 : TypeR ·
  .6               IsFunctionTypeR(le_t)(t_1, t_2) ∧
  .7               IsSubtypeR(π)(re_t, NAT)(env) ∧
  .8               (IsGTE2(π')(re_t)(env) ⇒ IsSubtypeR(π)(t_2, t_1)(env)) ∧
  .9               if IsZero(π')(re_t)(env)
  .10              then rt = μ (le_t, fnrng ↦ t_1)
  .11              else rt = le_t)
  .12         ∨
  .13         (∃ t_1, t_2 : TypeR ·
  .14               IsMapTypeR(le_t)(t_1, t_2) ∧
  .15               IsSubtypeR(π)(re_t, NAT)(env) ∧
  .16               (IsGTE2(π')(re_t)(env) ⇒ IsSubtypeR(π)(t_2, t_1)(env)) ∧
  .17               if IsZero(π')(re_t)(env)
  .18               then rt = μ (le_t, maprng ↦ t_1)
  .19               else rt = le_t)
  .20         ∨
  .21         cases π :
  .22           mk-DEF(-) → le_t = NATONE ∧ re_t = REAL ∧ rt = REAL,
  .23           mk-POS(-) → le_t = REAL ∧ re_t = REAL ∧ rt = REAL
  .24         end
  .25     in
  .26   wf-SCompExpr(π)(CP)([le, re])(type)(env)
```

Note that iteration is an overloaded operator. It can iterate functions (line 250.5-.11), mappings (line 250.13-.19) and

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

raise numbers to power (line 250.21-.24). If a function or map is iterated one or more times, it retains the type of the first argument. If a function or map is iterated zero times the result is the identity function or map (line 250.10 and 250.18). If a function or a map is iterated two ore more times, it is required that the range type is contained in the domain type (line 250.8 and 250.16).

When the ITERATE operator is used as a numeric exponentiation operator both operands *lv* and *rv* are numeric values. In the Dynamic Semantics there are 3 cases to consider: if $lv = 0$ then $rv > 0$ and the result will always be zero; if $lv < 0$ then the right operand is a rational number with odd denominator, and finally if $lv > 0$ then the result is always greater than 0. The Static Semantics makes a very simple analysis of these situations. Definite well-formedness can only be assured for operand types describing a subset of values belonging to the third case.

251.0 $IsGTE2 : \Pi \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.1 $IsGTE2\,(\pi)(type)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi) \wedge type = \text{NATONE}$

IsGTE2 checks that *type* represents natural numbers greater than or equal to 2.

252.0 $IsZero : \Pi \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.1 $IsZero\,(\pi)(type)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi) \wedge type = \text{NAT}$

14.3.9 Quantified Expressions

253.0 $wf\text{-}AllExpr : \Pi \xrightarrow{t} AllExpr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}AllExpr\,(\pi)(mk\text{-}AllExpr(bindl, pred))(type)(env) \triangleq$
.2 $\quad type = \text{BOOLEAN} \wedge$
.3 $\quad \exists\, venv : ValEnv, bindattrs : BindAttrs \cdot$
.4 $\qquad wf\text{-}BindList(\pi)(bindl)(venv, bindattrs)(env) \wedge$
.5 $\qquad (\text{EMPTY} \in bindattrs \vee$
.6 $\qquad\ \text{let } wfe = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
.7 $\qquad\ wfe(pred)(\text{BOOLEAN})(UpdateEnv(venv)(env)))$;

254.0 $wf\text{-}ExistsExpr : \Pi \xrightarrow{t} ExistsExpr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}ExistsExpr\,(\pi)(mk\text{-}ExistsExpr(bindl, pred))(type)(env) \triangleq$
.2 $\quad type = \text{BOOLEAN} \wedge$
.3 $\quad \exists\, venv : ValEnv, bindattrs : BindAttrs \cdot$
.4 $\qquad wf\text{-}BindList(\pi)(bindl)(venv, bindattrs)(env) \wedge$
.5 $\qquad (\text{EMPTY} \in bindattrs \vee$
.6 $\qquad\ \text{let } wfe = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
.7 $\qquad\ wfe(pred)(\text{BOOLEAN})(UpdateEnv(venv)(env)))$;

255.0 $wf\text{-}ExistsUniqueExpr : \Pi \xrightarrow{t} ExistsUniqueExpr \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}ExistsUniqueExpr\,(\pi)(mk\text{-}ExistsUniqueExpr(bind, pred))(type)(env) \triangleq$
.2 $\quad type = \text{BOOLEAN} \wedge$
.3 $\quad \exists\, venv : ValEnv, bindattrs : BindAttrs \cdot$
.4 $\qquad wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \wedge$
.5 $\qquad (\text{EMPTY} \in bindattrs \vee$
.6 $\qquad\ \text{let } wfe = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
.7 $\qquad\ wfe(pred)(\text{BOOLEAN})(UpdateEnv(venv)(env)))$

All three quantified expressions are checked in the same way (formulae 253, 254, 255). The result *type* is BOOLEAN(line 255.2). The binding or list of bindings must be well-formed with respect to some value environment *venv* and some binding attributes *bindattrs* (line 255.4).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If the set of bindings is empty there are no requirements about well-formedness of the predicate (line 255.5). Otherwise the predicate *pred* must be well-formed in the original environment extended with the value environment (line 255.7). The *In-Scope* rule has been applied to the basic well-formedness predicate *wf-Expr* in order to make the subenvironment checks possible. Because the predicate is checked with respect to the specific type BOOLEAN the *InType* rule is applied.

14.3.10 Iota Expression

256.0 $wf\text{-}IotaExpr : \Pi \xrightarrow{t} IotaExpr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}IotaExpr\,(\pi)(mk\text{-}IotaExpr(bind, pred))(type)(env) \triangleq$
.2 $\quad \exists\, venv : ValEnv, n : Name, bindattrs : BindAttrs \cdot$
.3 $\qquad wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \wedge$
.4 $\qquad \mathsf{let}\ wfe = (InScope[Expr](\pi)(\mathsf{dom}\ venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi))\ \mathsf{in}$
.5 $\qquad wfe(pred)(\text{BOOLEAN})(UpdateEnv(venv)(env)) \wedge$
.6 $\qquad UniqueVal(\pi)(mk\text{-}IotaExpr(bind, pred))(type)(env) \wedge$
.7 $\qquad \mathsf{dom}\ venv = \{n\} \wedge$
.8 $\qquad type = venv(n).type$

Unlike the quantified expressions (formulae 253, 254, 255) the iota expression is not well-formed if the set of bindings is empty, so the *bindattrs* parameter is not used. Besides checking the binding and predicate two extra checks are added: the iota expressions predicate must be true in the Dynamic Semantics for exactly one dynamic binding. This is in the Static Semantics checked in *UniqueVal* (line 256.6). With the type system available there is no way to check this in the Static Semantics (line 257.2). For the iota expression to be well-formed, it is furthermore required that the binding introduces exactly one identifier (line 256.7).

257.0 $UniqueVal : \Pi \xrightarrow{t} IotaExpr \to Type \to Env \to \mathbb{B}$

.1 $UniqueVal\,(\pi)(mk\text{-}IotaExpr(bind, pred))(type)(env) \triangleq$
.2 $\quad is\text{-}POS(\pi)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.11 Set Expressions

```
258.0 wf-SetEnumeration : Π →t SetEnumeration → TypeR → Env → 𝔹
  .1  wf-SetEnumeration (π)(mk-SetEnumeration(exprl))(type)(env) ≙
  .2    let CP : TypeR* →t TypeR → Env → 𝔹
  .3        CP (ts)(rt)(env) ≙
  .4          (∀ t ∈ elems ts · IsFlatType(π)(t)(env)) ∧
  .5          rt = if ts = [] then EMPTYSET else mk-SetTypeR(mk-UnionTypeR(elems ts))
  .6        in
  .7    wf-SCompExpr(π)(CP)(exprl)(type)(env);

259.0 wf-SetComprehension : Π →t SetComprehension → TypeR → Env → 𝔹
  .1  wf-SetComprehension (π)(mk-SetComprehension(elem, bindl, opred))(type)(env) ≙
  .2    ∃ venv : ValEnv, bindattrs, interprattrs : BindAttrs, mk-UnionTypeR(etset) : TypeR ·
  .3          wf-BindList(π)(bindl)(venv, bindattrs)(env)
  .4          ∧
  .5          let newenv : Env = UpdateEnv(venv)(env),
  .6              wfe = InScope[Expr](π)(dom venv)(wf-Expr(π)) in
  .7          EMPTY ∈ bindattrs ∧ type = EMPTYSET
  .8          ∨
  .9          wf-PredInScope(π)(opred)(venv, interprattrs)(env) ∧
  .10         (EMPTY ∈ interprattrs ∧ type = EMPTYSET
  .11          ∨
  .12          (FINITE ∈ bindattrs ∨ FINITE ∈ interprattrs) ∧
  .13          type = mk-SetTypeR(mk-UnionTypeR(etset)) ∧
  .14          ∀ elemt ∈ etset ·
  .15              wfe(elem)(elemt)(newenv) ∧
  .16              IsFlatTypeR(π)(elemt)(env))
```

The well-formedness of set comprehension is not expressed by a characteristic predicate. The list of bindings must be well-formed with respect to some value environment *venv* and binding attributes *bindattrs* (line 259.3).

If the bindings generate the empty set of dynamic bindings (line 259.7) then the result is the empty set and there are no requirements about well-formedness of the predicate and element expression. Otherwise the optional predicate *opred* must – if it exists – be well-formed in the original environment extended with the value environment *venv* (line 259.9). *wf-PredInScope* applies the *InScope* rule to the basic well-formedness predicate *wf-Expr* in order to make the subenvironment based checks possible. This predicate also provides information about the number of bindings which make the predicate true (via the *interprattrs*).

If the set of bindings which make the predicate true is empty (line 259.10) then the result is the empty set and there is no requirement about well-formedness of the element expression. Otherwise the set of bindings which make the predicate true must be finite (line 259.12), and the element expression must be well-formed in the extended environment (line 259.15). The element expression *elem* in the set comprehension is in the Dynamic Semantics evaluated for all the bindings which make the predicate true thereby yielding many element values. In the Static Semantics, if the element expression is well-formed for the types t_1 and t_2 then the set comprehension is (among others) well-formed for the types t_1-set, t_2-set *and* (t_1 | t_2)-set. The union of the element types has been introduced explicitly (in line 259.13). This is only necessary for possible well-formedness whereas for definite well-formedness the explicit introduction of the union type adds nothing. For all the member types of the union type the element expression must be well-formed (259.15) and the type must be flat (259.16).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

260.0 $wf\text{-}SetRange : \Pi \xrightarrow{t} SetRange \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}SetRange\,(\pi)(mk\text{-}SetRange(lb, ub))(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([lbt, ubt])(rt)(env) \triangleq$
.4 $(\forall\, bt \in \{lbt, ubt\} \cdot IsSubtypeR(\pi)(bt, \text{REAL})(env)) \wedge$
.5 let $et =$ if $IsSubtypeR(\pi)(lbt, \text{INT})(env)$ then lbt else INT in
.6 $rt = mk\text{-}SetTypeR(et)$
.7 in
.8 $wf\text{-}SCompExpr(\pi)(CP)([lb, ub])(type)(env)$

14.3.12 Sequence Expressions

261.0 $wf\text{-}SeqEnumeration : \Pi \xrightarrow{t} SeqEnumeration \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}SeqEnumeration\,(\pi)(mk\text{-}SeqEnumeration(exprl))(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,(ts)(rt)(env) \triangleq$
.4 $rt =$ if $ts = []$
.5 then EMPTYSEQ
.6 else $mk\text{-}Seq1TypeR(mk\text{-}UnionTypeR(\text{elems } ts))$
.7 in
.8 $wf\text{-}SCompExpr(\pi)(CP)(exprl)(type)(env)$;

262.0 $wf\text{-}SeqComprehension : \Pi \xrightarrow{t} SeqComprehension \to TypeR \to Env \to \mathbb{B}$
.1 $wf\text{-}SeqComprehension\,(\pi)(mk\text{-}SeqComprehension(elem, setbind, opred))(type)(env) \triangleq$
.2 $\exists\, venv : ValEnv, bindattrs, interprattrs : BindAttrs, mk\text{-}UnionTypeR(etset) : TypeR \cdot$
.3 $wf\text{-}SetBind(\pi)(setbind)(venv, bindattrs)(env) \wedge$
.4 $\exists\, n : Name \cdot \text{dom } venv = \{n\} \wedge IsSubtypeR(\pi)(venv(n).type, \text{REAL})(env)$
.5 $\wedge$
.6 let $newenv : Env = UpdateEnv(venv)(env)$,
.7 $wfe = InScope[Expr](\pi)(\text{dom } venv)(wf\text{-}Expr(\pi))$ in
.8 $\text{EMPTY} \in bindattrs \wedge type = \text{EMPTYSEQ}$
.9 $\vee$
.10 $wf\text{-}PredInScope(\pi)(opred)(venv, interprattrs)(env) \wedge$
.11 $(\text{EMPTY} \in interprattrs \wedge type = \text{EMPTYSEQ}$
.12 $\vee$
.13 $(\text{FINITE} \in bindattrs \vee \text{FINITE} \in interprattrs) \wedge$
.14 $type = mk\text{-}Seq0TypeR(mk\text{-}UnionTypeR(etset)) \wedge$
.15 $\forall\, elemt \in etset \cdot wfe(elem)(elemt)(newenv))$

The well-formedness predicate for sequence comprehension is structured in the same way as the well-formedness predicate for set comprehension (259) so in the following only some of the differences are mentioned. The result is a sequence type (line 262.14). In the sequence comprehension the binding is a set binding, which must be well-formed with respect to some value environment *venv* (line 262.3). The value environment should introduce exactly one identifier, which should be bound to a numeric type (line 262.4).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
263.0 wf-SubSequence : Π →t SubSequence → TypeR → Env → 𝔹
  .1 wf-SubSequence (π)(mk-SubSequence(seqe, frome, toe))(type)(env) ≙
  .2   let CP : TypeR* →t TypeR → Env → 𝔹
  .3       CP ([seqt, fromt, tot])(rt)(env) ≙
  .4         (∀ t ∈ {fromt, tot} · t = REAL) ∧
  .5         ∃ et : TypeR · IsSeqTypeR(seqt)(et) ∧
  .6         rt = mk-Seq0TypeR(seqt.elemtp)
  .7       in
  .8   wf-SCompExpr(π)(CP)([seqe, frome, toe])(type)(env)
```

14.3.13 MapExpressions

```
264.0 wf-MapEnumeration : Π →t MapEnumeration → TypeR → Env → 𝔹
  .1 wf-MapEnumeration (π)(mk-MapEnumeration(maplets))(type)(env) ≙
  .2   let CP : TypeR* →t TypeR → Env → 𝔹
  .3       CP (pts)(rt)(env) ≙
  .4         let mdts = {let mk-ProductTypeR([dt, -]) = pts(i) in dt | i ∈ inds pts},
  .5             mrts = {let mk-ProductTypeR([-, rt]) = pts(i) in rt | i ∈ inds pts} in
  .6         (∀ dt ∈ mdts · IsFlatTypeR(π)(dt)(env)) ∧
  .7         ∀ i1, i2 ∈ inds pts ·
  .8               let mk-ProductTypeR([dt1, rt1]) = pts(i1),
  .9                   mk-ProductTypeR([dt2, rt2]) = pts(i2) in
  .10              i1 ≠ i2 ∧
  .11              SameValue(NegWfClass(π))(dt1, dt2)(env) ⇒ SameValue(π)(rt1, rt2)(env)
  .12        ∧
  .13        rt =
  .14        if pts = [] then EMPTYMAP
  .15        else mk-GeneralMapTypeR(mk-UnionTypeR(mdts), mk-UnionTypeR(mrts))
  .16      in
  .17  let tupleseq : TupleConstructor* =
  .18          [let mk-Maplet(de, re) = maplets(i) in
  .19           mk-TupleConstructor([de, re]) | i ∈ inds maplets] in
  .20  wf-SCompExpr(π)(CP)(tupleseq)(type)(env)
```

In order to be able to state the type properties of the Map Enumeration by a characteristic predicate and hence use the general well-formedness predicate *wf-SCompExpr* for strict compound expressions, a transformation is applied to the Map Enumeration: the sequence of maplets (which are not expressions) is transformed to a sequence of two-element tuples (line 264.18-.19). The so obtained sequence of *expressions* is now a legal argument for *wf-SCompExpr*. The characteristic predicate *CP* must state the type properties of this transformed maplets, so the first argument *pts* of *CP* is now a sequence of two-element product types, where the first component type is the type of a domain expression and the second component type is the corresponding range expression type. The domain expressions must not evaluate to values containing functions, i.e. the corresponding domain expression types in *mdts* must be flat (line 264.6). If two domain expressions evaluate to the same value, then the corresponding range expressions must also evaluate to the same value. This is in the Static Semantics checked in line 264.7-.11.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
265.0 wf-MapComprehension : Π →t MapComprehension → TypeR → Env → 𝔹
  .1 wf-MapComprehension (π)(mk-MapComprehension(ml, bindl, opred))(type)(env) ≙
  .2   let mk-Maplet(de, re) = ml in
  .3   ∃ venv : ValEnv, bindattrs, interprattrs : BindAttrs ·
  .4       wf-BindList(π)(bindl)(venv, bindattrs)(env) ∧
  .5       let newenv : Env = UpdateEnv(venv)(env),
  .6           wfe = InScope[Expr](π)(dom venv)(wf-Expr(π)) in
  .7       EMPTY ∈ bindattrs ∧ type = EMPTYMAP
  .8        ∨
  .9       wf-PredInScope(π)(opred)(venv, interprattrs)(env)
  .10       ∧
  .11      (EMPTY ∈ interprattrs ∧ type = EMPTYMAP
  .12       ∨
  .13       (FINITE ∈ bindattrs ∨ FINITE ∈ interprattrs) ∧
  .14       ∃ mk-UnionTypeR(dtset), mk-UnionTypeR(rtset) : TypeR ·
  .15           type = mk-GeneralMapTypeR(mk-UnionTypeR(dtset), mk-UnionTypeR(rtset)) ∧
  .16           ∀ dt ∈ dtset · wfe(de)(dt)(newenv) ∧ IsFlatTypeR(π)(dt)(env) ∧
  .17           ∀ rt ∈ rtset · wfe(re)(rt)(newenv) ∧
  .18           let mc = mk-MapComprehension(ml, bindl, opred) in
  .19           CompatibleMapLetBindings(π)(mc)(type)(env));

266.0 CompatibleMapLetBindings : Π →t MapComprehension → Type → Env → 𝔹
  .1 CompatibleMapLetBindings (π)(mk-MapComprehension(ml, bindl, opred))(-)(-) ≙
  .2   is-POS(π)
```

The Map Comprehension expression (265) is checked in the same way as the Set Comprehension expression (259). In the Dynamic Semantics, the maplet $ml = mk\text{-}Maplet(de, re)$ is evaluated in value environments originating from the bindlist *bindl*. If the domain expression in two value environments $venv_1, venv_2$ evaluates to the same value, then it is required that the range expression *re* in the same value environments $venv_1, venv_2$ also evaluate to the same value. The Static Semantics makes this check in *CompatibleMapLetBindings*.

14.3.14 Tuple Constructor

```
267.0 wf-TupleConstructor : Π →t TupleConstructor → TypeR → Env → 𝔹
  .1 wf-TupleConstructor (π)(mk-TupleConstructor(fields))(type)(env) ≙
  .2   let CP : TypeR* →t TypeR → Env → 𝔹
  .3       CP (fieldtypes)(rt)(env) ≙
  .4         rt = mk-ProductTypeR(fieldtypes)
  .5       in
  .6   wf-SCompExpr(π)(CP)(fields)(type)(env)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.15 Record Expressions

```
268.0 wf-RecordConstructor : Π -t-> RecordConstructor → TypeR → Env → 𝔹
   .1 wf-RecordConstructor (π)(mk-RecordConstructor(tag, fields))(type)(env) ≙
   .2   let CP : TypeR* -t-> TypeR → Env → 𝔹
   .3       CP (ftypes)(rt)(env) ≙
   .4         ∃ t : TypeR ·
   .5              IsTypeRwithTagInEnv(t, tag)(env) ∧ IsSubtypeR(π)(rt, t)(env) ∧
   .6              cases rt :
   .7                mk-CompositeTypeR(-, fieldRs) →
   .8                     len fieldRs = len ftypes ∧
   .9                     ∀ i ∈ inds fieldRs · ftypes(i) = fieldRs(i).type,
   .10               others → false
   .11             end
   .12      in
   .13   wf-SCompExpr(π)(CP)(fields)(type)(env)
```

In the record Constructor the used tag identifier *tag* must occur in some composite type in the type definitions (line 268.5). Note, the type *rt* of the constructed record may be a subtype of the composite type *t* indicated by the *tag*.

```
269.0 wf-RecordModifier : Π -t-> RecordModifier → TypeR → Env → 𝔹
   .1 wf-RecordModifier (π)(mk-RecordModifier(rec, modfs))(type)(env) ≙
   .2   let CP : TypeR* -t-> TypeR → Env → 𝔹
   .3       CP ([rect] ↷ newfldts)(rt)(env) ≙
   .4         cases rect :
   .5           mk-CompositeTypeR(tag, fields) →
   .6                let newfieldmap =
   .7                       {modfs(i).field ↦ newfldts(i) | i ∈ inds modfs},
   .8                   newfields =
   .9                       [if fields(i).id ∈ dom newfieldmap
   .10                      then mk-FieldR(fields(i).id, newfieldmap(fields(i).id))
   .11                      else fields(i) |
   .12                           i ∈ inds fields] in
   .13               rt = mk-CompositeTypeR(tag, newfields) ∧
   .14               ∃ definedt : TypeR ·
   .15                    IsTypeRwithTagInEnv(definedt, tag)(env) ∧
   .16                    IsSubtypeR(π)(rt, definedt)(env),
   .17          others → false
   .18        end
   .19     in
   .20   wf-SCompExpr(π)(CP)
   .21   ([rec] ↷ [modfs(i).new | i ∈ inds modfs])(type)(env)
```

The Record modifier is checked by using the general mechanism for checking strict compound expressions. The subcomponent expressions to be checked by *wf-SCompExpr* (line 269.20-.21) are the record expression *rec* (denoting the record value to be modified) and the modifiers *new*–expression (denoting the new value to be inserted in the record). Accordingly, the characteristic predicate *CP* states the type relations between the type *rect* of the original record value and the types *newfldts* of the field values.

The type *rect* must be a composite type (line 269.5). *newfieldmap* (line .269.6) is a map from the identifiers of the fields to be modified to the new type of the field. *newfields* (line 269.8) is the sequence of new field representations. It is composed of field representations from the type of the original record and field representations constructed from the *newfieldmap*. The type *rt* of the modified record (and consequently the type of the modifier expressions) must be a subtype of the composite type indicated by the record *tag* (line 269.14-.16). Notice that the type *rect* of the actual

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

record value to be modified migth already be such a subtype, so the originally defined composite type corresponding to *tag* must be looked up.

14.3.16 Apply Expressions

270.0 $wf\text{-}Apply : \Pi \xrightarrow{t} Apply \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}Apply\,(\pi)(mk\text{-}Apply(fnc, args))(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([fnct, argt])(rt)(env) \triangleq$
.4 $IsApplicationType(fnct)(argt, rt) \wedge$
.5 $(is\text{-}POS(\pi) \vee is\text{-}TotalFnTypeR(fnct))$
.6 in
.7 $wf\text{-}SCompExpr(\pi)(CP)([fnc, Exprs2Expr(args)])(type)(env)$;

271.0 $IsApplicationType : TypeR \xrightarrow{t} TypeR \times TypeR \to \mathbb{B}$

.1 $IsApplicationType\,(t)(at, rt) \triangleq$
.2 cases t :
.3 $mk\text{-}PartialFnTypeR((at), (rt)), mk\text{-}TotalFnTypeR((at), (rt)) \to$ true,
.4 $mk\text{-}GeneralMapTypeR((at), (rt)), mk\text{-}InjectiveMapTypeR((at), (rt)) \to$ true,
.5 $mk\text{-}Seq1\,TypeR((rt)) \to at =$ NATONE,
.6 others $\to$ false
.7 end;

272.0 $wf\text{-}FieldSelect : \Pi \xrightarrow{t} FieldSelect \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}FieldSelect\,(\pi)(mk\text{-}FieldSelect(record, field))(type)(env) \triangleq$
.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $CP\,([rect])(rt)(env) \triangleq$
.4 cases $rect$:
.5 $mk\text{-}CompositeTypeR(\text{-}, fields) \to$
.6 $\exists\, mk\text{-}FieldR(id, ft) \in$ elems $fields \cdot rt = ft \wedge field = id$,
.7 others $\to$ false
.8 end
.9 in
.10 $wf\text{-}SCompExpr(\pi)(CP)([record])(type)(env)$;

273.0 $wf\text{-}FctTypeInst : \Pi \xrightarrow{t} FctTypeInst \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}FctTypeInst\,(\pi)(mk\text{-}FctTypeInst(polyfctname, inst))(type)(env) \triangleq$
.2 $\exists\, mk\text{-}PolyValR(tpars, valr) : PolyValR \cdot$
.3 $IsBoundTo(polyfctname)(mk\text{-}PolyValR(tpars, valr))(env) \wedge$
.4 len $inst =$ len $tpars \wedge (is\text{-}DEF(\pi) \Rightarrow is\text{-}DEF(valr.wfclass)) \wedge$
.5 $\exists\, ts : TypeR^* \cdot$
.6 len $ts =$ len $tpars \wedge$
.7 $\forall\, i \in$ inds $inst \cdot wf\text{-}Type(\pi)(inst(i))(ts(i))(env) \wedge$
.8 $type = TypeSubst(\{tpars(i) \mapsto ts(i) \mid i \in$ inds $tpars\})(valr.type)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.17 Lambda Expressions

274.0 $wf\text{-}Lambda : \Pi \xrightarrow{t} Lambda \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}Lambda\,(\pi)(mk\text{-}Lambda(tbl, body_1))(type)(env) \triangleq$
.2 let $t = Types2Type([tbl(i).type \mid i \in \mathsf{inds}\ tbl])$,
.3 $ap = Pats2Pat([tbl(i).pat \mid i \in \mathsf{inds}\ tbl])$ in
.4 $\exists\, id : Id, body_2 : Expr, at, et, mk\text{-}UnionTypeR(rtset) : TypeR \cdot$
.5 $IsLetExpansion(body_2)(ap, \mathsf{nil}, mk\text{-}Name(id), body_1)(env) \wedge$
.6 $wf\text{-}Type(\pi)(t)(at)(env) \wedge IsEssEquivTypeR(\pi)(et, at)(env) \wedge$
.7 $\forall\, rt \in rtset \cdot$
.8 $wf\text{-}LambdaR(\pi)(mk\text{-}LambdaR(id, at, body_2, \mathsf{nil}, \mathsf{false}))(mk\text{-}PartialFnTypeR(at, rt))(env)$

The body expression *body* may be evaluated for all argument values corresponding to the argument type *at*, thereby yielding many different result values. In the Static Semantics, if the body expression is well-formed for the types t_1 and t_2, then the lambda expression is well-formed (among others) for the types $at \to t_1$, $at \to t_2 and\ at \to t_1 \mid t_2$. The result union type has been explicitly introduced in order to make possible well-formedness complete; for definite well-formedness this explicit introduction of union adds nothing (compare with set-, sequence- and map-comprehension, where one of the component expressions also represent many different values).

275.0 $wf\text{-}LambdaR : \Pi \xrightarrow{t} LambdaR \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}LambdaR\,(\pi)(mk\text{-}LambdaR(id, at, body, opre, assertedtotal))(fnt)(env) \triangleq$
.2 let $venv = \{mk\text{-}Name(id) \mapsto mk\text{-}ValR(at, \pi)\}$,
.3 $lenv = UpdateEnv(venv)(env)$ in
.4 $\exists\, verifiedtotal : \mathbb{B} \cdot$
.5 $Is[FunctionTypeR, TypeR](fnt) \wedge \neg\, IsEmptyType(NegWfClass(\pi))(fnt)(env) \wedge$
.6 $fnt.fndom = at$
.7 $\wedge$
.8 $(wf\text{-}FnBody(\pi)(body, opre, verifiedtotal, \mathsf{dom}\ venv)(fnt.fnrng)(lenv) \vee$
.9 $IsEmptyType(\pi)(at)(env))$
.10 $\wedge$
.11 $(assertedtotal \Rightarrow verifiedtotal) \wedge$
.12 $(is\text{-}TotalFnTypeR(fnt) \Rightarrow verifiedtotal \wedge$
.13 $(PreAlwaysTrue(\pi)(opre)(lenv) \vee IsEmptyType(\pi)(at)(env)))$

Note that there are three different propositions regarding totality involved here: (1) the specifier's assertion about definedness of the function on all arguments in the argument type for which the pre-condition holds; (2) the verified proposition regarding the definedness of the function on all arguments in the argument type for which the pre-condition holds; and (3) the function type which, if total, expresses that the function is defined on *all* arguments of the argument type (disregarding the pre-condition). Note that a general notion of total function types is used only in the Static Semantics. The concrete syntax of VDM-SL allows total function types only in the signature of explicit function definitions; in the core abstract syntax used for the Dynamic Semantics this usage has been abstracted as a Boolean value like the above "*assertedtotal*". Therefore, in the Static Semantics the meaning of total function types is as described above; this makes it possible to conclude that the application of a total function to an argument in the argument type produces a result in the result type. Functions with a non-trivial pre-condition may fail to behave this way and they will therefore not be well-formed with respect to a total function type. Also note that use of an empty argument type gives rise to a well-defined, total function even if the body is not well-formed. The result given for this case in the Dynamic Semantics is the function with the empty graph.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
276.0 wf-FnBody : Π →t Expr × [Expr] × 𝔹 × Name-set → TypeR → Env → 𝔹
  .1  wf-FnBody (π)(body, opre, verifiedtotal, pars)(restp)(env) ≜
  .2    let wfpred = if verifiedtotal
  .3                 then λ p : Π · InTotalFnScope(p)(pars)(wf-Expr(p))
  .4                 else λ p : Π · InScope[Expr](p)(pars)(wf-Expr(p)) in
  .5    PreNeverTrue(π)(opre)(env) ∨
  .6    cases π :
  .7      mk-DEF(cs) →
  .8          wfpred(π)(body)(restp)(env) ∨
  .9          ¬ verifiedtotal ∧
  .10         ¬ ∃ t : TypeR · wfpred(mk-POS(cs))(body)(t)(env),
  .11     mk-POS(cs) →
  .12         PreAlwaysTrue(mk-DEF(cs))(opre)(env) ⇒
  .13         if verifiedtotal
  .14         then ∃ rtset : TypeR-set ·
  .15                  restp = mk-UnionTypeR(rtset) ∧
  .16                  ∀ rt ∈ rtset ·
  .17                      wfpred(π)(body)(rt)(env)
  .18         else ¬ ∃ t : TypeR ·
  .19                  wfpred(mk-DEF(cs))(body)(t)(env) ∧
  .20                  IsDisjoint(mk-DEF(cs))(t, restp)(env)
  .21   end
```

Note that there are several special circumstances which will ensure well-formedness of a function body. If the pre-condition may never be true then the function result does not depend on the body. Therefore, the body may be considered well-formed no matter how it is expressed (276.5). In the case where totality is not being verified, the body is definitely well-formed if it always evaluates to $\perp$, since this is an allowed result of a partial function. The body will always evaluate to $\perp$ if it is not possibly well-formed for any type (276.10).

If it is not definitely the case that the pre-condition is always true then there may be cases where the function result does not depend on the body. Therefore, the body must be considered possibly well-formed (276.12). If, however, the pre-condition is definitely true then the function result always depends on the body. Considering partial functions, the body is allowed to evaluate to $\perp$. It must, however, not be the case that it definitely evaluates to values of a type which is disjoint from the result type (276.18-.20).

The body expression *body* may be evaluated for all argument values, thereby yielding many different result values. In the Static Semantics, if the body expression is well-formed for the types t_1 and t_2, then the lambda expression is well-formed (among others) for the types $at \to t_1$, $at \to t_2$ and $at \to t_1 \mid t_2$. The result union type has been explicitly introduced (line 276.15-.17) in order to make possible well-formedness for expressions complete (compare with set-, sequence- and map-comprehension, where one of the component expressions also represent many different values).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.18 Is Expressions

277.0 $wf\text{-}IsDefTypeExpr : \Pi \xrightarrow{t} IsDefTypeExpr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}IsDefTypeExpr\,(\pi)(mk\text{-}IsDefTypeExpr(tag, arg))(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([argt])(rt)(env) \triangleq$

.4 $rt = \text{BOOLEAN}$

.5 in

.6 $(\exists\, dt : TypeR \cdot IsTypeRwithTagInEnv(dt, tag)(env)) \wedge$

.7 $wf\text{-}SCompExpr(\pi)(CP)([arg])(type)(env)$;

278.0 $wf\text{-}IsBasicTypeExpr : \Pi \xrightarrow{t} IsBasicTypeExpr \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}IsBasicTypeExpr\,(\pi)(mk\text{-}IsBasicTypeExpr(type, arg))(type)(env) \triangleq$

.2 let $CP : TypeR^* \xrightarrow{t} TypeR \to Env \to \mathbb{B}$

.3 $CP\,([argt])(rt)(env) \triangleq$

.4 $rt = \text{BOOLEAN}$

.5 in

.6 $wf\text{-}SCompExpr(\pi)(CP)([arg])(type)(env)$

14.3.19 Names

279.0 $wf\text{-}Name : \Pi \xrightarrow{t} Name \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}Name\,(\pi)(name)(type)(env) \triangleq$

.2 $IsBoundToValTypeR(\pi)(name)(type)(env)$;

280.0 $wf\text{-}OldName : \Pi \xrightarrow{t} OldName \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}OldName\,(\pi)(oldname)(type)(env) \triangleq$

.2 $IsBoundToValTypeR(\pi)(oldname)(type)(env)$

14.3.20 Literals

281.0 $wf\text{-}Literal : \Pi \xrightarrow{t} Literal \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}Literal\,(\pi)(lit)(type)(env) \triangleq$

.2 cases lit :

.3 $mk\text{-}NumLit(\text{-}) \to wf\text{-}NumLit(\pi)(lit)(type)(env)$,

.4 $mk\text{-}BoolLit(\text{-}) \to type = \text{BOOLEAN}$,

.5 $mk\text{-}NilLit() \to type = \text{UNITTYPE}$,

.6 $mk\text{-}CharLit(\text{-}) \to type = \text{CHAR}$,

.7 $mk\text{-}TextLit(\text{-}) \to wf\text{-}TextLit(\pi)(lit)(type)(env)$,

.8 $mk\text{-}QuoteLit(\text{-}) \to type = mk\text{-}QuoteTypeR(lit)$

.9 end;

282.0 $wf\text{-}NumLit : \Pi \xrightarrow{t} NumLit \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}NumLit\,(\pi)(mk\text{-}NumLit(z))(type)(env) \triangleq$

.2 $type =$

.3 if is-$\mathbb{N}_1(z)$ then NATONE

.4 elseif is-$\mathbb{N}(z)$ then NAT

.5 elseif is-$\mathbb{Z}(z)$ then INT else RAT;

283.0 $wf\text{-}TextLit : \Pi \xrightarrow{t} TextLit \to TypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}TextLit\,(\pi)(mk\text{-}TextLit(cs))(type)(env) \triangleq$

.2 $type =$

.3 if len $cs = 0$ then EMPTYSEQ else $mk\text{-}Seq1TypeR(\text{CHAR})$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.3.21 Auxiliary Well-formedness Predicates

```
284.0 wf-PredInScope : Π →t [Expr] → ValEnv × BindAttrs → Env → 𝔹
  .1 wf-PredInScope (π)(opred)(venv, bindattrs)(env) ≜
  .2   opred = nil ∧ bindattrs = {}
  .3   ∨
  .4   let wfp = (InScope[Expr](π)(dom venv)) ∘ (InType[Expr](π))(wf-Expr(π)) in
  .5   wfp(opred)(BOOLEAN)(UpdateEnv(venv)(env)) ∧
  .6   cases π :
  .7     mk-POS(-) → bindattrs = {EMPTY, FINITE},
  .8     mk-DEF(-) → bindattrs = {}
  .9   end
```

wf-PredInScope checks the well-formedness of an optional expression *opred* which – if it exists – must be of type BOOLEAN (line 284.5). Because the expression is checked with respect to the specific type BOOLEAN is the *InType* rule applied (line 284.4). The expression is checked in the scope of the original environment extended with the value environment *venv*. In order to make the subenvironment based checks possible the *InScope* rule is applied (line 284.4).

The *bindattrs* describes how many dynamic environments conforming to the static semantics environment *env* that exists in which *opred* evaluates to true. For *is-DEF*(π) if EMPTY and/or FINITE is in the set *bindattrs* then the set of dynamic environments in which *opred* evaluates to true will be empty and/or finite respectively. For *is-POS*(π) if this set of dynamic environments is empty or finite then EMPTY or FINITE respectively will be in *bindattrs*.

14.4 State Designators

State designators occur either as the 'using' component in operation calls or as the left hand side of assignment statement s. Using subsumption on definite well-formedness of state designators in connection with assignment statement would yield an unsound system. In connection with operation calls, subsumption on well-formedness of the 'using' component is not necessary. Therefore subsumption is only used on possible well-formedness of state designators.

```
285.0 wf-StateDesignator : Π →t StateDesignator → TypeR → Env → 𝔹
  .1 wf-StateDesignator (π)(sd)(t)(env) ≜
  .2   cases sd :
  .3     mk-Name(-)          → ∃ sd_t : TypeR ·
  .4                               IsBoundTo(sd)(mk-VarR(sd_t))(env) ∧
  .5                               cases π :
  .6                                 mk-DEF(-) → sd_t = t,
  .7                                 mk-POS(-) → IsSubtype(π)(t, sd_t)(env)
  .8                               end,
  .9     mk-FieldRef(-, -)   → wf-FieldRef(π)(sd)(t)(env),
  .10    mk-MapOrSeqRef(-, -) → wf-MapOrSeqRef(π)(sd)(t)(env)
  .11  end;

286.0 wf-FieldRef : Π →t FieldRef → TypeR → Env → 𝔹
  .1 wf-FieldRef (π)(mk-FieldRef(var, sel))(t)(env) ≜
  .2   ∃ var_t : TypeR ·
  .3     wf-StateDesignator(π)(var)(var_t)(env) ∧
  .4     ∀ var_t' : TypeR ·
  .5       IsUnionComponent(var_t', var_t)(env) ⇒
  .6       is-CompositeTypeR(var_t') ∧ mk-FieldR(sel, t) ∈ elems var_t'.fields;
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
287.0 wf-MapOrSeqRef : Π →t MapOrSeqRef → TypeR → Env → 𝔹
   .1 wf-MapOrSeqRef (π)(mk-MapOrSeqRef(var, arg))(t)(env) ≜
   .2   cases π :
   .3     mk-DEF(-) → false,
   .4     mk-POS(-) →
   .5         ∃ var_t, arg_t : TypeR ·
   .6             wf-StateDesignator(π)(var)(var_t)(env) ∧
   .7             wf-Expr(π)(arg)(arg_t)(env) ∧
   .8             cases var_t :
   .9               mk-Seq1 TypeR(et) →
   .10                  IsSubtype(π)(arg_t, NATONE)(env) ∧ Subsume(π)(et, t)(env),
   .11              mk-GeneralMapTypeR(mapdom, maprng) →
   .12                  IsSubtype(π)(arg_t, mapdom)(env) ∧ Subsume(π)(maprng, t)(env),
   .13              mk-InjectiveMapTypeR(mapdom, maprng) →
   .14                  IsSubtype(π)(arg_t, mapdom)(env) ∧ Subsume(π)(maprng, t)(env),
   .15              others → false
   .16            end
   .17 end
```

14.5 Statements

Statements are well-formed with respect to statement types *StmtTypeR*. In operational terms, the execution of a statement may either terminate abnormally by exiting or returning, or it may terminate normally and leave the control to the executing context. If a statement s definitely has type t it means that s will definitely terminate its executing with a result of type t. If a statement s is possible well-formed with respect to a statement type t, it means that it cannot be rejected that the statement yields a result of statement type t. In this way it is possible to check whether the body of an operation is well-defined, i.e., whether it returns with a value of the right type. Note that a statement is only definitely well-formed if it does not loop. Soundness requirements for the statement well-formedness predicates are similar to those of expressions. For this reason, we use subsumption in connection with statements as well as done in connection with expressions.

```
288.0 wf-Stmt : Π →t Stmt → StmtTypeR → Env → 𝔹
   .1 wf-Stmt (π)(s)(t)(env) ≜
   .2   cases s :
   .3     mk-LetStmt(-,-)              → wf-LetStmt(π)(s)(t)(env),
   .4     mk-LetBeSTStmt(-,-,-)        → wf-LetBeSTStmt(π)(s)(t)(env),
   .5     mk-DefStmt(-,-)              → wf-DefStmt(π)(s)(t)(env),
   .6     mk-BlockStmt(-,-)            → wf-BlockStmt(π)(s)(t)(env),
   .7     mk-AssignStmt(-,-)           → wf-AssignStmt(π)(s)(t)(env),
   .8     mk-IfStmt(-,-,-,-)           → wf-IfStmt(π)(s)(t)(env),
   .9     mk-CasesStmt(-,-,-)          → wf-CasesStmt(π)(s)(t)(env),
   .10    mk-SeqForLoop(-,-,-,-)       → wf-SeqForLoop(π)(s)(t)(env),
   .11    mk-SetForLoop(-,-,-)         → wf-SetForLoop(π)(s)(t)(env),
   .12    mk-IndexForLoop(-,-,-,-,-)   → wf-IndexForLoop(π)(s)(t)(env),
   .13    mk-WhileLoop(-,-)            → wf-WhileLoop(π)(s)(t)(env),
   .14    mk-NonDetStmt(-)             → wf-NonDetStmt(π)(s)(t)(env),
   .15    mk-Call(-,-,-)               → wf-Call(π)(s)(t)(env),
   .16    mk-ReturnStmt(-)             → wf-ReturnStmt(π)(s)(t)(env),
   .17    mk-AlwaysStmt(-,-)           → wf-AlwaysStmt(π)(s)(t)(env),
   .18    mk-TrapStmt(-,-,-)           → wf-TrapStmt(π)(s)(t)(env),
   .19    mk-RecTrapStmt(-,-)          → wf-RecTrapStmt(π)(s)(t)(env),
   .20    mk-ExitStmt(-)               → wf-ExitStmt(π)(s)(t)(env),
   .21    (SKIP)                       → wf-IdentStmt(π)(s)(t)(env)
   .22  end
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.1 Local Binding Statements

```
289.0 wf-LetStmt : Π →t LetStmt → StmtTypeR → Env → 𝔹
  .1 wf-LetStmt (π)(mk-LetStmt(letdefs, body))(t)(env) ≙
  .2   ∃ venv : VisibleEnv ·
  .3         wf-ValFnOpDefs(π)(letdefs)(venv)(env) ∧
  .4         InScope[Stmt](π)(dom venv)(wf-Stmt(π))(body)(t)(UpdateEnv(venv)(env));

290.0 wf-LetBeSTStmt : Π →t LetBeSTStmt → StmtTypeR → Env → 𝔹
  .1 wf-LetBeSTStmt (π)(mk-LetBeSTStmt(bind, cond, body))(t)(env) ≙
  .2   (∃ venv : ValEnv, bindattrs : BindAttrs ·
  .3         wf-Bind(π)(bind)(venv, bindattrs)(env) ∧
  .4         let newenv = UpdateEnv(venv)(env),
  .5             wfs = (InScope[Stmt](π)(dom venv)) ∘ (InType[Stmt](π))(wf-Stmt(π)) in
  .6         (cond ≠ nil ⇒
  .7          ∃ truetp : StmtTypeR ·
  .8               wfs(cond)(truetp)(newenv) ∧ IsTrueType(π)(truetp)(env)) ∧
  .9         InScope[Stmt](π)(dom venv)(wf-Stmt(π))(body)(t)(newenv)) ∧
  .10  ¬ ∃ venv : ValEnv, bindattrs : BindAttrs ·
  .11        wf-Bind(NegWfClass(π))(bind)(venv, bindattrs)(env) ∧
  .12        EMPTY ∈ bindattrs;

291.0 wf-DefStmt : Π →t DefStmt → StmtTypeR → Env → 𝔹
  .1 wf-DefStmt (π)(mk-DefStmt(eqdefs, body))(t)(env) ≙
  .2   ∃ venv : VisibleEnv, eqdef_t : StmtTypeR ·
  .3         wf-EqDef(π)(hd eqdefs)(venv, eqdef_t)(env) ∧
  .4         ∀ eqdef_t' : StmtTypeR ·
  .5              IsUnionComponent(eqdef_t', eqdef_t)(env) ⇒
  .6              cases eqdef_t' :
  .7                (CONT) → let stmt = if len eqdefs = 1
  .8                                    then body
  .9                                    else mk-DefStmt(tl eqdefs, body),
  .10                            wfs = InScope[Stmt](π)(dom venv)(wf-Stmt(π)) in
  .11                        wfs(stmt)(t)(UpdateEnv(venv)(env)),
  .12                others   → Subsume(π)(eqdef_t', t)(env)
  .13             end
```

A 'def' statement is checked by considering the equalities one at a time. Note that the well-formedness of the entailing equalities and the body is only required if the initialization terminates normally (291.7-.11).

```
292.0 wf-EqDef : Π →t EqDef → VisibleEnv × StmtTypeR → Env → 𝔹
  .1 wf-EqDef (π)(mk-EqDef(lhs, rhs))(venv, t)(env) ≙
  .2   ∃ rhs_t : TypeR ·
  .3         wf-CallOrExpr(π)(rhs)(rhs_t)(env) ∧
  .4         ∀ rhs_t' : TypeR ·
  .5              IsUnionComponent(rhs_t', rhs_t)(env) ∧
  .6              cases rhs_t' :
  .7                (CONT)                → false,
  .8                mk-RetTypeR((VOID))   → false,
  .9                mk-ExitTypeR(-)       → Subsume(π)(rhs_t', t)(env),
  .10               mk-RetTypeR(rt)       → wf-Pattern(π)(lhs)(mk-TVE(rt, venv))(env) ∧
  .11                                       Subsume(π)(CONT, t)(env),
  .12               others                → wf-Pattern(π)(lhs)(mk-TVE(rhs_t', venv))(env) ∧
  .13                                       Subsume(π)(CONT, t)(env)
  .14             end
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If the initialization exits, the entire statement exits. Otherwise the initialization expression or operation call must return with a well-defined value (292.6-.14).

14.5.2 Block and Assignment Statements

293.0 $wf\text{-}BlockStmt : \Pi \xrightarrow{t} BlockStmt \to StmtTypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}BlockStmt\,(\pi)(mk\text{-}BlockStmt(dcls, stmts))(t)(env) \triangleq$
.2 if $dcls = []$
.3 then $wf\text{-}Sequence(\pi)(stmts)(t)(env)$
.4 else $\exists\, venv : VisibleEnv, dcl_t : StmtTypeR \cdot$
.5 $wf\text{-}AssignDef(\pi)(\mathsf{hd}\ dcls)(venv, dcl_t)(env) \wedge$
.6 $\forall\, dcl_{t'} : StmtTypeR \cdot$
.7 $IsUnionComponent(dcl_{t'}, dcl_t)(env) \Rightarrow$
.8 cases $dcl_{t'}$:
.9 $(\text{CONT}) \to$ let $stmt = mk\text{-}BlockStmt(\mathsf{tl}\ dcls, stmts)$ in
.10 $wf\text{-}Stmt(\pi)(stmt)(t)(UpdateEnv(venv)(env))$,
.11 others $\to Subsume(\pi)(dcl_{t'}, t)(env)$
.12 end

The *decls*-part of a *BlockStmt* declares local *variables*, which can be changed by assignment statements in the *stmts*-part. Accordingly, the sub-environment based checks (imposed by *InScope*) cannot be applied here. Block statements are checked by considering the declarations one at a time. Note that the well-formedness of the entailing declarations and the body is only required if the initialisation terminates normally (293.9-.10).

294.0 $wf\text{-}Sequence : \Pi \xrightarrow{t} Stmt^* \to StmtTypeR \to Env \to \mathbb{B}$

.1 $wf\text{-}Sequence\,(\pi)(stmts)(t)(env) \triangleq$
.2 if $stmts = []$
.3 then $Subsume(\pi)(\text{CONT}, t)(env)$
.4 else $\exists\, stmt_t : StmtTypeR \cdot$
.5 $wf\text{-}Stmt(\pi)(\mathsf{hd}\ stmts)(stmt_t)(env) \wedge$
.6 $\forall\, stmt_{t'} : StmtTypeR \cdot$
.7 $IsUnionComponent(stmt_{t'}, stmt_t)(env) \Rightarrow$
.8 cases $stmt_{t'}$:
.9 $(\text{CONT}) \to wf\text{-}Sequence(\pi)(\mathsf{tl}\ stmts)(t)(env)$,
.10 others $\to Subsume(\pi)(stmt_{t'}, t)(env)$
.11 end;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
295.0 wf-AssignDef : Π →t AssignDef → VisibleEnv × StmtTypeR → Env → 𝔹
  .1 wf-AssignDef (π)(mk-AssignDef(var, tp, dclinit))(venv, t)(env) ≜
  .2   ∃ tp_t, et : TypeR ·
  .3       wf-Type(π)(tp)(tp_t)(env) ∧
  .4       IsNonEmptyEssEquivTypeR(π)(et, tp_t)(env) ∧
  .5       venv = {mk-Name(var) ↦ mk-VarR(et)} ∧
  .6       if dclinit ≠ nil
  .7       then ∃ dclinit_t : TypeR ·
  .8                wf-CallOrExpr(π)(dclinit)(dclinit_t)(env) ∧
  .9                ∀ dclinit_t' : TypeR ·
  .10                   IsUnionComponent(dclinit_t', dclinit_t)(env) ⇒
  .11                   cases dclinit_t' :
  .12                     (CONT)              → false,
  .13                     mk-RetTypeR((VOID)) → false,
  .14                     mk-ExitTypeR(-)     → Subsume(π)(dclinit_t', t)(env),
  .15                     mk-RetTypeR(rt)     → IsSubtypeR(π)(rt, tp_t)(env) ∧
  .16                                           Subsume(π)(CONT, t)(env),
  .17                     others              → IsSubtypeR(π)(dclinit_t', tp_t)(env) ∧
  .18                                           Subsume(π)(CONT, t)(env)
  .19                   end
  .20       else Subsume(π)(CONT, t)(env)
```

If the initialisation exits, the whole declaration exits. Otherwise the initialisation expression or operation call—if present—must return with a well-defined value (295.11-.19).

```
296.0 wf-CallOrExpr : Π →t Call | Expr → TypeR → Env → 𝔹
  .1 wf-CallOrExpr (π)(c)(t)(env) ≜
  .2   if is-Call(c) then wf-Call(π)(c)(t)(env)
  .3   else wf-Expr(π)(c)(t)(env);
```

```
297.0 wf-AssignStmt : Π →t AssignStmt → StmtTypeR → Env → 𝔹
  .1 wf-AssignStmt (π)(mk-AssignStmt(lhs, rhs))(t)(env) ≜
  .2   ∃ lhs_t, rhs_t : TypeR ·
  .3       wf-StateDesignator(π)(lhs)(lhs_t)(env) ∧
  .4       wf-CallOrExpr(π)(rhs)(rhs_t)(env) ∧
  .5       ∀ rhs_t' : TypeR ·
  .6           IsUnionComponent(rhs_t', rhs_t)(env) ∧
  .7           cases rhs_t' :
  .8             (CONT)              → false,
  .9             mk-RetTypeR((VOID)) → false,
  .10            mk-ExitTypeR(-)     → Subsume(π)(rhs_t', t)(env),
  .11            mk-RetTypeR(rt)     → IsSubtype(π)(rt)(lhs_t)(env) ∧
  .12                                  Subsume(π)(CONT, t)(env),
  .13            others              → IsSubtype(π)(rhs_t)(lhs_t)(env) ∧
  .14                                  Subsume(π)(CONT, t)(env)
  .15          end
```

If the right hand side of an assignment raises an exception, the entire assignment statement raises an exception (297.10). Otherwise, the right hand side must yield a well-defined value (297.11-.12, 297.13-.14).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.3 Conditional Statements

```
298.0 wf-IfStmt : Π →t IfStmt → StmtTypeR → Env → 𝔹
  .1 wf-IfStmt (π)(mk-IfStmt(test, cons, elsifaltn, altn))(t)(env) ≜
  .2   let altn' = if elsifaltn = []
  .3                then altn
  .4                else mk-IfStmt((hd elsifaltn).test, (hd elsifaltn).cons, tl elsifaltn, altn) in
  .5   InType[Expr](π)(wf-Expr(π))(test)(BOOLEAN)(env) ∧
  .6   cases π :
  .7     mk-DEF(-) → wf-Stmt(π)(cons)(t)(env) ∧
  .8                 wf-Stmt(π)(altn')(t)(env),
  .9     mk-POS(-) → wf-Stmt(π)(cons)(t)(env) ∨
  .10                wf-Stmt(π)(altn')(t)(env)
  .11  end;

299.0 wf-CasesStmt : Π →t CasesStmt → StmtTypeR → Env → 𝔹
  .1 wf-CasesStmt (π)(e)(t)(env) ≜
  .2   let mk-CasesStmt(sel, altns, (nil)) = CasesStmt2CasesStmt(e),
  .3     π_POS = mk-POS(π.checkset) in
  .4   ∃ sel_t : TypeR ·
  .5       wf-Expr(π)(sel)(sel_t)(env) ∧
  .6       let altns' = SimplifyCaseStmtAlternatives(π.checkset)(altns)(sel_t)(env) in
  .7       cases π :
  .8         mk-DEF(-) →
  .9             ∀ alt ∈ elems altns' ·
  .10                ∀ pat ∈ elems alt.match ·
  .11                    ∃ venv : ValEnv ·
  .12                        wf-Pattern(π_POS)(pat)(mk-TVE(sel_t, venv))(env) ⇒
  .13                        wf-Stmt(π)(alt.body)(t)(UpdateEnv(venv)(env)),
  .14        mk-POS(-) →
  .15            ∃ alt ∈ elems altns' ·
  .16                ∃ pat ∈ elems alt.match ·
  .17                    ∃ venv : ValEnv ·
  .18                        wf-Pattern(π_POS)(pat)(mk-TVE(sel_t, venv))(env) ⇒
  .19                        let wfs = InScope[Stmt](π)(dom venv)(wf-Stmt(π)) in
  .20                        wfs(alt.body)(t)(UpdateEnv(venv)(env))
  .21      end;

300.0 SimplifyCaseStmtAlternatives : Check-set →t CaseStmtAlternatives →
  .1                                 TypeR → Env → CaseStmtAlternatives
  .2 SimplifyCaseStmtAlternatives (checkset)(altns)(sel_t)(env) ≜
  .3   let match = (hd altns).match,
  .4     π_POS = mk-POS(checkset),
  .5     π_DEF = mk-DEF(checkset),
  .6     match' = [match(i) | i ∈ inds match ·
  .7                  ∃ venv : ValEnv ·
  .8                      wf-Pattern(π_POS)(match(i))(mk-TVE(sel_t, venv))(env)],
  .9     altns' = if len altns = 1 ∨
  .10               ∃ venv : ValEnv, i ∈ inds match ·
  .11                   wf-Pattern(π_DEF)(match(i))(mk-TVE(sel_t, venv))(env)
  .12             then []
  .13             else SimplifyCaseStmtAlternatives(checkset)(tl altns)(sel_t)(env) in
  .14  if match' = []
  .15  then altns'
  .16  else [mk-CaseStmtAltn(match, (hd altns).body)] ⌢ altns'
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Given a type of the selector in the 'cases' statement, some of the alternatives may never come into consideration during ex ecution.

```
301.0 CasesStmt2CasesStmt : CasesStmt -t-> CasesStmt
  .1 CasesStmt2CasesStmt (mk-CasesStmt(sel, altns, other)) ≜
  .2   let otherbody = if other = nil then SKIP else other,
  .3       otheraltn = mk-CaseStmtAltn([mk-PatternId(nil)], otherbody) in
  .4   mk-CasesStmt(sel, altns ↷ [otheraltn], nil)
```

14.5.4 Loop Statements

```
302.0 wf-SeqForLoop : Π -t-> SeqForLoop → StmtTypeR → Env → 𝔹
  .1 wf-SeqForLoop (π)(mk-SeqForLoop(cv, dirn, forseq, body))(t)(env) ≜
  .2   ∃ forseq_t : SeqTypeR | EMPTYSEQ ·
  .3       wf-Expr(π)(forseq)(forseq_t)(env) ∧
  .4       cases forseq_t :
  .5         (EMPTYSEQ) →
  .6             Subsume(π)(CONT, t)(env),
  .7         mk-Seq1TypeR(elemtp) →
  .8             ∃ venv : ValEnv ·
  .9                 wf-PatternBind(π)(cv)(mk-TVE(elemtp, venv))(env) ∧
  .10                InScope[Stmt](π)(dom venv)(wf-Stmt(π))(body)(t)(UpdateEnv(venv)(env)),
  .11        others → false
  .12      end
```

Iterating over a sequence terminates normally if the sequence is empty (302.6). Otherwise the body will be executed (302.8-.10). Note that infinite looping is not possible since sequences have finite length.

```
303.0 wf-SetForLoop : Π -t-> SetForLoop → StmtTypeR → Env → 𝔹
  .1 wf-SetForLoop (π)(mk-SetForLoop(cv, forset, body))(t)(env) ≜
  .2   ∃ forset_t : SetTypeR | EMPTYSET ·
  .3       wf-Expr(π)(forset)(forset_t)(env) ∧
  .4       cases forset_t :
  .5         (EMPTYSET) →
  .6             Subsume(π)(CONT, t)(env),
  .7         mk-SetTypeR(elemtp) →
  .8             ∃ venv : ValEnv ·
  .9                 wf-Pattern(π)(cv)(mk-TVE(elemtp, venv))(env) ∧
  .10                InScope[Stmt](π)(dom venv)(wf-Stmt(π))(body)(t)(UpdateEnv(venv)(env)),
  .11        others → false
  .12      end
```

Iteration over sets is almost similar to iteration over sequences.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

304.0 $wf\text{-}IndexForLoop : \Pi \xrightarrow{t} IndexForLoop \to StmtTypeR \to Env \to \mathbb{B}$

```
.1  wf-IndexForLoop (π)(mk-IndexForLoop(cv, lb, ub, step, body))(t)(env) ≙
.2    let CP : TypeR* → TypeR → Env → 𝔹
.3        CP ([lb_t, ub_t, step_t])(t)(env) ≙
.4          IsSubtype(π)(lb_t, INT)(env) ∧
.5          IsSubtype(π)(ub_t, INT)(env) ∧
.6          IsSubtype(π)(step_t, INT)(env) ∧
.7          t = mk-UnionTypeR({lb_t, ub_t})
.8        in
.9    ∃ index_t : TypeR ·
.10       let step' = if step = nil then mk-NumLit(1) else step in
.11       wf-SCompExpr(π)(CP)([lb, ub, step'])(index_t)(env) ∧
.12       let env_body = UpdateEnv({mk-Name(cv) ↦ mk-ValR(index_t, π)})(env) in
.13       cases π :
.14         mk-DEF(-) → Subsume(π)(CONT, t)(env) ∧
.15                     wf-Stmt(π)(body)(t)(env_body),
.16         mk-POS(-) → Subsume(π)(CONT, t)(env) ∨
.17                     InScope[Stmt](π)({mk-Name(cv)})(wf-Stmt(π))(body)(t)(env_body)
.18       end
```

The lower bound, upper bound and—if present—the step expression of an 'index-for-loop' statement must be integers (304.4-.7). Given the possibly empty set of indices the body may be executed or not.

305.0 $wf\text{-}WhileLoop : \Pi \xrightarrow{t} WhileLoop \to StmtTypeR \to Env \to \mathbb{B}$

```
.1  wf-WhileLoop (π)(mk-WhileLoop(test, body))(t)(env) ≙
.2    InType[Expr](π)(wf-Expr(π))(test)(BOOLEAN)(env) ∧
.3    cases π :
.4      mk-DEF(-) → Subsume(π)(CONT, t)(env) ∧
.5                  ∃ body_t : StmtTypeR ·
.6                      wf-Stmt(π)(body)(body_t)(env) ∧
.7                      IsDisjoint(π)(CONT, body_t)(env) ∧
.8                      Subsume(π)(body_t, t)(env),
.9      mk-POS(-) → Subsume(π)(CONT, t)(env) ∨
.10                 wf-Stmt(π)(body)(t)(env)
.11   end
```

The test of a while loop must be Boolean (305.2). The body of a while statement may be executed or not (305.3-.11). If the body of a while loop does not terminate by exiting or returning, termination cannot be ensured and the entire while statement is therefore not definitely well-formed (305.5-.8).

14.5.5 Non-Deterministic Statement

306.0 $wf\text{-}NonDetStmt : \Pi \xrightarrow{t} NonDetStmt \to StmtTypeR \to Env \to \mathbb{B}$

```
.1  wf-NonDetStmt (π)(mk-NonDetStmt(stmts))(t)(env) ≙
.2    cases π :
.3      mk-DEF(-) → ∀ stmts' : Stmt* ·
.4                      IsPermutationOf[Stmt](stmts', stmts) ⇒
.5                      wf-Sequence(π)(stmts')(t)(env),
.6      mk-POS(-) → ∃ stmts' : Stmt* ·
.7                      IsPermutationOf[Stmt](stmts', stmts) ⇒
.8                      wf-Sequence(π)(stmts')(t)(env)
.9    end
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.5.6 Call and Return Statements

```
307.0 wf-Call : Π →t Call → StmtTypeR → Env → 𝔹
   .1 wf-Call (π)(mk-Call(oprt, args, callst))(t)(env) ≜
   .2   let arg = Exprs2Expr(args) in
   .3   ∃ oprt_t : OpTypeR, arg_t, callst_t : TypeR ·
   .4         wf-Expr(π)(oprt)(oprt_t)(env) ∧
   .5         wf-Expr(π)(arg)(arg_t)(env) ∧
   .6         IsSubtype(π)(arg_t, oprt_t.opdom)(env) ∧
   .7         Subsume(π)(oprt_t.oprng, t)(env) ∧
   .8         (if callst = nil
   .9          then oprt_t.opstate = nil
   .10         else oprt_t.opstate ≠ nil ∧
   .11               wf-StateDesignator(π)(callst)(callst_t)(env) ∧
   .12               IsSubtype(π)(callst_t, oprt_t.opstate)(env)) ∧
   .13         is-POS(π)
```

In an operation call, the actual argument must be of the formal argument type (307.6). If the operation is defined not to operate on a global state, it must be called without a 'using' state (307.9). Otherwise the type of the 'using' state must be a subtype of the state the operation is defined to operate on (307.10-.12). Note that an operation call cannot be accepted as being definitely well-formed since termination cannot be ensured (307.13).

```
308.0 wf-ReturnStmt : Π →t ReturnStmt → StmtTypeR → Env → 𝔹
   .1 wf-ReturnStmt (π)(mk-ReturnStmt(expr))(t)(env) ≜
   .2   ∃ expr_t : [TypeR] ·
   .3         (if expr = nil
   .4          then expr_t = VOID
   .5          else wf-Expr(π)(expr)(expr_t)(env)) ∧
   .6         Subsume(π)(mk-RetTypeR(expr_t), t)(env)
```

14.5.7 Exception Handling Statements

```
309.0 wf-AlwaysStmt : Π →t AlwaysStmt → StmtTypeR → Env → 𝔹
   .1 wf-AlwaysStmt (π)(mk-AlwaysStmt(alwpost, body))(t)(env) ≜
   .2   ∃ alwpost_t : StmtTypeR ·
   .3         wf-Stmt(π)(alwpost)(alwpost_t)(env) ∧
   .4         ∀ alwpost_t' : StmtTypeR ·
   .5               IsUnionComponent(alwpost_t', alwpost_t)(env) ⇒
   .6               cases alwpost_t' :
   .7                  mk-ExitTypeR(-) → Subsume(π)(alwpost_t', t)(env),
   .8                  others          → wf-Stmt(π)(body)(t)(env)
   .9               end
```

If the post statement of the 'always' handler statement exits, the whole statement exits (309.7). Otherwise the result of the execution is the result of the body statement (309.8).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
310.0 wf-TrapStmt : Π -t-> TrapStmt → StmtTypeR → Env → 𝔹
  .1 wf-TrapStmt (π)(mk-TrapStmt(pat, trappost, body))(t)(env) ≜
  .2   let π_DEF = mk-DEF(π.checkset),
  .3       π_POS = mk-POS(π.checkset) in
  .4   ∃ body_t : StmtTypeR ·
  .5       wf-Stmt(π)(body)(body_t)(env) ∧
  .6       ∀ body_t' : StmtTypeR ·
  .7           IsUnionComponent(body_t', body_t)(env) ⇒
  .8           cases body_t' :
  .9             mk-ExitTypeR((VOID)) → false,
  .10            mk-ExitTypeR(et) →
  .11                cases π :
  .12                  mk-DEF(-) →
  .13                      (∀ venv : ValEnv ·
  .14                          wf-Pattern(π_POS)(pat)(mk-TVE(et, venv))(env) ⇒
  .15                          wf-Stmt(π)(trappost)(t)(UpdateEnv(venv)(env))) ∧
  .16                      (¬ ∃ venv : ValEnv ·
  .17                          wf-Pattern(π_DEF)(pat)(mk-TVE(et, venv))(env) ⇒
  .18                       Subsume(π)(body_t', t)(env)),
  .19                  mk-POS(-) →
  .20                      (∃ venv : ValEnv ·
  .21                          let wfs = InScope[Stmt](π)(dom venv)(wf-Stmt(π)) in
  .22                          wf-Pattern(π_POS)(pat)(mk-TVE(et, venv))(env) ∧
  .23                          wfs(trappost)(t)(UpdateEnv(venv)(env))) ∨
  .24                      ¬ ∃ venv : ValEnv ·
  .25                          wf-Pattern(π_DEF)(pat)(mk-TVE(et, venv))(env) ∧
  .26                      Subsume(π)(body_t', t)(env)
  .27                end,
  .28              others → Subsume(π)(body_t', t)(env)
  .29          end
```

The body of the non-recursive trap statement is not allowed to exit without a value (310.9). If the body raises an exception with a value which matches the trap pattern (310.13-.15, 310.21-.23), the result of the execution is the result of the post statement. If the exit value does not match the trap pattern, the exception is passed (310.16-.18, 310.24-.26). If no exception is raised in the body, the result of the body is passed (310.28).

```
311.0 wf-RecTrapStmt : Π -t-> RecTrapStmt → StmtTypeR → Env → 𝔹
  .1 wf-RecTrapStmt (π)(mk-RecTrapStmt(traps, body))(t)(env) ≜
  .2   ∃ body_t : StmtTypeR ·
  .3       wf-Stmt(π)(body)(body_t)(env) ∧
  .4       wf-Traps(π)(traps)(body_t)(t)(env)
```

The result of the execution of the body of a recursive trap statement is passed to the traps.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
312.0 wf-Traps : Π →t Traps → StmtTypeR → StmtTypeR → Env → 𝔹
   .1 wf-Traps (π)(traps)(body_t)(t)(env) ≙
   .2   let π_DEF = mk-DEF(π.checkset),
   .3       π_POS = mk-POS(π.checkset) in
   .4   ∀ body_t' : StmtTypeR ·
   .5       IsUnionComponent(body_t', body_t)(env) ⇒
   .6       cases body_t' :
   .7         mk-ExitTypeR((VOID)) → false,
   .8         mk-ExitTypeR(et) →
   .9             cases π :
   .10             mk-DEF(cs) →
   .11                 ¬ ∃ trp ∈ elems traps, venv : ValEnv ·
   .12                     wf-PatternBind(π_POS)(trp.match)(mk-TVE(et, venv))(env),
   .13             mk-POS(cs) →
   .14                 (∃ trp ∈ elems traps, venv : ValEnv, trappost_t : StmtTypeR ·
   .15                     let newenv = UpdateEnv(venv)(env),
   .16                         pat = trp.match,
   .17                         wfs = InScope[Stmt](π)(dom venv)(wf-Stmt(π)) in
   .18                     wf-PatternBind(π_POS)(pat)(mk-TVE(et, venv))(env) ∧
   .19                     wfs(trp.trappost)(trappost_t)(newenv) ∧
   .20                     wf-Traps(π)(traps)(trappost_t)(t)(env)) ∨
   .21                 ¬ ∃ trp ∈ elems traps, venv : ValEnv ·
   .22                     wf-PatternBind(π_DEF)(trp.match)(mk-TVE(et, venv))(env) ∧
   .23                 Subsume(π)(body_t', t)(env)
   .24             end,
   .25         others → Subsume(π)(body_t', t)(env)
   .26       end
```

The body of a recursive trap statement is not allowed to raise an exception without a value (312.7). For definite well-formedness, we do not allow the body to exit with a value which matches any of the patterns in the traps (312.11-.12). This is a safe way to ensure termination. If the body raises an exception, the recursive trap statement may result in the execution of any of the trap post statements with a guard which matches the exception value (312.14-.20). Otherwise, if the exception value does not match any of the guards, the exception is passed (312.21-.23). If the body does not raise an exception, the result of the body is passed (312.25).

```
313.0 wf-ExitStmt : Π →t ExitStmt → StmtTypeR → Env → 𝔹
   .1 wf-ExitStmt (π)(mk-ExitStmt(expr))(t)(env) ≙
   .2   ∃ expr_t : [TypeR] ·
   .3       (if expr = nil
   .4        then expr_t = VOID
   .5        else wf-Expr(π)(expr)(expr_t)(env)) ∧
   .6       Subsume(π)(mk-ExitTypeR(expr_t), t)(env)
```

14.5.8 Identity Statements

```
314.0 wf-IdentStmt : Π →t SKIP → StmtTypeR → Env → 𝔹
   .1 wf-IdentStmt (π)((SKIP))(t)(env) ≙
   .2   Subsume(π)(CONT, t)(env)
```

14.6 Patterns and Bindings

In this clause the well-formedness predicates for patterns and bindings are defined.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.1 Patterns

The Static Semantics defines for each pattern class (*XPattern*) a basic well-formedness predicate indexed by the the well-formedness classification:

$$wf\text{-}XPattern : \Pi \xrightarrow{t} Pattern \to TVE \to Env \to \mathbb{B}$$
$$wf\text{-}XPattern\,(\pi)(pat)(mk\text{-}TVE(t, venv))(env) \triangleq$$
$$\quad some\text{-}expr\text{-}to\text{-}be\text{-}explained\text{-}in\text{-}the\text{-}following$$

where

315.0 $ValEnv = Name \xrightarrow{m} ValR;$

316.0 $TVE :: type : TypeR$
.1 $venv : ValEnv$

The *TVE* groups together the *type* representing the value to be matched with the pattern and the value environment $venv : ValEnv$ representing the corresponding dynamic value environments generated in the dynamic semantics. A dynamic semantics value environment *dvenv* conforms to a static semantics value environment $venv : ValEnv$ if *dvenv* and *venv* have the same domain, and for every identifier *id* in the domain is $dvenv(id)$ a value in the set of values described by the type $venv(id)$. If a pattern is well-formed for some $mk\text{-}TVE(t, venv)$, then the generated dynamic value environments *conform* with *venv*.

14.6.1.1 Pattern Characteristic Predicates

All compound patterns are defined in terms of so-called *pattern characteristic predicates.* A pattern characteristic predicate defines the relationship between the type-value-environments (TVE) of the subcomponent patterns and the type-value-environment of the whole pattern. The pattern characteristic predicate has the type *PatCP*. In the definition below it is named *PCP*. The part of the pattern characteristic predicate which concerns the value environments is common for all compound patterns. The part of the pattern characteristic predicate which concerns the relationship between the match type for the compound pattern and the subcomponent patterns depends on the considered pattern construct. It is defined locally for each compound pattern's well-formedness predicate and is called the *pattern type characteristic predicate.* It has the type $PatTCP$, and is named $PTCP$.

317.0 $PatCP = TVE^+ \to TVE \to Env \to \mathbb{B};$

318.0 $PatTCP = TypeR^+ \to TypeR \to Env \to \mathbb{B}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
319.0 wf-CompPatn : Π →t PatTCP → Pattern+ → TVE → Env → 𝔹
   .1 wf-CompPatn (π)(PTCP)(cp-seq)(mtve)(env) ≙
   .2   let PCP : TVE+ → TVE → Env → 𝔹
   .3       PCP (tve-seq)(mk-TVE(mt, mvenv))(env) ≙
   .4         PTCP([tve-seq(i).type | i ∈ inds tve-seq])(mt)(env) ∧
   .5         IsVenvSetMerge(π)({tve-seq(i).venv | i ∈ inds tve-seq})(mvenv)(env)
   .6       in
   .7   ∃ ctve-seq : TVE+ ·
   .8         len ctve-seq = len cp-seq ∧
   .9         ∀ i ∈ inds cp-seq ·
  .10               wf-Pattern(π)(cp-seq(i))(ctve-seq(i))(env) ∧
  .11         cases π :
  .12           mk-DEF({(UNIONCLOSE)} ∪ -) →
  .13                UnionClosePCP(PCP)(ctve-seq)(mtve)(env),
  .14           mk-DEF(-) → PCP(ctve-seq)(mtve)(env),
  .15           mk-POS(-) → ComplPCP(π)(PCP)(ctve-seq)(mtve)(env)
  .16         end
```

wf-CompPatn checks that the sequence of component patterns *cp-seq* from some compound pattern is well-formed with respect to the specified pattern type characteristic predicate *PTCP* and the match type and resulting value environment *mtve*.

The pattern type characteristic predicate defined locally for a compound pattern is (just as for expressions) defined so it only shows the basic properties of the considered pattern construct. Consequently, all pattern characteristic predicates are completed and relaxed to also hold for union types before they are used.

```
320.0 ComplPCP : Π →t PatCP → PatCP
   .1 ComplPCP (π)(PCP)(ctve-seq)(mk-TVE(t, venv))(env) ≙
   .2   let liftedtoPCP : TVE+ →t 𝔹
   .3       liftedtoPCP (mtves) ≙
   .4         len mtves = len ctve-seq ∧
   .5         ∀ i ∈ inds mtves · IsSubTVE(π)(ctve-seq(i), mtves(i))(env) ∧
   .6         ∃ mtve : TVE · PCP(mtves)(mtve)(env)
   .7       in
   .8   ∃ mtves : TVE+ ·
   .9         liftedtoPCP(mtves) ∧ PCP(mtves)(mk-TVE(t, venv))(env) ∧
  .10         ∀ omtves : TVE+ ·
  .11               liftedtoPCP(omtves) ⇒
  .12               ∀ i ∈ inds omtves ·
  .13                     IsSubTVE(π)(omtves(i), mtves(i))(env) ⇒
  .14                     IsSubTVE(π)(mtves(i), omtves(i))(env);

321.0 IsSubTVE : Π →t TVE × TVE → Env → 𝔹
   .1 IsSubTVE (π)(mk-TVE(t, venv), mk-TVE(mt, mvenv))(env) ≙
   .2   IsSubtypeR(π)(t, mt)(env) ∧ IsSubValEnv(π)(venv, mvenv)(env)
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.6.1.2 Relaxation of Pattern Characteristic Predicates

322.0 $UnionClosePCP : PatCP \xrightarrow{t} TVE^+ \to TVE \to Env \to \mathbb{B}$

.1 $UnionClosePCP\,(PCP)(ctveseq)(rtve)(env) \triangleq$
.2 $\quad PCP(ctveseq)(rtve)(env) \lor$
.3 $\quad \exists\, rtve_1, rtve_2 : TVE, ctveseq_1, ctveseq_2 : TVE^+ \cdot$
.4 $\qquad IsTveUnion(rtve_1, rtve_2)(rtve)(env) \land$
.5 $\qquad IsTveSeqUnion(ctveseq_1, ctveseq_2)(ctveseq)(env) \land$
.6 $\qquad UnionClosePCP(PCP)(ctveseq_1)(rtve_1)(env) \land$
.7 $\qquad UnionClosePCP(PCP)(ctveseq_2)(rtve_2)(env)$

UnionClosePCP relaxes pattern characteristic predicates so they may be able to accept match types which are union types.

323.0 $IsTveSeqUnion : TVE^+ \times TVE^+ \xrightarrow{t} TVE^+ \to Env \to \mathbb{B}$

.1 $IsTveSeqUnion\,(tveseq_1, tveseq_2)(tveseq)(env) \triangleq$
.2 $\quad \mathsf{len}\ tveseq_1 = \mathsf{len}\ tveseq \land \mathsf{len}\ tveseq_2 = \mathsf{len}\ tveseq \land$
.3 $\quad \forall\, i \in \mathsf{inds}\ tveseq \cdot$
.4 $\qquad IsTveUnion(tveseq_1(i), tveseq_2(i))(tveseq(i))(env);$

324.0 $IsTveUnion : TVE \times TVE \xrightarrow{t} TVE \to Env \to \mathbb{B}$

.1 $IsTveUnion\,(mk\text{-}TVE(t_1, ve_1), mk\text{-}TVE(t_2, ve_2))(mk\text{-}TVE(t, ve))(env) \triangleq$
.2 $\quad t = mk\text{-}UnionTypeR(\{t_1, t_2\}) \land IsVenvUnion(ve_1, ve_2)(ve)(env)$

14.6.1.3 Well-formedness of Patterns

325.0 $wf\text{-}PatternBasic : \Pi \xrightarrow{t} Pattern \to TVE \to Env \to \mathbb{B}$

.1 $wf\text{-}PatternBasic\,(\pi)(pat)(tve)(env) \triangleq$
.2 $\quad \mathsf{cases}\ pat :$
.3 $\qquad mk\text{-}PatternId(\text{-}) \to wf\text{-}PatternId(\pi)(pat)(tve)(env),$
.4 $\qquad mk\text{-}MatchVal(\text{-}) \to wf\text{-}MatchVal(\pi)(pat)(tve)(env),$
.5 $\qquad mk\text{-}SetEnumPattern(\text{-}) \to wf\text{-}SetEnumPattern(\pi)(pat)(tve)(env),$
.6 $\qquad mk\text{-}SetUnionPattern(\text{-}, \text{-}) \to wf\text{-}SetUnionPattern(\pi)(pat)(tve)(env),$
.7 $\qquad mk\text{-}SeqEnumPattern(\text{-}) \to wf\text{-}SeqEnumPattern(\pi)(pat)(tve)(env),$
.8 $\qquad mk\text{-}SeqConcPattern(\text{-}, \text{-}) \to wf\text{-}SeqConcPattern(\pi)(pat)(tve)(env),$
.9 $\qquad mk\text{-}TuplePattern(\text{-}) \to wf\text{-}TuplePattern(\pi)(pat)(tve)(env),$
.10 $\qquad mk\text{-}RecordPattern(\text{-}, \text{-}) \to wf\text{-}RecordPattern(\pi)(pat)(tve)(env)$
.11 $\quad \mathsf{end}$

wf-PatternBasic and the basic well-formedness predicates for the different pattern constructs defines the basic properties of patterns; there is no subsumption rule included in these basic well-formedness predicates. In the next formula the general pattern well-formedness predicate *wf-Pattern* is defined. This predicate includes a subsumption rule for possible well-formedness. *wf-Pattern* is the well-formedness predicate used when checking the well-formedness of patterns, both subcomponent patterns and patterns occurring in other constructs like expressions, statements and bindings.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

326.0 $wf\text{-}Pattern : \Pi \xrightarrow{t} Pattern \to TVE \to Env \to \mathbb{B}$

.1 $wf\text{-}Pattern\,(\pi)(pat)(tve)(env) \triangleq$
.2 cases π :
.3 $mk\text{-}DEF(\text{-}) \to wf\text{-}PatternBasic(\pi)(pat)(tve)(env)$,
.4 $mk\text{-}POS(\text{-}) \to$
.5 $\exists\, mtve : TVE \cdot$
.6 $wf\text{-}PatternBasic(\pi)(pat)(mtve)(env) \wedge$
.7 $SubsumeTVE(\pi)(mtve, tve)(env)$
.8 end;

327.0 $SubsumeTVE : \Pi \xrightarrow{t} TVE \times TVE \to Env \to \mathbb{B}$

.1 $SubsumeTVE\,(\pi)(mk\text{-}TVE(mt, mvenv), mk\text{-}TVE(t, venv))(env) \triangleq$
.2 $IsSubtypeR(\pi)(t, mt)(env) \wedge$
.3 $IsSubValEnv(\pi)(venv, mvenv)(env)$;

328.0 $IsSubValEnv : \Pi \xrightarrow{t} ValEnv \times ValEnv \to Env \to \mathbb{B}$

.1 $IsSubValEnv\,(\pi)(svenv, venv)(env) \triangleq$
.2 dom $svenv$ = dom $venv \wedge$
.3 $\forall\, id \in$ dom $svenv \cdot$
.4 $IsSubtypeR(\pi)(svenv(id).type, venv(id).type)(env)$;

329.0 $wf\text{-}PatternBind : \Pi \xrightarrow{t} PatternBind \to TVE \to Env \to \mathbb{B}$

.1 $wf\text{-}PatternBind\,(\pi)(patbind)(tve)(env) \triangleq$
.2 cases $patbind$:
.3 $mk\text{-}SetBind(\text{-}, \text{-}) \to wf\text{-}SetConstrPattern(\pi)(patbind)(tve)(env)$,
.4 $mk\text{-}TypeBind(\text{-}, \text{-}) \to wf\text{-}TypeConstrPattern(\pi)(patbind)(tve)(env)$,
.5 others $\to wf\text{-}Pattern(\pi)(patbind)(tve)(env)$
.6 end

The *PatternBind* construction is only used as a pattern, in spite of its name.

330.0 $wf\text{-}PatternId : \Pi \xrightarrow{t} PatternId \to TVE \to Env \to \mathbb{B}$

.1 $wf\text{-}PatternId\,(\pi)(mk\text{-}PatternId(id))(mk\text{-}TVE(t, ve))(env) \triangleq$
.2 cases id :
.3 (nil) $\to ve = EmptyValEnv$,
.4 $id \to ve = \{mk\text{-}Name(id) \mapsto mk\text{-}ValR(t, \pi)\}$
.5 end;

331.0 $wf\text{-}MatchVal : \Pi \xrightarrow{t} MatchVal \to TVE \to Env \to \mathbb{B}$

.1 $wf\text{-}MatchVal\,(\pi)(mk\text{-}MatchVal(e))(tve)(env) \triangleq$
.2 $tve.venv = EmptyValEnv \wedge$
.3 $\exists\, et : TypeR \cdot$
.4 $wf\text{-}Expr(\pi)(e)(et)(env) \wedge SameValue(\pi)(et, tve.type)(env)$;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
332.0 wf-SetEnumPattern : Π →t SetEnumPattern → TVE → Env → 𝔹
  .1  wf-SetEnumPattern (π)(mk-SetEnumPattern(pats))(tve)(env) ≜
  .2    let PTCP : TypeR+ →t TypeR → Env → 𝔹
  .3        PTCP (ct-seq)(t)(env) ≜
  .4          Shape(t) = mk-SetTypeR(mk-UnionTypeR({ct-seq(i) | i ∈ inds ct-seq})) ∧
  .5          CardofSet(π)(t, len ct-seq)(env)
  .6        in
  .7    wf-CompPatn(π)(PTCP)(pats)(tve)(env)
```

For the set enumeration pattern to be well-formed with respect to the match type t the match type must describe set values with a cardinality equal to the length of the pattern sequence (line 332.5). In order to express properties about the cardinality the match type must be some (type equivalent to an) invariant type, so *Shape* is used (in line 332.4).

```
333.0 wf-SetUnionPattern : Π →t SetUnionPattern → TVE → Env → 𝔹
  .1  wf-SetUnionPattern (π)(mk-SetUnionPattern(lp, rp))(tve)(env) ≜
  .2    let PTCP : TypeR+ →t TypeR → Env → 𝔹
  .3        PTCP ([t1, t2])(t)(env) ≜
  .4          (is-SetTypeR(t) ∨ t = EMPTYSET) ∧ t1 = t ∧ t2 = t
  .5        in
  .6    wf-CompPatn(π)(PTCP)([lp, rp])(tve)(env)
```

Notice that information about cardinality has been dropped, so component patterns making requirements about cardinality cannot be checked.

```
334.0 wf-SeqEnumPattern : Π →t SeqEnumPattern → TVE → Env → 𝔹
  .1  wf-SeqEnumPattern (π)(mk-SeqEnumPattern(pats))(tve)(env) ≜
  .2    let PTCP : TypeR+ →t TypeR → Env → 𝔹
  .3        PTCP (ct-seq)(t)(env) ≜
  .4          Shape(t) = mk-Seq1TypeR(mk-UnionTypeR({ct-seq(i) | i ∈ inds ct-seq})) ∧
  .5          LengthofSeq(π)(t, len ct-seq)(env)
  .6        in
  .7    wf-CompPatn(π)(PTCP)(pats)(tve)(env)
```

For the sequence enumeration pattern to be well-formed with respect to the match type t the match type must describe sequence values with a length equal to the length of the pattern sequence (line 334.5). Concerning the use of *Shape* see formula 332 .

```
335.0 wf-SeqConcPattern : Π →t SeqConcPattern → TVE → Env → 𝔹
  .1  wf-SeqConcPattern (π)(mk-SeqConcPattern(lp, rp))(tve)(env) ≜
  .2    let PTCP : TypeR+ →t TypeR → Env → 𝔹
  .3        PTCP ([t1, t2])(t)(env) ≜
  .4          cases {t1, t2} :
  .5            {mk-Seq0TypeR(et)} → t = mk-Seq0TypeR(et),
  .6            {mk-Seq0TypeR(et), mk-Seq1TypeR(et)} → t = mk-Seq1TypeR(et),
  .7            {mk-Seq1TypeR(et)} → t = mk-Seq1TypeR(et),
  .8            others → false
  .9          end
  .10       in
  .11   wf-CompPatn(π)(PTCP)([lp, rp])(tve)(env)
```

Specific information about sequence length that might be available has been dropped. Only the length information in the *Seq0TypeR* and the *Seq1TypeR* types is available. Considering the two component patterns (335.4-.9) if one of

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

the subcomponent patterns will not match the empty sequence, then the whole sequence concatenation pattern will not be able to match the empty sequence.

336.0 $wf\text{-}TuplePattern : \Pi \xrightarrow{t} TuplePattern \to TVE \to Env \to \mathbb{B}$
.1 $wf\text{-}TuplePattern\,(\pi)(mk\text{-}TuplePattern(fldpats))(tve)(env) \triangleq$
.2 let $PTCP : TypeR^{+} \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $PTCP\,(ft\text{-}seq)(t)(env) \triangleq$
.4 $t = mk\text{-}ProductTypeR([ft\text{-}seq(i) \mid i \in \mathsf{inds}\ ft\text{-}seq])$
.5 in
.6 $wf\text{-}CompPatn(\pi)(PTCP)(fldpats)(tve)(env)$

The type of a tuple pattern must be a *ProductTypeR*. Each of the factor types must be equal to the type of the corresponding component pattern of the tuple pattern.

337.0 $wf\text{-}RecordPattern : \Pi \xrightarrow{t} RecordPattern \to TVE \to Env \to \mathbb{B}$
.1 $wf\text{-}RecordPattern\,(\pi)(mk\text{-}RecordPattern(tag, fldpats))(tve)(env) \triangleq$
.2 let $PTCP : TypeR^{+} \xrightarrow{t} TypeR \to Env \to \mathbb{B}$
.3 $PTCP\,(ft\text{-}seq)(t)(env) \triangleq$
.4 $t = mk\text{-}CompositeTypeR(tag, [mk\text{-}FieldR(\mathsf{nil}, ft\text{-}seq(i)) \mid i \in \mathsf{inds}\ ft\text{-}seq])$
.5 in
.6 $wf\text{-}CompPatn(\pi)(PTCP)(fldpats)(tve)(env)$;

338.0 $wf\text{-}SetConstrPattern : \Pi \xrightarrow{t} SetBind \to TVE \to Env \to \mathbb{B}$
.1 $wf\text{-}SetConstrPattern\,(\pi)(mk\text{-}SetBind(pat, setexpr))(tve)(env) \triangleq$
.2 $wf\text{-}Pattern(\pi)(pat)(tve)(env) \wedge$
.3 $\exists\, elemtype : TypeR \cdot$
.4 $wf\text{-}Expr(\pi)(setexpr)(mk\text{-}SetTypeR(elemtype))(env) \wedge$
.5 $SameValue(\pi)(elemtype, tve.type)(env)$;

339.0 $wf\text{-}TypeConstrPattern : \Pi \xrightarrow{t} TypeBind \to TVE \to Env \to \mathbb{B}$
.1 $wf\text{-}TypeConstrPattern\,(\pi)(mk\text{-}TypeBind(pat, constrtype))(tve)(env) \triangleq$
.2 $wf\text{-}Pattern(\pi)(pat)(tve)(env) \wedge$
.3 $\exists\, cstrt : TypeR \cdot$
.4 $wf\text{-}Type(\pi)(constrtype)(cstrt)(env) \wedge IsSubtypeR(\pi)(tve.type, cstrt)(env)$

14.6.2 Bindings

The static semantics defines for each bind construct a well-formedness predicate (as usually indexed by the well-formedness classification):

$wf\text{-}Bind : \Pi \xrightarrow{t} Bind \to ValEnv \times BindAttrs \to Env \to \mathbb{B}$
$wf\text{-}Bind\,(\pi)(bind)(venv, bindattrs)(env) \triangleq$
body

where *ValEnv* is defined in 315 and *BindAttrs* is defined below:

340.0 $BindAttrs$ = (EMPTY | FINITE)-set

In the static semantics a single value environment $venv : ValEnv$ is used to describe the range of the set of dynamic value environments produced by a bind construct (including variations due to loosness). The number of bindings produced is described by some binding attributes $bindattrs : BindAttrs$.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
341.0 wf-Bind : Π -t-> Bind → ValEnv × BindAttrs → Env → 𝔹
  .1 wf-Bind (π)(bind)(venv, bindattrs)(env) ≙
  .2   cases bind :
  .3     mk-SetBind(-, -) → wf-SetBind(π)(bind)(venv, bindattrs)(env),
  .4     mk-TypeBind(-, -) → wf-TypeBind(π)(bind)(venv, bindattrs)(env)
  .5   end;

342.0 wf-SetBind : Π -t-> SetBind → ValEnv × BindAttrs → Env → 𝔹
  .1 wf-SetBind (π)(mk-SetBind(pat, setexpr))(venv, bindattrs)(env) ≙
  .2   let π‘ = NegWfClass(π),
  .3       SetMatch : Π -t-> TVE → 𝔹
  .4       SetMatch (π)(mk-TVE(elemt, venv)) ≙
  .5         ∃ sett : TypeR ·
  .6               wf-Expr(π)(setexpr)(sett)(env) ∧
  .7               wf-Pattern(π)(pat)(mk-TVE(elemt, venv))(env) ∧
  .8               IsNonEmptySet(π)(sett)(env) ∧
  .9               Shape(sett) = mk-SetTypeR(elemt)
  .10      in
  .11  cases π :
  .12    mk-DEF(-) →
  .13         (∃ et : TypeR · wf-Expr(π)(setexpr)(mk-SetTypeR(et))(env)) ∧
  .14         ∃ tve : TVE · wf-Pattern(π)(pat)(tve)(env) ∧
  .15         ∀ ptve : TVE ·
  .16              SetMatch(π‘)(ptve) ⇒ IsSubValEnv(π)(ptve.venv, venv)(env) ∧
  .17         (EMPTY ∈ bindattrs ⇒ ¬ ∃ ptve : TVE · SetMatch(π‘)(ptve)),
  .18    mk-POS(-) →
  .19         (∃ sett, et : TypeR ·
  .20               wf-Expr(π)(setexpr)(sett)(env) ∧
  .21               Shape(sett) = mk-SetTypeR(et) ∧
  .22               (wf-Pattern(π)(pat)(mk-TVE(et, venv))(env) ∨
  .23                venv = { ↦}))
  .24          ∧
  .25         (¬ ∃ dtve : TVE · SetMatch(π‘)(dtve) ⇔ EMPTY ∈ bindattrs) ∧
  .26         FINITE ∈ bindattrs
  .27  end;

343.0 IsNonEmptySet : Π -t-> TypeR → Env → 𝔹
  .1 IsNonEmptySet (π)(sett)(env) ≙
  .2   is-POS(π)
```

According to the Dynamic Semantics the setbind *mk-SetBind*(*pat*, *setexpr*) as such is well-formed if the *setexpr* evaluates to a set and the pattern *pat* does not generate the *err* value. It is not required that any of the set elements match the pattern. When checking constructs in which the setbind is used, the Static Semantics must have some information about the number of generated value environments and the form of the generated value environments; this information is available in the *bindattrs* respectively the *venv* parameter. The membership of EMPTY and FINITE in *bindattrs* contains information about whether the set of generated dynamic value environments is empty or finite.

For definite well-formedness it is checked that *setexpr* is definitely well-formed for some set-type (342.13) and that the pattern is definitly well-formed (342.14); the check ensures that the pattern will not generate the *err* value.

For a setbind to be definitely well-formed with respect to the value environment *venv* all generated dynamic value environments must conform with *venv*. To make this check the auxiliary function *SetMatch* is used. For definite well-formedness it checks for a *mk-TVE*(*elemt*, *venv*) that the *setexpr* is *possibly* well-formed for a set-type having *elemt* as the element type and that the pattern is *possibly* well-formed with the given tve, i.e. if the *setexpr* possibly evaluates

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

to a value which possibly matches the pattern. The Static Semantics now requires that for all *mk-TVE*(*pelemt*, *pvenv*) (342.15-.16) for which *SetMatch* is true, must *pvenv* be a sub value environment of *venv*. In terms of the Dynamic Semantics: every possibly generated dynamic value environment must conform with *venv*.

Finally, for EMPTY to be a member of *bindattrs* we must be sure that no dynamic value environment is generated; this is so if no possible element value from *setexpr* possibly matches the pattern (342.17). Concerning FINITE the Static Semantics does not say anything, but it is safe to have FINITE as a member of *bindattrs*, because a *setexpr* always evaluates to a finite set

Next consider possible well-formedness. It is checked that the *setexpr* possibly evaluates to a set value (342.20-.21) and that the pattern is possibly well-formed (342.22-.23). If the pattern is not possibly well-formed it is certain that the pattern will generate the *err* value. There are two cases for the pattern well-formedness. Line 342.22 is the case where the pattern possibly matches an element value from the setexpr. Line 342.23 is the case where the pattern does not match.

If the pattern definitly does not match an element value from the *setexpr* then EMPTY must be a member of *bindattrs* (342.25). Consequently, if EMPTY is not a member of *bindattrs* it is certain that some value from the *setexpr* will match the pattern. Finally, FINITE is always a member of *bindattrs* (342.26), so one cannot conclude that the set of generated value environments is not finite.

```
344.0 wf-TypeBind : Π -t-> TypeBind → ValEnv × BindAttrs → Env → 𝔹
   .1 wf-TypeBind (π)(mk-TypeBind(pat, type))(venv, bindattrs)(env) ≙
   .2   let π‘ = NegWfClass(π),
   .3       TypeMatch : Π → TVE → TypeR → 𝔹
   .4       TypeMatch (π)(mk-TVE(stp, venv))(t) ≙
   .5         IsSubtypeR(π)(stp, t)(env) ∧
   .6         wf-Pattern(π)(pat)(mk-TVE(stp, venv))(env)
   .7       in
   .8   ∃ t : TypeR ·
   .9         wf-Type(π)(type)(t)(env) ∧
   .10        cases π :
   .11          mk-DEF(-) →
   .12               (∃ tve : TVE · wf-Pattern(π)(pat)(tve)(env)) ∧
   .13               ∀ ptve : TVE ·
   .14                    TypeMatch(π‘)(ptve)(t) ⇒ IsSubValEnv(π)(ptve.venv, venv)(env) ∧
   .15               (EMPTY ∈ bindattrs ⇒ ¬ ∃ ptve : TVE · TypeMatch(π‘)(ptve)(t)) ∧
   .16               (FINITE ∈ bindattrs ⇒ IsFiniteTypeR(π)(t)(env)),
   .17          mk-POS(-) →
   .18               ((∃ stp : TypeR · TypeMatch(π)(mk-TVE(stp, venv))(t)) ∨
   .19                ∃ ptve : TVE · wf-Pattern(π)(pat)(ptve)(env) ∧ venv = { ↦})
   .20                ∧
   .21               (¬ ∃ dtve : TVE · TypeMatch(π‘)(dtve)(t) ⇒ EMPTY ∈ bindattrs) ∧
   .22               (IsFiniteTypeR(π)(t)(env) ⇒ FINITE ∈ bindattrs)
   .23       end
```

The well-formedness predicate for a type bind is very similar to the well-formedness predicate for a set bind. In a type bind the values to be matched with the pattern belong to the set of values for the given type. That set may be finite or infinite. The predicate *IsFiniteTypeR* checks whether the type is finite or not.

```
345.0 wf-BindList : Π -t-> BindList → ValEnv × BindAttrs → Env → 𝔹
   .1 wf-BindList (π)(multbindlist)(venv, bindattrs)(env) ≙
   .2   ∃ venv-battrs-s : (ValEnv × BindAttrs)* ·
   .3         len venv-battrs-s = len multbindlist ∧
   .4         IsBindSetMerge(π)(elems venv-battrs-s)(venv, bindattrs)(env) ∧
   .5         ∀ i ∈ inds multbindlist ·
   .6              wf-MultBind(π)(multbindlist(i))(venv-battrs-s(i))(env);
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

346.0 $wf\text{-}MultBind : \Pi \xrightarrow{t} MultBind \to ValEnv \times BindAttrs \to Env \to \mathbb{B}$
.1 $wf\text{-}MultBind\,(\pi)(multbind)(venv, bindattrs)(env) \triangleq$
.2 cases $multbind$:
.3 $mk\text{-}MultSetBind(\text{-}, \text{-}) \to wf\text{-}MultSetBind(\pi)(multbind)(venv, bindattrs)(env)$,
.4 $mk\text{-}MultTypeBind(\text{-}, \text{-}) \to wf\text{-}MultTypeBind(\pi)(multbind)(venv, bindattrs)(env)$
.5 end;

347.0 $wf\text{-}MultSetBind : \Pi \xrightarrow{t} MultSetBind \to ValEnv \times BindAttrs \to Env \to \mathbb{B}$
.1 $wf\text{-}MultSetBind\,(\pi)(mk\text{-}MultSetBind(pats, setexpr))(venv, bindattrs)(env) \triangleq$
.2 $\exists\, venv\text{-}battrs\text{-}s : (ValEnv \times BindAttrs)^* \cdot$
.3 len $venv\text{-}battrs\text{-}s$ = len $pats \wedge$
.4 $IsBindSetMerge(\pi)(\text{elems } venv\text{-}battrs\text{-}s)(venv, bindattrs)(env) \wedge$
.5 $\forall\, i \in \text{inds } pats \cdot$
.6 $wf\text{-}SetBind(\pi)(mk\text{-}SetBind(pats(i), setexpr))(venv\text{-}battrs\text{-}s(i))(env)$;

348.0 $wf\text{-}MultTypeBind : \Pi \xrightarrow{t} MultTypeBind \to ValEnv \times BindAttrs \to Env \to \mathbb{B}$
.1 $wf\text{-}MultTypeBind\,(\pi)(mk\text{-}MultTypeBind(pats, type))(venv, bindattrs)(env) \triangleq$
.2 $\exists\, venv\text{-}battrs\text{-}s : (ValEnv \times BindAttrs)^* \cdot$
.3 len $venv\text{-}battrs\text{-}s$ = len $pats \wedge$
.4 $IsBindSetMerge(\pi)(\text{elems } venv\text{-}battrs\text{-}s)(venv, bindattrs)(env) \wedge$
.5 $\forall\, i \in \text{inds } pats \cdot$
.6 $wf\text{-}TypeBind(\pi)(mk\text{-}TypeBind(pats(i), type))(venv\text{-}battrs\text{-}s(i))(env)$

14.6.3 Value Environment

In formula 315 *ValEnv* was defined as

$$ValEnv = Name \xrightarrow{m} ValR$$

ValEnv (which is a subtype of *VisibleEnv*) is the type of value environments.

349.0 $EmptyValEnv : ValEnv = \{ \mapsto \}$

350.0 $IsVenvUnion : ValEnv \times ValEnv \xrightarrow{t} ValEnv \to Env \to \mathbb{B}$
.1 $IsVenvUnion\,(ve_1, ve_2)(ve)(env) \triangleq$
.2 $(\forall\, id \in \text{dom } ve_1 \cap \text{dom } ve_2 \cdot$
.3 $ve(id).type = mk\text{-}UnionTypeR(\{ve_1(id).type, ve_2(id).type\})) \wedge$
.4 $\forall\, id \in \text{dom } ve_1 \setminus \text{dom } ve_2 \cdot ve(id) = ve_1(id) \wedge$
.5 $\forall\, id \in \text{dom } ve_2 \setminus \text{dom } ve_1 \cdot ve(id) = ve_2(id)$

IsVenvUnion checks that *ve* is the *union* of two value environments (ve_1, ve_2), which potentially may have non-disjoint domains. In the *union* of the two value environments, names which occur in the domain of both value environments are mapped to the union of the corresponding types in the two value environments.

351.0 $IsVenvMerge : \Pi \xrightarrow{t} ValEnv \times ValEnv \to ValEnv \to Env \to \mathbb{B}$
.1 $IsVenvMerge\,(\pi)(ve_1, ve_2)(ve)(env) \triangleq$
.2 dom ve = dom $ve_1 \cup$ dom $ve_2 \wedge$
.3 $\forall\, id \in \text{dom } ve_1 \cap \text{dom } ve_2 \cdot$
.4 let $mk\text{-}(t_1, t_2, t) = mk\text{-}(ve_1(id).type, ve_2(id).type, ve(id).type)$ in
.5 $SameValue(\pi)(t_1, t_2)(env) \wedge$
.6 $IsIntersectionType(\pi)(t_1, t_2)(t)(env) \wedge$
.7 $\forall\, id \in \text{dom } ve_1 \setminus \text{dom } ve_2 \cdot ve(id) = ve_1(id) \wedge$
.8 $\forall\, id \in \text{dom } ve_2 \setminus \text{dom } ve_1 \cdot ve(id) = ve_2(id)$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

IsVenvMerge$(\pi)(ve_1, ve_2)(ve)(env)$ is true if ve is a *merge* of the two value environments ve_1, ve_2, which potentially may have non disjoint domains.

352.0 $IsVenvSetMerge : \Pi \xrightarrow{t} ValEnv\text{-set} \to ValEnv \to Env \to \mathbb{B}$
.1 $IsVenvSetMerge\,(\pi)(ve\text{-}set)(ve)(env) \triangleq$
.2 cases $ve\text{-}set$:
.3 $(\{\}) \to ve = EmptyValEnv$,
.4 $\{ve_1\} \to ve = ve_1$,
.5 $ves_1 \cup ves_2 \to$
.6 $\exists\, ve_1, ve_2 : ValEnv \cdot$
.7 $IsVenvSetMerge(\pi)(ves_1)(ve_1)(env) \wedge$
.8 $IsVenvSetMerge(\pi)(ves_2)(ve_2)(env) \wedge$
.9 $IsVenvMerge(\pi)(ve_1, ve_2)(ve)(env)$
.10 end;

353.0 $IsBindSetMerge : \Pi \xrightarrow{t} (ValEnv \times BindAttrs)\text{-set} \to ValEnv \times BindAttrs \to Env \to \mathbb{B}$
.1 $IsBindSetMerge\,(\pi)(ve\text{-}bats\text{-}set)(ve, bats)(env) \triangleq$
.2 let $ve\text{-}set = \{ve \mid mk\text{-}(ve, \text{-}) \in ve\text{-}bats\text{-}set\}$,
.3 $bats\text{-}set = \{bats \mid mk\text{-}(\text{-}, bats) \in ve\text{-}bats\text{-}set\}$ in
.4 $IsVenvSetMerge(\pi)(ve\text{-}set)(ve)(env) \wedge$
.5 $(\text{EMPTY} \in \bigcup bats\text{-}set \Leftrightarrow \text{EMPTY} \in bats) \wedge$
.6 $(\text{FINITE} \in \bigcap bats\text{-}set \Leftrightarrow \text{FINITE} \in bats)$

14.7 Auxiliary Functions

14.7.1 Dependency Relations

354.0 $DefDepRel = (ValFnOpTpDef \times ValFnOpTpDef)\text{-set}$;

355.0 $TypeRDepRel = (TypeR \times TypeR)\text{-set}$

356.0 $ValFnOpTpTransDefDep : ValFnOpDefs \xrightarrow{t} DefDepRel$
.1 $ValFnOpTpTransDefDep\,(ds) \triangleq$
.2 $TransClosure[ValFnOpTpDef](ValFnOpTpDefDep(ds))$;

357.0 $ValFnOpTpDefDep : ValFnOpTpDefs \xrightarrow{t} DefDepRel$
.1 $ValFnOpTpDefDep\,(ds) \triangleq$
.2 $\{mk\text{-}(x, y) \mid x, y \in \text{elems } ds \cdot$
.3 $FreeNamesInDef(x) \cap DefNamesInDef(y) \neq \{\}\}$;

358.0 $TransTypeRDep : \Pi \xrightarrow{t} TypeR \to Env \to TypeRDepRel$
.1 $TransTypeRDep\,(\pi)(type)(env) \triangleq$
.2 $TransClosure[TypeR](TypeRDep(\pi)(type)(env))$;

359.0 $TypeRDep : \Pi \xrightarrow{t} TypeR \to Env \to TypeRDepRel$
.1 $TypeRDep\,(\pi)(type)(env) \triangleq$
.2 $TypeRSubExprDep(\pi)(type) \cup$
.3 $\{mk\text{-}(x, env.typemap(x)) \mid x \in \text{dom } env.typemap\} \cup$
.4 $\bigcup \{TypeRSubExprDep(\pi)(t) \mid t \in \text{rng } env.typemap\}$;

360.0 $TypeRSubExprDep : \Pi \xrightarrow{t} TypeR \to TypeRDepRel$
.1 $TypeRSubExprDep\,(\pi)(x) \triangleq$
.2 $\{mk\text{-}(x, y) \mid y \in TypeRDefBasis(\pi)(x) \setminus \{x\}\}$;

361.0 $TransClosure[@T]\,(rel : (@T \times @T)\text{-set})\ res : (@T \times @T)\text{-set}$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.1 post $rel \subseteq res \wedge$
.2 $\quad IsTransitivelyClosed[@T](res) \wedge$
.3 $\quad \forall\, r : (@T \times @T)\text{-set} \cdot$
.4 $\quad\quad rel \subseteq r \wedge r \subset res \;\Rightarrow\; \neg\, IsTransitivelyClosed[@T](r)$;

362.0 $IsTransitivelyClosed[@T] : (@T \times @T)\text{-set} \xrightarrow{t} \mathbb{B}$

.1 $IsTransitivelyClosed\,(rel) \triangleq$
.2 $\quad \forall\, mk\text{-}(x, y), mk\text{-}(z, v) \in rel \cdot$
.3 $\quad\quad y = z \;\Rightarrow\; mk\text{-}(x, v) \in rel;$

363.0 $ReflExtension[@T] : (@T \times @T)\text{-set} \xrightarrow{t} (@T \times @T)\text{-set}$

.1 $ReflExtension\,(rel) \triangleq$
.2 $\quad rel \cup$
.3 $\quad \{mk\text{-}(x, x) \mid x : @T \cdot \exists\, y : @T \cdot$
.4 $\quad\quad\quad mk\text{-}(x, y) \in rel \vee mk\text{-}(y, x) \in rel\};$

364.0 $IsPermutationOf[@T] : @T^* \times @T^* \xrightarrow{t} \mathbb{B}$

.1 $IsPermutationOf\,(s_1, s_2) \triangleq$
.2 $\quad \exists\, m : \mathbb{N}_1 \xleftrightarrow{m} \mathbb{N}_1 \cdot$
.3 $\quad\quad \mathsf{card}\,\{\mathsf{dom}\ m, \mathsf{rng}\ m, \mathsf{inds}\ s_1, \mathsf{inds}\ s_2\} = 1 \wedge$
.4 $\quad\quad \forall\, i \in \mathsf{inds}\ s_1 \cdot s_1(i) = s_2(m(i))$

14.7.1.1 Free and Defined Names

365.0 $FreeNamesInDefs : ValFnOpTpDefs \xrightarrow{t} Name\text{-set}$

.1 $FreeNamesInDefs\,(ds) \triangleq$
.2 $\quad \bigcup \{FreeNamesInDef(d) \mid d \in \mathsf{elems}\ ds\} \setminus DefNamesInDefs(ds);$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
366.0 FreeNamesInDef : ValFnOpTpDef →t Name-set
  .1 FreeNamesInDef (d) ≜
  .2   cases d :
  .3     mk-ValueDef(pat, -, val) →
  .4         FreeNamesInPat(pat) ∪ FreeNamesInExpr(val),
  .5     mk-ExplFnDef(-, -, -, -, parmslist, body, fpre) →
  .6         let bnames = FreeNamesInExpr(body),
  .7             pnames = if fpre = nil then {} else FreeNamesInExpr(fpre),
  .8             dnames = DefNamesInPats(conc parmslist) in
  .9         bnames ∪ pnames \ dnames
  .10        ∪ FreeNamesInPats(conc parmslist),
  .11    mk-ImplFnDef(-, -, partps, mk-IdType(ri, -), fpre, fpost) →
  .12        let ponames = FreeNamesInExpr(fpost),
  .13            prnames = if fpre = nil then {} else FreeNamesInExpr(fpre),
  .14            dnames = DefNamesInParTypes(partps) ∪ {mk-Name(ri)} in
  .15        ponames ∪ prnames \ dnames
  .16        ∪ FreeNamesInParTypes(partps),
  .17    mk-ExplOprtDef(-, -, -, parms, body, opre) →
  .18        let bnames = FreeNamesInStmt(body),
  .19            pnames = if opre = nil then {} else FreeNamesInExpr(opre),
  .20            dnames = DefNamesInPats(parms) in
  .21        bnames ∪ pnames \ dnames
  .22        ∪ FreeNamesInPats(parms),
  .23    mk-ImplOprtDef(-, partps, ritp, oexts, opre, opost, oexcp) →
  .24        let ponames = FreeNamesInExpr(opost),
  .25            prnames = if opre = nil then {} else FreeNamesInExpr(opre),
  .26            enames = FreeNamesInExceptions(oexcp),
  .27            etnames = FreeNamesInExternals(oexts),
  .28            rnames = if ritp = nil then {} else {mk-Name(ritp.id)},
  .29            dnames = DefNamesInParTypes(partps) ∪ rnames in
  .30        ponames ∪ prnames ∪ enames ∪ etnames \ dnames
  .31        ∪ FreeNamesInParTypes(partps),
  .32    others → let {t} = rng ExtractTypeMap([d]) in
  .33             FreeNamesInTypeR(t)
  .34  end;

367.0 FreeNamesInType : [Type] →t Name-set
  .1 FreeNamesInType (t1) ≜
  .2   if t1 = nil then {} else FreeNamesInTypeR(ExtractTypeR(t1)(EmptyEnv));

368.0 FreeNamesInTypeR : TypeR →t Name-set
  .1 FreeNamesInTypeR (t1) ≜
  .2   {n | n : Name ·
  .3         ∃ t2 : TypeR ·
  .4             IsTypeRSubExpr(t2, t1) ∧
  .5             is-Name(t2) ∧ n = t2 ∨
  .6             is-InvTypeR(t2) ∧ n ∈ FreeNamesInInvInitFn(t2.invariant)};

369.0 FreeNamesInInvInitFn : InvInitFn →t Name-set
  .1 FreeNamesInInvInitFn (mk-InvInitFn(pat, expr)) ≜
  .2   FreeNamesInExpr(expr) \ DefNamesInPat(pat) ∪ FreeNamesInPat(pat);

370.0 FreeNamesInExprs : [Expr]* →t Name-set
  .1 FreeNamesInExprs (es) ≜
  .2   ⋃{FreeNamesInExpr(e) | e ∈ elems es};
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
371.0 FreeNamesInExpr : [Expr] -t-> Name-set
  .1  FreeNamesInExpr (e) ≜
  .2    cases e :
  .3      mk-BracketedExpr(e1) → FreeNamesInExpr(e1),
  .4      mk-DefExpr(defs, body) →
  .5          FreeNamesInExpr(body) \ DefNamesInDefBinds(defs) ∪
  .6          FreeNamesInDefBinds(defs),
  .7      mk-LetExpr(ldefs, body) →
  .8          FreeNamesInExpr(body) \ DefNamesInDefs(ldefs) ∪
  .9          FreeNamesInDefs(ldefs),
  .10     mk-LetBeSTExpr(bind, cond, body) →
  .11         FreeNamesInExpr(body) ∪ FreeNamesInExpr(cond) \ DefNamesInBind(bind)
  .12         ∪ FreeNamesInBind(bind),
  .13     mk-IfExpr(test, cons, altns, altn) →
  .14         FreeNamesInExpr(test) ∪ FreeNamesInExpr(cons) ∪
  .15         FreeNamesInExpr(altn) ∪
  .16         ⋃{FreeNamesInExpr(t) ∪ FreeNamesInExpr(c) |
  .17             mk-ElsifExpr(t, c) ∈ elems altns},
  .18     mk-CasesExpr(sel, altns, other) →
  .19         FreeNamesInExpr(sel) ∪ FreeNamesInExpr(other) ∪
  .20         ⋃{FreeNamesInExpr(body) \ DefNamesInPats(ps) ∪ FreeNamesInPats(ps) |
  .21             mk-CaseAltn(ps, body) ∈ elems altns},
  .22     mk-PrefixExpr(-, e1) → FreeNamesInExpr(e1),
  .23     mk-BinaryExpr(left, -, right) → FreeNamesInExpr(left) ∪ FreeNamesInExpr(right),
  .24     others → FreeNamesInSpecExpr(e)
  .25   end;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
372.0 FreeNamesInSpecExpr : [Expr] -t-> Name-set
  .1 FreeNamesInSpecExpr (e) ≜
  .2   cases e :
  .3     mk-MapInverseExpr(e1) → FreeNamesInExpr(e1),
  .4     mk-ExistsExpr(bs, p), mk-ExistsUniqueExpr(bs, p) →
  .5         FreeNamesInExpr(p) \ DefNamesInBind(bs) ∪
  .6         FreeNamesInBind(bs),
  .7     mk-AllExpr(bs, p), mk-IotaExpr(bs, p) →
  .8         FreeNamesInExpr(p) \ DefNamesInBind(bs) ∪
  .9         FreeNamesInBind(bs),
  .10    mk-SetEnumeration(es) → FreeNamesInExprs(es),
  .11    mk-SetComprehension(e, bs, p) →
  .12        FreeNamesInExpr(e) ∪ FreeNamesInExpr(p) \ DefNamesInBind(bs)
  .13        ∪ FreeNamesInBind(bs),
  .14    mk-SetRange(lb, ub) → FreeNamesInExpr(lb) ∪ FreeNamesInExpr(ub),
  .15    mk-SeqEnumeration(es) → FreeNamesInExprs(es),
  .16    mk-SeqComprehension(e, b, p) →
  .17        FreeNamesInExpr(e) ∪ FreeNamesInExpr(p) \ DefNamesInBind(b)
  .18        ∪ FreeNamesInBind(b),
  .19    mk-SubSequence(s, lb, ub) → FreeNamesInExpr(lb) ∪ FreeNamesInExpr(ub)
  .20                                ∪ FreeNamesInExpr(s),
  .21    mk-MapEnumeration(mls) →
  .22        ⋃{FreeNamesInExpr(d) ∪ FreeNamesInExpr(r) |
  .23            mk-Maplet(d, r) ∈ elems mls},
  .24    mk-MapComprehension(mk-Maplet(d, r), bs, p) →
  .25        FreeNamesInExpr(d) ∪ FreeNamesInExpr(r) ∪ FreeNamesInExpr(p)
  .26        \ DefNamesInBind(bs)
  .27        ∪ FreeNamesInBind(bs),
  .28    mk-TupleConstructor(es) → FreeNamesInExprs(es),
  .29    mk-RecordConstructor(-, es) → FreeNamesInExprs(es),
  .30    mk-RecordModifier(rec, rms) →
  .31        FreeNamesInExpr(rec) ∪
  .32        ⋃{FreeNamesInExpr(new) |
  .33            mk-RecordModification(-, new) ∈ elems rms},
  .34    mk-Apply(e1, es) → FreeNamesInExpr(e1) ∪ FreeNamesInExprs(es),
  .35    mk-FieldSelect(e1, -) → FreeNamesInExpr(e1),
  .36    mk-Lambda(tbs, body) → FreeNamesInExpr(body) \ DefNamesInBind(tbs)
  .37                           ∪ FreeNamesInBind(tbs),
  .38    mk-IsDefTypeExpr(-, e1) → FreeNamesInExpr(e1),
  .39    mk-IsBasicTypeExpr(-, e1) → FreeNamesInExpr(e1),
  .40    others → if is-Name(e) then {e} else {}
  .41  end;

373.0 FreeNamesInDefBinds : DefBind* -t-> Name-set
  .1 FreeNamesInDefBinds (ds) ≜
  .2   if ds = [] then {}
  .3   else FreeNamesInDefBind(hd ds) ∪
  .4        (FreeNamesInDefBinds(tl ds) \ DefNamesInDefBind(hd ds));

374.0 FreeNamesInDefBind : DefBind -t-> Name-set
  .1 FreeNamesInDefBind (mk-DefBind(pb, rhs)) ≜
  .2   FreeNamesInPatBind(pb) ∪ FreeNamesInExpr(rhs);

375.0 FreeNamesInPatBind : PatternBind -t-> Name-set
  .1 FreeNamesInPatBind (pb) ≜
  .2   if IsPattern(pb) then FreeNamesInPat(pb) else FreeNamesInBind(pb);
```

--`,,```,,,,````-`-`,,`,,`,`,,`---

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

376.0 $FreeNamesInBind : Bind \mid MultBind \mid BindList \mid TypeBindList \xrightarrow{t} Name\text{-set}$

.1 $FreeNamesInBind\,(b) \triangleq$
.2 cases b :
.3 $mk\text{-}SetBind(pat, bset) \rightarrow FreeNamesInPat(pat) \cup FreeNamesInExpr(bset)$,
.4 $mk\text{-}TypeBind(pat, type) \rightarrow FreeNamesInPat(pat) \cup FreeNamesInType(type)$,
.5 $mk\text{-}MultSetBind(pats, bset) \rightarrow FreeNamesInPats(pats) \cup FreeNamesInExpr(bset)$,
.6 $mk\text{-}MultTypeBind(pats, type) \rightarrow FreeNamesInPats(pats) \cup FreeNamesInType(type)$,
.7 others $\rightarrow \bigcup \{FreeNamesInBind(b') \mid b' \in \text{elems } b\}$
.8 end;

377.0 $FreeNamesInPats : Pattern^* \xrightarrow{t} Name\text{-set}$

.1 $FreeNamesInPats\,(ps) \triangleq$
.2 $\bigcup \{FreeNamesInPat(p) \mid p \in \text{elems } ps\}$;

378.0 $FreeNamesInPat : Pattern \xrightarrow{t} Name\text{-set}$

.1 $FreeNamesInPat\,(p) \triangleq$
.2 cases p :
.3 $mk\text{-}MatchVal(e) \rightarrow FreeNamesInExpr(e)$,
.4 $mk\text{-}SetEnumPattern(ps) \rightarrow FreeNamesInPats(ps)$,
.5 $mk\text{-}SetUnionPattern(lp, rp) \rightarrow FreeNamesInPats([lp, rp])$,
.6 $mk\text{-}SeqEnumPattern(ps) \rightarrow FreeNamesInPats(ps)$,
.7 $mk\text{-}SeqConcPattern(lp, rp) \rightarrow FreeNamesInPats([lp, rp])$,
.8 $mk\text{-}TuplePattern(ps) \rightarrow FreeNamesInPats(ps)$,
.9 $mk\text{-}RecordPattern(\text{-}, ps) \rightarrow FreeNamesInPats(ps)$,
.10 $mk\text{-}PatternId(n) \rightarrow \{\}$
.11 end;

379.0 $FreeNamesInStmts : Stmt^* \xrightarrow{t} Name\text{-set}$

.1 $FreeNamesInStmts\,(ss) \triangleq$
.2 $\bigcup \{FreeNamesInStmt(s) \mid s \in \text{elems } ss\}$;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
380.0 FreeNamesInStmt : Stmt -t-> Name-set
  .1  FreeNamesInStmt (s) ≜
  .2    cases s :
  .3      mk-BlockStmt(decls, stmts) →
  .4          FreeNamesInStmts(stmts) \ DefNamesInAssignDefs(decls) ∪
  .5          FreeNamesInAssignDefs(decls),
  .6      mk-AlwaysStmt(alwpost, body) →
  .7          FreeNamesInStmt(alwpost) ∪ FreeNamesInStmt(body),
  .8      mk-TrapStmt(pat, trappost, body) →
  .9          FreeNamesInStmt(trappost) ∪ FreeNamesInStmt(body) ∪
  .10         FreeNamesInPat(pat),
  .11     mk-RecTrapStmt(traps, body) →
  .12         FreeNamesInStmt(body) ∪
  .13         ⋃{FreeNamesInPatBind(match) ∪ FreeNamesInStmt(trappost) |
  .14             mk-Trap(match, trappost) ∈ elems traps},
  .15     mk-DefStmt(eqdefs, body) →
  .16         FreeNamesInStmt(body) \ DefNamesInEqDefs(eqdefs) ∪
  .17         FreeNamesInEqDefs(eqdefs),
  .18     mk-LetStmt(letdefs, body) →
  .19         FreeNamesInStmt(body) \ DefNamesInDefs(letdefs) ∪
  .20         FreeNamesInDefs(letdefs),
  .21     mk-LetBeSTStmt(bind, cond, body) →
  .22         FreeNamesInStmt(body) ∪ FreeNamesInExpr(cond)
  .23         \ DefNamesInBind(bind)
  .24         ∪ FreeNamesInBind(bind),
  .25     mk-AssignStmt(lhs, rhs) →
  .26         FreeNamesInStateDesignator(lhs) ∪ FreeNamesInExprOrCall(rhs),
  .27     mk-NonDetStmt(stmts) → FreeNamesInStmts(stmts),
  .28     mk-SeqForLoop(cv, -, forseq, body) →
  .29         FreeNamesInStmt(body) \ DefNamesInPatBind(cv) ∪
  .30         FreeNamesInExpr(forseq) ∪ FreeNamesInPatBind(cv),
  .31     mk-SetForLoop(cv, forset, body) →
  .32         FreeNamesInStmt(body) \ DefNamesInPat(cv) ∪
  .33         FreeNamesInExpr(forset) ∪ FreeNamesInPat(cv),
  .34     mk-IndexForLoop(cv, lb, ub, step, body) →
  .35         FreeNamesInStmt(body) \ {mk-Name(cv)} ∪
  .36         FreeNamesInExpr(lb) ∪ FreeNamesInExpr(ub) ∪
  .37         FreeNamesInExpr(step),
  .38     mk-WhileLoop(test, body) → FreeNamesInExpr(test) ∪ FreeNamesInStmt(body),
  .39     mk-IfStmt(test, cons, altns, altn) →
  .40         FreeNamesInExpr(test) ∪ FreeNamesInStmt(cons) ∪
  .41         FreeNamesInStmt(altn) ∪
  .42         ⋃{FreeNamesInExpr(t) ∪ FreeNamesInStmt(c) |
  .43             mk-ElsifStmt(t, c) ∈ elems altns},
  .44     mk-CasesStmt(sel, altns, other) →
  .45         FreeNamesInExpr(sel) ∪ FreeNamesInStmt(other) ∪
  .46         ⋃{FreeNamesInStmt(body) \ DefNamesInPats(ps) ∪ FreeNamesInPats(ps) |
  .47             mk-CaseStmtAltn(ps, body) ∈ elems altns},
  .48     mk-ReturnStmt(-) → {},
  .49     mk-ExitStmt(-) → {}
  .50   end;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
381.0 FreeNamesInAssignDefs : AssignDef* -t-> Name-set
  .1 FreeNamesInAssignDefs (ads) ≜
  .2   if ads = [] then {}
  .3   else FreeNamesInAssignDef(hd ads) ∪
  .4        (FreeNamesInAssignDefs(tl ads) \ DefNamesInAssignDef(hd ads));

382.0 FreeNamesInAssignDef : AssignDef -t-> Name-set
  .1 FreeNamesInAssignDef (mk-AssignDef(-, tp, dcli)) ≜
  .2   FreeNamesInType(tp) ∪ FreeNamesInExprOrCall(dcli);

383.0 FreeNamesInEqDefs : EqDef* -t-> Name-set
  .1 FreeNamesInEqDefs (eqds) ≜
  .2   if eqds = [] then {}
  .3   else FreeNamesInEqDef(hd eqds) ∪
  .4        (FreeNamesInEqDefs(tl eqds) \ DefNamesInEqDef(hd eqds));

384.0 FreeNamesInEqDef : EqDef -t-> Name-set
  .1 FreeNamesInEqDef (mk-EqDef(lhs, rhs)) ≜
  .2   FreeNamesInPatBind(lhs) ∪ FreeNamesInExprOrCall(rhs);

385.0 FreeNamesInStateDesignator : StateDesignator -t-> Name-set
  .1 FreeNamesInStateDesignator (std) ≜
  .2   cases std :
  .3     mk-FieldRef(v, -) → FreeNamesInStateDesignator(v),
  .4     mk-MapOrSeqRef(v, a) →
  .5          FreeNamesInStateDesignator(v) ∪ FreeNamesInExpr(a),
  .6     mk-Name(-) → {std}
  .7   end;

386.0 FreeNamesInExprOrCall : [Expr | Call] -t-> Name-set
  .1 FreeNamesInExprOrCall (ec) ≜
  .2   if ec = nil then {}
  .3   elseif Is-Expr(ec) then FreeNamesInExpr(ec)
  .4   else let mk-Call(o, a, c) = ec in
  .5        {o} ∪ FreeNamesInExprs(a) ∪ FreeNamesInStateDesignator(c);

387.0 FreeNamesInExceptions : Error* -t-> Name-set
  .1 FreeNamesInExceptions (es) ≜
  .2   ⋃{FreeNamesInExpr(c) ∪ FreeNamesInExpr(a) |
  .3        mk-Error(-, c, a) ∈ elems es};

388.0 FreeNamesInExternals : VarInf* -t-> Name-set
  .1 FreeNamesInExternals (es) ≜
  .2   ⋃{FreeNamesInType(t) | mk-VarInf(-, -, t) ∈ elems es};

389.0 FreeNamesInParTypes : PatTypePair* -t-> Name-set
  .1 FreeNamesInParTypes (ptps) ≜
  .2   ⋃{FreeNamesInPats(ps) ∪ FreeNamesInType(t) |
  .3        mk-PatTypePair(ps, t) ∈ elems ptps};

390.0 DefNamesInDefs : ValFnOpTpDefs -t-> Name-set
  .1 DefNamesInDefs (ds) ≜
  .2   ⋃{DefNamesInDef(d) | d ∈ elems ds};
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
391.0 DefNamesInDef : ValFnOpTpDef →t Name-set
  .1 DefNamesInDef (d) ≜
  .2   cases d :
  .3     mk-ValueDef(pat, -, -) → DefNamesInPat(pat),
  .4     mk-ExplFnDef(id, -, -, -, -, -) → {mk-Name(id)},
  .5     mk-ImplFnDef(id, -, -, -, -, -) → {mk-Name(id)},
  .6     mk-ExplOprtDef(id, -, -, -, -, -) → {mk-Name(id)},
  .7     mk-ImplOprtDef(id, -, -, -, -, -, -) → {mk-Name(id)},
  .8     others → dom ExtractTypeMap([d])
  .9   end;

392.0 DefNamesInPats : Pattern* →t Name-set
  .1 DefNamesInPats (ps) ≜
  .2   ⋃{DefNamesInPat(p) | p ∈ elems ps};

393.0 DefNamesInPat : Pattern →t Name-set
  .1 DefNamesInPat (p) ≜
  .2   cases p :
  .3     mk-MatchVal(e) → {},
  .4     mk-SetEnumPattern(ps) → DefNamesInPats(ps),
  .5     mk-SetUnionPattern(lp, rp) → DefNamesInPats([lp, rp]),
  .6     mk-SeqEnumPattern(ps) → DefNamesInPats(ps),
  .7     mk-SeqConcPattern(lp, rp) → DefNamesInPats([lp, rp]),
  .8     mk-TuplePattern(ps) → DefNamesInPats(ps),
  .9     mk-RecordPattern(-, ps) → DefNamesInPats(ps),
  .10    mk-PatternId(n) → if n = nil then {} else {n}
  .11  end;

394.0 DefNamesInPatBind : PatternBind →t Name-set
  .1 DefNamesInPatBind (pb) ≜
  .2   if IsPattern(pb) then DefNamesInPat(pb) else DefNamesInBind(pb);

395.0 DefNamesInBind : Bind | MultBind | BindList | TypeBindList →t Name-set
  .1 DefNamesInBind (b) ≜
  .2   cases b :
  .3     mk-SetBind(pat, -) → DefNamesInPat(pat),
  .4     mk-TypeBind(pat, -) → DefNamesInPat(pat),
  .5     mk-MultSetBind(pats, -) → DefNamesInPats(pats),
  .6     mk-MultTypeBind(pats, -) → DefNamesInPats(pats),
  .7     others → ⋃{DefNamesInBind(b') | b' ∈ elems b}
  .8   end;

396.0 DefNamesInDefBinds : DefBind* →t Name-set
  .1 DefNamesInDefBinds (ds) ≜
  .2   ⋃{DefNamesInDefBind(db) | db ∈ elems ds};

397.0 DefNamesInDefBind : DefBind →t Name-set
  .1 DefNamesInDefBind (mk-DefBind(pb, -)) ≜
  .2   DefNamesInPatBind(pb);

398.0 DefNamesInAssignDefs : AssignDef* →t Name-set
  .1 DefNamesInAssignDefs (ads) ≜
  .2   ⋃{DefNamesInAssignDef(ad) | ad ∈ elems ads};
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

399.0 $DefNamesInAssignDef : AssignDef \xrightarrow{t} Name\text{-set}$
.1 $DefNamesInAssignDef\ (mk\text{-}AssignDef(v, -, -)) \triangleq$
.2 $\{mk\text{-}Name(v)\};$

400.0 $DefNamesInEqDefs : EqDef^* \xrightarrow{t} Name\text{-set}$
.1 $DefNamesInEqDefs\ (eqds) \triangleq$
.2 $\bigcup\{DefNamesInEqDef(eqd) \mid eqd \in \text{elems } eqds\};$

401.0 $DefNamesInEqDef : EqDef \xrightarrow{t} Name\text{-set}$
.1 $DefNamesInEqDef\ (mk\text{-}EqDef(lhs, -)) \triangleq$
.2 $DefNamesInPatBind(lhs);$

402.0 $DefNamesInParTypes : PatTypePair^* \xrightarrow{t} Name\text{-set}$
.1 $DefNamesInParTypes\ (ptps) \triangleq$
.2 $\bigcup\{DefNamesInPats(ps) \mid$
.3 $mk\text{-}PatTypePair(ps, -) \in \text{elems } ptps\}$

NOTE: The names defined in patterns are those formed from embedded identifiers (however not including identifiers in match value expressions).

14.7.1.2 Syntactical Sub-Components

403.0 $Syn = ValFnOpDef \mid Bind \mid Expr \mid Stmt \mid Pattern \mid$
.1 $Type \mid OpType \mid TotalFnType \mid TypeR \mid Call \mid AssignDef \mid StateDesignator \mid Id;$

404.0 $SynCompRel = ([Syn] \times Syn)\text{-set}$

405.0 $SynComps[@T] : Syn \xrightarrow{t} [Syn]\text{-set}$
.1 $SynComps\ (d) \triangleq$
.2 $\{s \mid mk\text{-}(s, -) \in TransReflSynSubComp[Syn](d) \cdot \exists\, x : @T \cdot x = s\};$

406.0 $TransReflSynSubComp[@T] : Syn \to SynCompRel$
.1 $TransReflSynSubComp\ (s) \triangleq$
.2 $ReflExtension[Syn](TransClosure[Syn](SynSubComp[@T](s)));$

407.0 $SynSubComp[@T] : Syn \xrightarrow{t} SynCompRel$
.1 $SynSubComp\ (s) \triangleq$
.2 if $\forall\, x : @T \cdot x \neq s$ then $\{\}$
.3 else let $rel = SynComp(s)$ in
.4 $rel \cup \bigcup\{SynSubComp[@T](c) \mid mk\text{-}(c, -) \in rel\};$

408.0 $SynComp : Syn \xrightarrow{t} SynCompRel$
.1 $SynComp\ (s) \triangleq$
.2 $ValFnOpDefSynComp(s) \cup$
.3 $BindSynComp(s) \cup$
.4 $GenExprSynComp(s) \cup$
.5 $SpecExprSynComp(s) \cup$
.6 $PatSynComp(s) \cup$
.7 $TypeRSynComp(s) \cup$
.8 $TypeRTypeRel(s) \cup$
.9 $StmtSynComp(s) \cup$
.10 $AssignDefSynComp(s) \cup$
.11 $CallSynComp(s) \cup$
.12 $StateDesignatorSynComp(s);$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

409.0 *ValFnOpDefSynComp* $(d : Syn)$ *screl* : *SynCompRel*

.1 post $\forall\, sc : Syn \cdot$
.2 $\quad mk\text{-}(sc, d) \in screl \;\Leftrightarrow$
.3 $\quad$ cases d :
.4 $\quad\quad mk\text{-}ValueDef(pat, tp, val) \rightarrow sc \in \{pat, tp, val\},$
.5 $\quad\quad mk\text{-}ExplFnDef(i_1, tvs, tp, i_2, parms, body, fnpre) \rightarrow$
.6 $\quad\quad\quad sc \in \{i_1, i_2, tp, body, fnpre\} \cup \text{elems conc } parms \cup \text{elems } tvs,$
.7 $\quad\quad mk\text{-}ImplFnDef(i, tvs, partps, idtp, fnpre, fnpost) \rightarrow$
.8 $\quad\quad\quad sc \in \{i, idtp.id, idtp.type, fnpre, fnpost\} \cup \text{elems } tvs \cup$
.9 $\quad\quad\quad \bigcup\{\text{elems } pt.pat \cup \{pt.type\} \mid pt \in \text{elems } partps\},$
.10 $\quad\quad mk\text{-}ExplOprtDef(i_1, tp, i_2, params, body, oppre) \rightarrow$
.11 $\quad\quad\quad sc \in \{i_1, tp, i_2, body, oppre\} \cup \text{elems } params,$
.12 $\quad\quad mk\text{-}ImplOprtDef(i, partps, idtp, exts, oppre, oppost, excps) \rightarrow$
.13 $\quad\quad\quad sc \in \{oppre, oppost\} \cup$
.14 $\quad\quad\quad$ if $idtp \neq$ nil then $\{idtp.id, idtp.type\}$ else $\{\} \cup$
.15 $\quad\quad\quad \bigcup\{\text{elems } ps \cup \{t\} \mid mk\text{-}PatTypePair(ps, t) \in \text{elems } partps\} \cup$
.16 $\quad\quad\quad \bigcup\{\text{elems } vs \cup \{t\} \mid mk\text{-}VarInf(\text{-}, vs, t) \in \text{elems } exts\} \cup$
.17 $\quad\quad\quad \bigcup\{\{i, c, a\} \mid mk\text{-}Error(i, c, a) \in \text{elems } excps\},$
.18 $\quad\quad$ others $\rightarrow$ false
.19 $\quad$ end ;

410.0 *BindSynComp* $(b : Syn)$ *screl* : *SynCompRel*

.1 post $\forall\, sc : Syn \cdot$
.2 $\quad mk\text{-}(sc, b) \in screl \;\Leftrightarrow$
.3 $\quad$ cases b :
.4 $\quad\quad mk\text{-}SetBind(pat, bset) \rightarrow sc \in \{pat, bset\},$
.5 $\quad\quad mk\text{-}TypeBind(pat, type) \rightarrow sc \in \{pat, type\},$
.6 $\quad\quad$ others $\rightarrow$ false
.7 $\quad$ end ;

411.0 *GenExprSynComp* $(e : Syn)$ *screl* : *SynCompRel*

.1 post $\forall\, sc : Syn \cdot$
.2 $\quad mk\text{-}(sc, e) \in screl \;\Leftrightarrow$
.3 $\quad$ cases e :
.4 $\quad\quad mk\text{-}BracketedExpr((sc)) \rightarrow$ true,
.5 $\quad\quad mk\text{-}DefExpr(defs, body) \rightarrow$
.6 $\quad\quad\quad sc = body \vee$
.7 $\quad\quad\quad \exists\, mk\text{-}DefBind(pb, rhs) \in \text{elems } defs \cdot$
.8 $\quad\quad\quad\quad sc \in \{pb, rhs\},$
.9 $\quad\quad mk\text{-}LetExpr(ldefs, body) \rightarrow$
.10 $\quad\quad\quad sc = body \vee sc \in \text{elems } ldefs,$
.11 $\quad\quad mk\text{-}LetBeSTExpr(bind, cond, body) \rightarrow$
.12 $\quad\quad\quad sc \in \{bind, cond, body\},$
.13 $\quad\quad mk\text{-}IfExpr(test, cons, altns, altn) \rightarrow$
.14 $\quad\quad\quad sc \in \{test, cons, altn\} \vee$
.15 $\quad\quad\quad \exists\, mk\text{-}ElsifExpr(test, cons) \in \text{elems } altns \cdot$
.16 $\quad\quad\quad\quad sc \in \{test, cons\},$
.17 $\quad\quad mk\text{-}CasesExpr(sel, altns, other) \rightarrow$
.18 $\quad\quad\quad sc \in \{sel, other\} \vee$
.19 $\quad\quad\quad \exists\, mk\text{-}CaseAltn(ps, body) \in \text{elems } altns \cdot$
.20 $\quad\quad\quad\quad sc \in \{body\} \cup \text{elems } ps,$
.21 $\quad\quad mk\text{-}PrefixExpr(\text{-}, (sc)) \rightarrow$ true,
.22 $\quad\quad mk\text{-}BinaryExpr(left, \text{-}, right) \rightarrow sc \in \{left, right\},$
.23 $\quad\quad$ others $\rightarrow$ false
.24 $\quad$ end ;

412.0 *SpecExprSynComp* $(e : Syn)$ *screl* : *SynCompRel*

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

.1 post $\forall\, sc : Syn \cdot$
.2 $\quad mk\text{-}(sc, e) \in screl \;\Leftrightarrow$
.3 $\quad$ cases e :
.4 $\quad\quad mk\text{-}MapInverseExpr((sc)) \rightarrow$ true,
.5 $\quad\quad mk\text{-}ExistsExpr(bs, p), mk\text{-}ExistsUniqueExpr(bs, p) \rightarrow$
.6 $\quad\quad\quad sc \in \{p\} \cup$ elems bs,
.7 $\quad\quad mk\text{-}AllExpr(bs, p), mk\text{-}IotaExpr(bs, p) \rightarrow$
.8 $\quad\quad\quad sc \in \{p\} \cup$ elems bs,
.9 $\quad\quad mk\text{-}SetEnumeration(es) \rightarrow sc \in$ elems es,
.10 $\quad\quad mk\text{-}SetComprehension(e, bs, p) \rightarrow$
.11 $\quad\quad\quad sc \in \{e, p\} \cup$ elems bs,
.12 $\quad\quad mk\text{-}SetRange(lb, ub) \rightarrow sc \in \{lb, ub\}$,
.13 $\quad\quad mk\text{-}SeqEnumeration(es) \rightarrow sc \in$ elems es,
.14 $\quad\quad mk\text{-}SeqComprehension(e, b, p) \rightarrow$
.15 $\quad\quad\quad sc \in \{e, b, p\}$,
.16 $\quad\quad mk\text{-}SubSequence(s, lb, ub) \rightarrow sc \in \{s, lb, ub\}$,
.17 $\quad\quad mk\text{-}MapEnumeration(mls) \rightarrow$
.18 $\quad\quad\quad \exists\, mk\text{-}Maplet(d, r) \in$ elems $mls \cdot sc \in \{d, r\}$,
.19 $\quad\quad mk\text{-}MapComprehension(mk\text{-}Maplet(d, r), bs, p) \rightarrow$
.20 $\quad\quad\quad sc \in \{d, r, p\} \cup$ elems bs,
.21 $\quad\quad mk\text{-}TupleConstructor(es) \rightarrow sc \in$ elems es,
.22 $\quad\quad mk\text{-}RecordConstructor(i, es) \rightarrow sc \in \{i\} \cup$ elems es,
.23 $\quad\quad mk\text{-}RecordModifier(rec, rms) \rightarrow$
.24 $\quad\quad\quad sc \in \{rec\} \cup$
.25 $\quad\quad\quad \bigcup \{\{f, new\} \mid mk\text{-}RecordModification(f, new) \in$ elems $rms\}$,
.26 $\quad\quad mk\text{-}Apply(f, args) \rightarrow sc \in \{f\} \cup$ elems $args$,
.27 $\quad\quad mk\text{-}FieldSelect(r, f) \rightarrow sc \in \{r, f\}$,
.28 $\quad\quad mk\text{-}Lambda(tbs, body) \rightarrow sc \in \{body\} \cup$ elems tbs,
.29 $\quad\quad mk\text{-}IsDefTypeExpr(\text{-}, (sc)) \rightarrow$ true,
.30 $\quad\quad mk\text{-}IsBasicTypeExpr(\text{-}, (sc)) \rightarrow$ true,
.31 $\quad\quad mk\text{-}Name(i), mk\text{-}OldName(i) \rightarrow sc = i$,
.32 $\quad\quad$ others $\rightarrow$ false
.33 $\quad$ end ;

413.0 $StmtSynComp\ (s : Syn)\ screl : SynCompRel$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
.1 post ∀ sc : Syn ·
.2          mk-(sc, s) ∈ screl  ⇔
.3          cases s :
.4            mk-BlockStmt(ds, ss) → sc ∈ elems ds ∪ elems ss,
.5            mk-AlwaysStmt(alwpost, body) → sc ∈ {alwpost, body},
.6            mk-TrapStmt(pat, trappost, body) → sc ∈ {pat, trappost, body},
.7            mk-RecTrapStmt(traps, body) →
.8                (∃ mk-Trap(match, trappost) ∈ elems traps · sc ∈ {match, trappost}) ∨
.9                sc = body,
.10           mk-DefStmt(eqdefs, body) →
.11               (∃ mk-EqDef(lhs, rhs) ∈ elems eqdefs · sc ∈ {lhs, rhs}) ∨
.12               sc = body,
.13           mk-LetStmt(letdefs, body) → sc ∈ elems letdefs ∪ {body},
.14           mk-LetBeSTStmt(bind, cond, body) → sc ∈ {bind, cond, body},
.15           mk-AssignStmt(lhs, rhs) → sc ∈ {lhs, rhs},
.16           mk-NonDetStmt(ss) → sc ∈ elems ss,
.17           mk-SeqForLoop(cv, d, forseq, body) → sc ∈ {cv, d, forseq, body},
.18           mk-SetForLoop(cv, forset, body) → sc ∈ {cv, forset, body},
.19           mk-IndexForLoop(cv, lb, ub, step, body) → sc ∈ {cv, lb, ub, step, body},
.20           mk-WhileLoop(test, body) → sc ∈ {test, body},
.21           mk-IfStmt(test, cons, elsifaltn, altn) →
.22               sc ∈ {test, cons, altn} ∨
.23               ∃ mk-ElsifStmt(test, cons) ∈ elems elsifaltn · sc ∈ {test, cons},
.24           mk-CasesStmt(sel, altns, other) →
.25               sc ∈ {sel, other} ∨
.26               ∃ mk-CaseStmtAltn(match, body) ∈ elems altns ·
.27                   sc ∈ elems match ∪ {body},
.28           mk-ReturnStmt((sc)) → true,
.29           mk-ExitStmt((sc)) → true,
.30           others → false
.31         end ;

414.0 AssignDefSynComp (s : Syn) screl : SynCompRel
.1 post ∀ sc : Syn ·
.2          mk-(sc, s) ∈ screl  ⇔
.3          cases s :
.4            mk-AssignDef(v, tp, dcli) → sc ∈ {v, tp, dcli},
.5            others → false
.6          end ;

415.0 CallSynComp (s : Syn) screl : SynCompRel
.1 post ∀ sc : Syn ·
.2          mk-(sc, s) ∈ screl  ⇔
.3          cases s :
.4            mk-Call(n, el, std) → sc ∈ {n, el, std},
.5            others → false
.6          end ;

416.0 StateDesignatorSynComp (s : Syn) screl : SynCompRel
.1 post ∀ sc : Syn ·
.2          mk-(sc, s) ∈ screl  ⇔
.3          cases s :
.4            mk-MapOrSeqRef(v, e) → sc ∈ {v, e},
.5            mk-FieldRef(v, i) → sc ∈ {v, i},
.6            others → false
.7          end ;

417.0 PatSynComp (p : Syn) screl : SynCompRel
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
.1 post ∀ sc : Syn ·
.2          mk-(sc, p) ∈ screl  ⇔
.3          cases p :
.4            mk-PatternId((sc)) → true,
.5            mk-MatchVal((sc)) → true,
.6            mk-SetEnumPattern(ps) → sc ∈ elems ps,
.7            mk-SetUnionPattern(lp, rp) → sc ∈ {lp, rp},
.8            mk-SeqEnumPattern(ps) → sc ∈ elems ps,
.9            mk-SeqConcPattern(lp, rp) → sc ∈ {lp, rp},
.10           mk-TuplePattern(ps) → sc ∈ elems ps,
.11           mk-RecordPattern(i, ps) → sc ∈ {i} ∪ elems ps,
.12           others → false
.13         end ;
```

418.0 *TypeRSynComp* $(t : Syn)$ *screl* : *SynCompRel*

```
.1 post ∀ sc : Syn ·
.2          mk-(sc, t) ∈ screl  ⇔
.3          cases t :
.4            mk-InvTypeR(sh, mk-InvInitFn(p, e)) → sc ∈ {sh, p, e},
.5            mk-CompositeTypeR(i, fs) →
.6                 sc = i ∨ ∃ mk-FieldR(f, t) ∈ elems fs · sc ∈ {f, t},
.7            mk-UnionTypeR(ts) → sc ∈ ts,
.8            mk-ProductTypeR(ts) → sc ∈ elems ts,
.9            mk-SetTypeR((sc)) → true,
.10           mk-Seq0TypeR((sc)) → true,
.11           mk-Seq1TypeR((sc)) → true,
.12           mk-GeneralMapTypeR(dt, rt) → sc ∈ {dt, rt},
.13           mk-InjectiveMapTypeR(dt, rt) → sc ∈ {dt, rt},
.14           mk-PartialFnTypeR(dt, rt) → sc ∈ {dt, rt},
.15           mk-TotalFnTypeR(dt, rt) → sc ∈ {dt, rt},
.16           mk-OpTypeR(i, dt, rt) → sc ∈ {i, dt, rt},
.17           others → false
.18         end ;
```

419.0 *TypeRTypeRel* : *Syn* $\xrightarrow{t}$ *SynCompRel*

```
.1 TypeRTypeRel (s) △
.2   if IsType(s) ∨ is-OpType(s) ∨ is-TotalFnType(s)
.3   then {mk-(ExtractTypeR(s)(EmptyEnv), s)}
.4   else {}
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

14.7.1.3 The Definitional Basis of Types

```
420.0 TypeRDefBasis : Π →t TypeR → TypeR-set
  .1  TypeRDefBasis (π)(type) ≜
  .2    {type} ∪
  .3    cases type :
  .4      mk-InvTypeR(t, -) → if is-DEF(π) then {} else TypeRDefBasis(π)(t),
  .5      mk-CompositeTypeR(id, fs) →
  .6          ⋃{TypeRDefBasis(π)(t) | mk-FieldR(-, t) ∈ elems fs},
  .7      mk-UnionTypeR(ts) → ⋃{TypeRDefBasis(π)(t) | t ∈ ts},
  .8      mk-ProductTypeR(ts) → ⋃{TypeRDefBasis(π)(t) | t ∈ elems ts},
  .9      mk-SetTypeR(t) → TypeRDefBasis(π)(t),
  .10     mk-Seq0TypeR(t) → TypeRDefBasis(π)(t),
  .11     mk-Seq1TypeR(t) → TypeRDefBasis(π)(t),
  .12     mk-GeneralMapTypeR(dt, rt) → TypeRDefBasis(π)(dt) ∪ TypeRDefBasis(π)(rt),
  .13     mk-InjectiveMapTypeR(dt, rt) → TypeRDefBasis(π)(dt) ∪ TypeRDefBasis(π)(rt),
  .14     mk-PartialFnTypeR(dt, rt) → TypeRDefBasis(π)(dt) ∪ TypeRDefBasis(π)(rt),
  .15     mk-TotalFnTypeR(dt, rt) → TypeRDefBasis(π)(dt) ∪ TypeRDefBasis(π)(rt),
  .16     others → {}
  .17   end
```

According to the Dynamic Semantics, there are restrictions on the kind of types which may appear in certain contexts. There are, for example, contexts where only flat types are allowed, i.e. types which do not build on function types. The above function identifies the type sub-expressions which a type builds on, taking into account the possibility that an invariant may exclude all elements of a type.

14.7.2 Syntax Transformations

The static semantics of function definitions must take into account the pattern matching in connection with formal parameters and also the special syntax for explicit definitions of curried functions. Both issues are dealt with by transformation: (1) pattern matching of formal parameters is transformed to a let-expression enclosing the function body; and (2) the special syntax for currying is transformed to lambda expressions enclosing the function body. The Static Semantics is then expressed relative to the transformed function bodies rather than the original ones.

```
421.0 IsCurriedFn : LambdaR →t
  .1                Parameters+ × FunctionTypeR × [Expr] × Expr → Env → 𝔹
  .2  IsCurriedFn (mk-LambdaR(id, argtp, body1, fnpre1, tot))(patseqseq, tp, fnpre2, body2)(env) ≜
  .3    let pat = Pats2Pat(hd patseqseq) in
  .4    ∃ xbody : Expr ·
  .5        tot = is-TotalFnTypeR(tp) ∧
  .6        argtp = tp.fndom ∧
  .7        IsLetExpansion(fnpre1)(pat, nil, mk-Name(id), fnpre2)(env) ∧
  .8        IsLetExpansion(xbody)(pat, nil, mk-Name(id), body2)(env) ∧
  .9        if len patseqseq = 1
  .10       then body1 = xbody
  .11       else Is[FunctionTypeR, TypeR](tp.fnrng) ∧
  .12            IsCurriedFn(body1)(tl patseqseq, tp.fnrng, nil, xbody)(env);

422.0 IsLetExpandedParTypes : [Stmt | Expr] →t
  .1                          ParameterTypes × Name × [Stmt | Expr] → Env → 𝔹
  .2  IsLetExpandedParTypes (letse)(pts, n, body)(env) ≜
  .3    let p = ParameterTypes2Pattern(pts),
  .4        t = ParameterTypes2Type(pts) in
  .5    IsLetExpansion(letse)(p, t, n, body)(env);
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
423.0 IsLetExpansion : [Stmt | Expr] -t->
   .1                  Pattern × [TypeR] × Name × [Stmt | Expr] → Env → 𝔹
   .2 IsLetExpansion (letse)(p, t, n, body)(env) ≜
   .3   let bodynames = if Is[Stmt, [Stmt | Expr]](body)
   .4                   then FreeNamesInStmt(body)
   .5                   else FreeNamesInExpr(body) in
   .6   n ∉ dom env.visibleenv ∪ StateVarNames(env) ∪ bodynames ∧
   .7   ({letse, body} = {nil} ∨
   .8    if Is[Stmt, [Stmt | Expr]](body)
   .9    then letse = mk-LetExpr([mk-ValueDef(p, t, n)], body)
   .10   else letse = mk-LetStmt([mk-ValueDef(p, t, n)], body));

424.0 ParameterTypes2Pattern : ParameterTypes -t-> Pattern
   .1 ParameterTypes2Pattern (pts) ≜
   .2   Pats2Pat(ParameterTypes2Pats(pts));

425.0 ParameterTypes2Pats : ParameterTypes -t-> Pattern*
   .1 ParameterTypes2Pats (pts) ≜
   .2   conc [[pts(i).pat(j) | j ∈ inds pts(i).pat] | i ∈ inds pts];

426.0 Pats2Pat : Pattern* -t-> Pattern
   .1 Pats2Pat (pats) ≜
   .2   cases len pats :
   .3     (0) → mk-MatchVal(mk-NilLit()),
   .4     (1) → hd pats,
   .5     others → mk-TuplePattern(pats)
   .6   end;

427.0 ParameterTypes2Type : ParameterTypes -t-> Type
   .1 ParameterTypes2Type (pts) ≜
   .2   Types2Type(ParameterTypes2Types(pts));

428.0 ParameterTypes2Types : ParameterTypes -t-> Type*
   .1 ParameterTypes2Types (pts) ≜
   .2   [pts(i).type | i ∈ inds pts];

429.0 Types2Type : Type* -t-> Type
   .1 Types2Type (ts) ≜
   .2   cases len ts :
   .3     (0) → UNITTYPE,
   .4     (1) → hd ts,
   .5     others → mk-ProductType(ts)
   .6   end;

430.0 OType2Types : [Type] -t-> Type*
   .1 OType2Types (t) ≜
   .2   cases t :
   .3     (nil) → [],
   .4     mk-ProductType(ts) → ts,
   .5     others → [t]
   .6   end;
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
431.0 Exprs2Expr : Expr* ~t~> Expr
  .1 Exprs2Expr (es) ≜
  .2   cases len es :
  .3     (0) → mk-NilLit(),
  .4     (1) → hd es,
  .5     others → mk-TupleConstructor(es)
  .6   end;

432.0 PreOExceptions2Pre : [Expr] × OExceptions ~t~> [Expr]
  .1 PreOExceptions2Pre (oppre, opcps) ≜
  .2   let oppre-s = if oppre = nil then {} else {oppre} in
  .3   GetOrExpr(oppre-s ∪ {opcps(i).cond | i ∈ inds opcps});

433.0 PostOExceptions2Post : Expr × OExceptions ~t~> Expr
  .1 PostOExceptions2Post (oppost, opcps) ≜
  .2   let err-post-s = {mk-BinaryExpr(cond, AND, action) |
  .3                     mk-Error(-, cond, action) ∈ elems opcps},
  .4       not-err-pre = GetOrExpr({opcps(i).cond | i ∈ inds opcps}),
  .5       post1 = mk-BinaryExpr(oppost, AND, mk-PrefixExpr(NOT, not-err-pre)) in
  .6   GetOrExpr({post1} ∪ err-post-s);

434.0 GetOrExpr : Expr-set ~t~> [Expr]
  .1 GetOrExpr (es) ≜
  .2   if es = {}
  .3   then nil
  .4   elseif card es = 1 then let e ∈ es in e
  .5   else let e ∈ es in mk-BinaryExpr(e, OR, GetOrExpr(es \ {e}))
```

14.7.3 Substitutions

Substitutions are necessary in two cases: (1) to find the type of instantiations of polymorphic functions, and (2) to get real variables instead of the "hooked" variables when extracting post-OP function definitions from pre/post operation definitions. In the syntax, hooked variables are the elements of the syntax class: "OldName". It is assumed that the substitutions for the hooked variables are names which will not be captured by local bindings.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
435.0 TypeSubst : TypeR -m-> TypeR -t-> TypeR → TypeR
  .1  TypeSubst (tm)(t) ≜
  .2    if t ∈ dom tm
  .3    then TypeSubst({t} ⩤ tm)(tm(t))
  .4    else cases t :
  .5          mk-InvTypeR(t1, inva) → mk-InvTypeR( TypeSubst(tm)(t1), inva),
  .6          mk-CompositeTypeR(id, fields) →
  .7              let newfields =
  .8                    [let mk-FieldR(fid, ft) = fields(i) in
  .9                     mk-FieldR(fid, TypeSubst(tm)(ft)) | i ∈ inds fields] in
  .10             mk-CompositeTypeR(id, newfields),
  .11         mk-UnionTypeR(summands) →
  .12             mk-UnionTypeR({ TypeSubst(tm)(sumt) | sumt ∈ summands}),
  .13         mk-ProductTypeR(factors) →
  .14             mk-ProductTypeR([TypeSubst(tm)(factors(i)) | i ∈ inds factors]),
  .15         mk-SetTypeR(et) → mk-SetTypeR( TypeSubst(tm)(et)),
  .16         mk-Seq0TypeR(et) → mk-Seq0TypeR( TypeSubst(tm)(et)),
  .17         mk-Seq1TypeR(et) → mk-Seq1TypeR( TypeSubst(tm)(et)),
  .18         mk-GeneralMapTypeR( dt, rt) →
  .19             mk-GeneralMapTypeR( TypeSubst(tm)(dt), TypeSubst(tm)(rt)),
  .20         mk-InjectiveMapTypeR(dt, rt) →
  .21             mk-InjectiveMapTypeR( TypeSubst(tm)(dt), TypeSubst(tm)(rt)),
  .22         mk-PartialFnTypeR(dt, rt) →
  .23             mk-PartialFnTypeR( TypeSubst(tm)(dt), TypeSubst(tm)(rt)),
  .24         mk-TotalFnTypeR(dt, rt) →
  .25             mk-TotalFnTypeR( TypeSubst(tm)(dt), TypeSubst(tm)(rt)),
  .26         mk-RetTypeR(et) → mk-RetTypeR( TypeSubst(tm)(et)),
  .27         mk-ExitTypeR(et) → mk-ExitTypeR( TypeSubst(tm)(et)),
  .28         mk-OpTypeR(stt, dt, rt) →
  .29             mk-OpTypeR( TypeSubst(tm)(stt), TypeSubst(tm)(dt), TypeSubst(tm)(rt)),
  .30         others → t
  .31       end;

436.0 ONInExprsSubst : OldName -m-> Name → Expr* → Expr*
  .1  ONInExprsSubst (nm)(es) ≜
  .2    [ONInExprSubst(nm)(es(i)) | i ∈ inds es];
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
437.0 ONInExprSubst : OldName -m-> Name -t-> Expr → Expr
  .1  ONInExprSubst (nm)(e) ≙
  .2    if e ∈ dom nm then nm(e)
  .3    else let S = ONInExprSubst(nm) in
  .4         cases e :
  .5           mk-BracketedExpr(e1) → mk-BracketedExpr(S(e1)),
  .6           mk-DefExpr(defs, body) →
  .7                mk-DefExpr(ONInDefBindsSubst(nm)(defs), S(body)),
  .8           mk-LetExpr(ldefs, body) →
  .9                mk-LetExpr(ONInLocalDefsSubst(nm)(ldefs), S(body)),
  .10          mk-LetBeSTExpr(bind, cond, body) →
  .11               mk-LetBeSTExpr(ONInBindSubst(nm)(bind), S(cond), S(body)),
  .12          mk-IfExpr(test, cons, altns, altn) →
  .13               mk-IfExpr(S(test), S(cons),
  .14                [mk-ElsifExpr(S(altns(i).test), S(altns(i).cons)) | i ∈ inds altns],
  .15                S(altn)),
  .16          mk-CasesExpr(sel, altns, other) →
  .17               mk-CasesExpr(S(sel),
  .18                [mk-CaseAltn(ONInPatsSubst(nm)(altns(i).match), S(altns(i).body)) |
  .19                     i ∈ inds altns],
  .20                S(other)),
  .21          mk-PrefixExpr(op, e1) → mk-PrefixExpr(op, S(e1)),
  .22          mk-BinaryExpr(left, op, right) → mk-BinaryExpr(S(left), op, S(right)),
  .23          others → ONInSpecExprSubst(nm)(e)
  .24        end;
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
438.0 ONInSpecExprSubst : OldName -m-> Name -t-> Expr → Expr
  .1 ONInSpecExprSubst (nm)(e) ≜
  .2   let S = ONInExprSubst(nm) in
  .3   cases e :
  .4     mk-MapInverseExpr(e1) → mk-MapInverseExpr(S(e1)),
  .5     mk-ExistsExpr(bs, p) →
  .6         mk-ExistsExpr(ONInBindsSubst(nm)(bs), S(p)),
  .7     mk-ExistsUniqueExpr(bs, p) →
  .8         mk-ExistsUniqueExpr(ONInBindSubst(nm)(bs), S(p)),
  .9     mk-AllExpr(bs, p) →
  .10        mk-AllExpr(ONInBindsSubst(nm)(bs), S(p)),
  .11    mk-IotaExpr(bs, p) →
  .12        mk-IotaExpr(ONInBindSubst(nm)(bs), S(p)),
  .13    mk-SetEnumeration(es) → mk-SetEnumeration(ONInExprsSubst(nm)(es)),
  .14    mk-SetComprehension(e, bs, p) →
  .15        mk-SetComprehension(S(e), ONInBindsSubst(nm)(bs), S(p)),
  .16    mk-SetRange(lb, ub) → mk-SetRange(S(lb), S(ub)),
  .17    mk-SeqEnumeration(es) → mk-SeqEnumeration(ONInExprsSubst(nm)(es)),
  .18    mk-SeqComprehension(e, b, p) →
  .19        mk-SeqComprehension(S(e), ONInBindSubst(nm)(b), S(p)),
  .20    mk-SubSequence(s, lb, ub) → mk-SubSequence(S(s), S(lb), S(ub)),
  .21    mk-MapEnumeration(mls) →
  .22        mk-MapEnumeration([mk-Maplet(S(mls(i).mapdom), S(mls(i).maprng)) |
  .23            i ∈ inds mls]),
  .24    mk-MapComprehension(mk-Maplet(d, r), bs, p) →
  .25        mk-MapComprehension(mk-Maplet(S(d), S(r)), ONInBindsSubst(nm)(bs), S(p)),
  .26    mk-TupleConstructor(es) → mk-TupleConstructor(ONInExprsSubst(nm)(es)),
  .27    mk-RecordConstructor(t, es) → mk-RecordConstructor(t, ONInExprsSubst(nm)(es)),
  .28    mk-RecordModifier(rec, rms) →
  .29        mk-RecordModifier(S(rec),
  .30         [mk-RecordModification(rms(i).field, S(rms(i).new)) | i ∈ inds rms]),
  .31    mk-Apply(e1, es) → mk-Apply(S(e1), ONInExprsSubst(nm)(es)),
  .32    mk-FieldSelect(e1, f) → mk-FieldSelect(S(e1), f),
  .33    mk-Lambda(tbs, body) → mk-Lambda(ONInTypeBindsSubst(nm)(tbs), S(body)),
  .34    mk-IsDefTypeExpr(dt, e1) → mk-IsDefTypeExpr(dt, S(e1)),
  .35    mk-IsBasicTypeExpr(bt, e1) → mk-IsBasicTypeExpr(bt, S(e1)),
  .36    others → e
  .37  end;

439.0 ONInDefsSubst : OldName -m-> Name → EqDef+ → EqDef+
  .1 ONInDefsSubst (nm)(eqdefs) ≜
  .2   [mk-EqDef(ONInPatBindSubst(nm)(eqdefs(i).lhs),
  .3     ONInExprOrCallSubst(nm)(eqdefs(i).rhs)) |
  .4       i ∈ inds eqdefs];

440.0 ONInDefBindsSubst : OldName -m-> Name → DefBind* → DefBind*
  .1 ONInDefBindsSubst (nm)(dbs) ≜
  .2   [mk-DefBind(ONInPatBindSubst(nm)(dbs(i).lhs),
  .3     ONInExprSubst(nm)(dbs(i).rhs)) |
  .4       i ∈ inds dbs];

441.0 ONInPatBindSubst : OldName -m-> Name → PatternBind → PatternBind
  .1 ONInPatBindSubst (nm)(pb) ≜
  .2   if Is[Pattern, PatternBind](pb) then ONInPatternSubst(nm)(pb)
  .3   else ONInBindSubst(nm)(pb);
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
442.0 ONInExprOrCallSubst : OldName ->m Name → Expr | Call → Expr | Call
  .1 ONInExprOrCallSubst (nm)(ec) ≜
  .2   if Is[Expr, Expr | Call](ec) then ONInExprSubst(nm)(ec)
  .3   else μ (ec, args ↦ ONInExprsSubst(nm)(ec.args));

443.0 ONInOExprSubst : OldName ->m Name → [Expr] → [Expr]
  .1 ONInOExprSubst (nm)(oe) ≜
  .2   if oe = nil then nil
  .3   else ONInExprSubst(nm)(oe);

444.0 ONInLocalDefsSubst : OldName ->m Name → LocalDef+ → LocalDef+
  .1 ONInLocalDefsSubst (nm)(ds) ≜
  .2   [ONInLocalDefSubst(nm)(ds(i)) | i ∈ inds ds];

445.0 ONInLocalDefSubst : OldName ->m Name → LocalDef → LocalDef
  .1 ONInLocalDefSubst (nm)(d) ≜
  .2   cases d :
  .3     mk-ValueDef(p, t, v) →
  .4         mk-ValueDef(ONInPatternSubst(nm)(p), t, ONInExprSubst(nm)(v)),
  .5     mk-ExplFnDef(i1, tv, t, i2, p, b, pr) →
  .6         mk-ExplFnDef(i1, tv, t, i2,
  .7          ONInParListSubst(nm)(p), ONInExprSubst(nm)(b), ONInOExprSubst(nm)(pr)),
  .8     mk-ImplFnDef(i, tv, pt, it, fpre, fpost) →
  .9         mk-ImplFnDef(i, tv, ONInParTypesSubst(nm)(pt), it,
  .10         ONInOExprSubst(nm)(fpre), ONInExprSubst(nm)(fpost))
  .11  end;

446.0 ONInParTypesSubst : OldName ->m Name → PatTypePair* → PatTypePair*
  .1 ONInParTypesSubst (nm)(pts) ≜
  .2   [μ (pts(i), pat ↦ ONInPatsSubst(nm)(pts(i).pat)) | i ∈ inds pts];

447.0 ONInPatsSubst : OldName ->m Name → Pattern* → Pattern*
  .1 ONInPatsSubst (nm)(ps) ≜
  .2   [ONInPatternSubst(nm)(ps(i)) | i ∈ inds ps];

448.0 OnInParListSubst : OldName ->m Name → ParametersList → ParametersList
  .1 OnInParListSubst (nm)(pl) ≜
  .2   [ONInPatsSubst(nm)(pl(i)) | i ∈ inds pl];

449.0 ONInPatternSubst : OldName ->m Name → Pattern → Pattern
  .1 ONInPatternSubst (nm)(p) ≜
  .2   let S = ONInPatternSubst(nm),
  .3       Ss = ONInPatsSubst(nm) in
  .4   cases p :
  .5     mk-MatchVal(e) → mk-MatchVal(ONInExprSubst(nm)(e)),
  .6     mk-SetEnumPattern(ps) → mk-SetEnumPattern(Ss(ps)),
  .7     mk-SetUnionPattern(lp, rp) → mk-SetUnionPattern(S(lp), S(rp)),
  .8     mk-SeqEnumPattern(ps) → mk-SeqEnumPattern(Ss(ps)),
  .9     mk-SeqConcPattern(lp, rp) → mk-SeqConcPattern(S(lp), S(rp)),
  .10    mk-TuplePattern(ps) → mk-TuplePattern(Ss(ps)),
  .11    mk-RecordPattern(i, ps) → mk-RecordPattern(i, Ss(ps)),
  .12    mk-PatternId(-) → p
  .13  end;
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
450.0 ONInBindsSubst : OldName -m-> Name → MultBind* → MultBind*
  .1 ONInBindsSubst (nm)(mbs) ≙
  .2   [ONInMultBindSubst(nm)(mbs(i)) | i ∈ inds mbs];

451.0 ONInTypeBindsSubst : OldName -m-> Name → TypeBind* → TypeBind*
  .1 ONInTypeBindsSubst (nm)(tbs) ≙
  .2   [ONInBindSubst(nm)(tbs(i)) | i ∈ inds tbs];

452.0 ONInMultBindSubst : OldName -m-> Name → MultBind → MultBind
  .1 ONInMultBindSubst (nm)(mb) ≙
  .2   cases mb :
  .3     mk-MultSetBind(ps, e) → mk-MultSetBind(ONInPatsSubst(nm)(ps), ONInExprSubst(nm)(e)),
  .4     mk-MultTypeBind(ps, t) → mk-MultTypeBind(ONInPatsSubst(nm)(ps), t)
  .5   end;

453.0 ONInBindSubst : OldName -m-> Name → Bind → Bind
  .1 ONInBindSubst (nm)(b) ≙
  .2   cases b :
  .3     mk-SetBind(p, e) → mk-SetBind(ONInPatternSubst(nm)(p), ONInExprSubst(nm)(e)),
  .4     mk-TypeBind(p, t) → mk-TypeBind(ONInPatternSubst(nm)(p), t)
  .5   end
```

14.7.4 Indirectly Defined Functions

In the following we specify the extraction of indirectly defined functions. Note, however, that some of the trivial extraction functions are left unspecified. Please refer to the syntax transformation where similar functions are defined.

```
454.0 ExtractFnPrePostDefs : FunctionDef* -t-> ExplFnDef*
  .1 ExtractFnPrePostDefs (fndefs) ≙
  .2   if fndefs = []
  .3   then []
  .4   else let prefn = if is-ExplFnDef(hd fndefs)
  .5                    then ExtractFnPreDefFromExplDef(hd fndefs)
  .6                    else ExtractFnPreDefFromImplDef(hd fndefs),
  .7            postfns = if is-ExplFnDef(hd fndefs)
  .8                      then []
  .9                      else [ExtractFnPostDefFromImplDef(hd fndefs)] in
  .10       [prefn] ↷ postfns ↷ ExtractFnPrePostDefs(tl fndefs);

455.0 ExtractFnPreDefFromExplDef : ExplFnDef -t-> ExplFnDef
  .1 ExtractFnPreDefFromExplDef (mk-ExplFnDef(id, tpparms, type, -, parms, -, fnpre)) ≙
  .2   let idpre = mk-PreId(GetToken(id)),
  .3       tp = mk-PartialFnType(type.fndom, BOOLEAN),
  .4       body = if fnpre ≠ nil
  .5              then fnpre
  .6              else mk-BoolLit(true) in
  .7   mk-ExplFnDef(idpre, tpparms, tp, idpre, parms, body, nil);
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
456.0 ExtractFnPreDefFromImplDef : ImplFnDef ~t~> ExplFnDef
  .1 ExtractFnPreDefFromImplDef (mk-ImplFnDef(id, tpparms, partps, -, fnpre, -)) ≙
  .2   let idpre = mk-PreId(GetToken(id)),
  .3       tp = mk-PartialFnType(ParameterTypes2Type(partps), BOOLEAN),
  .4       body = if fnpre ≠ nil
  .5              then fnpre
  .6              else mk-BoolLit(true),
  .7       parms = ParameterTypes2Pats(partps) in
  .8   mk-ExplFnDef(idpre, tpparms, tp, idpre, [parms], body, nil);

457.0 ExtractFnPostDefFromImplDef : ImplFnDef ~t~> ExplFnDef
  .1 ExtractFnPostDefFromImplDef (mk-ImplFnDef(id, tpparms, partps, residtype, -, fnpost)) ≙
  .2   let idpost = mk-PostId(GetToken(id)),
  .3       tp = mk-TotalFnType(Types2Type(ParameterTypes2Types(partps) ↷ [residtype.type]), BOOLEAN),
  .4       parms = ParameterTypes2Pats(partps) in
  .5   mk-ExplFnDef(idpost, tpparms, tp, idpost, [parms ↷ [mk-PatternId(residtype.id)]], fnpost, nil);

458.0 ExtractOpPrePostDefs : OperationDef* × StateDef* × Id-set ~t~> ExplFnDef*
  .1 ExtractOpPrePostDefs (opdefs, stdefs, used) ≙
  .2   if opdefs = []
  .3   then []
  .4   else let sttypes = if StateName(stdefs) = nil then [] else [StateName(stdefs)],
  .5            mk-(stpats, onsubst) = StatePats(stdefs, used),
  .6            prefns = if Is[ExplOprtDef, OperationDef](hd opdefs)
  .7                     then ExtractOpPreDefFromExplDef(hd opdefs)(sttypes, stpats(1, ..., 1))
  .8                     else ExtractOpPreDefFromImplDef(hd opdefs)(sttypes, stpats(1, ..., 1)),
  .9            postfns = if Is[ExplOprtDef, OperationDef](hd opdefs)
  .10                     then []
  .11                     else [ExtractOpPostDefFromImplDef(hd opdefs)(sttypes, stpats, onsubst)] in
  .12       prefns ↷ postfns ↷ ExtractOpPrePostDefs(tl opdefs, stdefs, used);

459.0 ExtractOpPreDefFromExplDef : ExplOprtDef ~t~> TypeR* × Pattern* → ExplFnDef*
  .1 ExtractOpPreDefFromExplDef
  .2      (mk-ExplOprtDef(id, mk-OpType(odom, -), -, params, body, oppre))(sttypes, stpats) ≙
  .3   let idpre = mk-PreId(GetToken(id)),
  .4       tp = mk-PartialFnType(Types2Type(OType2Types(odom) ↷ sttypes), BOOLEAN),
  .5       parms = params ↷ stpats,
  .6       body = if oppre ≠ nil
  .7              then oppre
  .8              else mk-BoolLit(true) in
  .9   [mk-ExplFnDef(idpre, [], tp, idpre, [parms], body, nil)];

460.0 ExtractOpPreDefFromImplDef : ImplOprtDef ~t~> TypeR* × Pattern* → ExplFnDef*
  .1 ExtractOpPreDefFromImplDef
  .2      (mk-ImplOprtDef(id, partps, residtype, -, oppre, -, -))(sttypes, stpats) ≙
  .3   let idpre = mk-PreId(GetToken(id)),
  .4       tp = mk-PartialFnType(Types2Type(ParameterTypes2Types(partps) ↷ sttypes), BOOLEAN),
  .5       parms = ParameterTypes2Pats(partps) ↷ stpats,
  .6       body = if oppre ≠ nil
  .7              then oppre
  .8              else mk-BoolLit(true) in
  .9   [mk-ExplFnDef(idpre, [], tp, idpre, [parms], body, nil)];
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
461.0 ExtractOpPostDefFromImplDef :
   .1     ImplOprtDef -t-> TypeR* × Pattern* × (OldName -m-> Name) → ExplFnDef*
   .2 ExtractOpPostDefFromImplDef
   .3     (mk-ImplOprtDef(id, partps, residtype, -, -, oppost, -))(sttypes, stpats, subst) ≙
   .4   let idpost = mk-PostId(GetToken(id)),
   .5       mk-(rpats, rtypes) = if residtype = nil
   .6                            then mk-([], [])
   .7                            else mk-([mk-PatternId(residtype.id)], [residtype.type]),
   .8       tp = mk-PartialFnType(Types2Type(ParameterTypes2Types(partps) ↷ sttypes ↷ rtypes), BOOLEAN),
   .9       parms = ParameterTypes2Pats(partps) ↷ stpats ↷ rpats,
   .10      body = ONInExprSubst(subst)(oppost) in
   .11  [mk-ExplFnDef(idpost, [], tp, idpost, [parms], body, nil)];

462.0 ExtractInvDefs : (TypeDef | StateDef)* -t-> ExplFnDef*
   .1 ExtractInvDefs (defs) ≙
   .2   if defs = [] then []
   .3   else let invfns = if Is[TypeDef, TypeDef | StateDef](hd defs)
   .4                     then ExtractInvFromTpDef(hd defs)
   .5                     else ExtractInvFromStDef(hd defs) in
   .6        invfns ↷ ExtractInvDefs(tl defs);

463.0 ExtractInvFromTpDef : TypeDef -t-> ExplFnDef*
   .1 ExtractInvFromTpDef (td) ≙
   .2   if td.typeinv = nil then []
   .3   else let id = td.id,
   .4            idinv = mk-InvId(GetToken(id)),
   .5            basetype = if is-UnTaggedTypeDef(td) then td.shape
   .6                       else mk-CompositeType(id, td.fields),
   .7            tp = mk-TotalFnType(basetype, BOOLEAN),
   .8            mk-InvInitFn(p, e) = td.typeinv in
   .9        [mk-ExplFnDef(idinv, [], tp, idinv, [[p]], e, nil)];

464.0 ExtractInvFromStDef : StateDef -t-> ExplFnDef*
   .1 ExtractInvFromStDef (mk-StateDef(id, fs, stinv, -)) ≙
   .2   ExtractInvFromTpDef(mk-TaggedTypeDef(id, fs, stinv));

465.0 ExtractInitDefs : StateDef* -t-> ExplFnDef*
   .1 ExtractInitDefs (stdefs) ≙
   .2   if stdefs = []
   .3   then []
   .4   else ExtractInitDef(hd stdefs);

466.0 ExtractInitDef : StateDef -t-> ExplFnDef*
   .1 ExtractInitDef (mk-StateDef(id, -, -, stinit)) ≙
   .2   if stinit = nil then []
   .3   else let idinit = mk-InitId(GetToken(id)),
   .4            basetype = mk-Name(id),
   .5            tp = mk-TotalFnType(basetype, BOOLEAN),
   .6            mk-InvInitFn(p, e) = stinit in
   .7        [mk-ExplFnDef(idinit, [], tp, idinit, [[p]], e, nil)];
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

467.0 $UsedIds : TypeDef^* \times StateDef^* \times ValueDef^* \times FunctionDef^* \times OperationDef^* \xrightarrow{t} Id\text{-set}$

.1 $UsedIds\,(ts, ss, vs, fs, os) \triangleq$
.2 let $tm = ExtractTypeMap(ts)$,
.3 $stm = StateTypeMap(ss)$ in
.4 $\bigcup\{\{n.name\} \cup SynComps[Id](tm(n)) \mid n \in \text{dom } tm\} \cup$
.5 $\bigcup\{\{n.name\} \cup SynComps[Id](stm(n)) \mid n \in \text{dom } stm\} \cup$
.6 $\bigcup\{SynComps[Id](d) \mid d \in \text{elems } (vs \frown fs \frown os)\}$;

468.0 $StatePats : StateDef^* \times Id\text{-set} \xrightarrow{t} Pattern^* \times (OldName \xrightarrow{m} Name)$

.1 $StatePats\,(ss, used) \triangleq$
.2 if $ss = []$ then $mk\text{-}([], \{\mapsto\})$
.3 else let $mk\text{-}StateDef(i, fs, \text{-}, \text{-}) = \text{hd } ss$,
.4 $on = OldNewIds(fs, used)$ in
.5 $mk\text{-}([mk\text{-}RecordPattern(i, [\text{let } mk\text{-}(o, \text{-}) = on(i) \text{ in } mk\text{-}PatternId(o) \mid i \in \text{inds } on]),$
.6 $mk\text{-}RecordPattern(i, [\text{let } mk\text{-}(\text{-}, n) = on(i) \text{ in } mk\text{-}PatternId(n) \mid i \in \text{inds } on])],$
.7 $\{mk\text{-}OldName(n) \mapsto mk\text{-}Name(o) \mid mk\text{-}(o, n) \in \text{elems } on\})$;

469.0 $OldNewIds : Field^* \times Id\text{-set} \xrightarrow{t} (Id \times Id)^*$

.1 $OldNewIds\,(fs, used) \triangleq$
.2 if $fs = []$ then $[]$
.3 else let $mk\text{-}Field(i, \text{-}) = \text{hd } fs$ in
.4 let $mk\text{-}(o, n) : Id \times Id$ be st
.5 $o \neq n \wedge$
.6 if $i \neq \text{nil}$ then $n = i$ else $n \notin used \wedge$
.7 $o \notin used$ in
.8 $[mk\text{-}(o, n)] \frown OldNewIds(\text{tl } fs, used \cup \{o, n\})$;

470.0 $GetToken : Id \xrightarrow{t} \text{token}$

.1 $GetToken\,(id) \triangleq$
.2 cases id :
.3 $mk\text{-}ValueId(t), PreId(t), PostId(t), InvId(t), InitId(t), MkId(t), IsId(t) \rightarrow t$
.4 end

14.7.5 Classification Functions

471.0 $Is[@T_1, @T_2] : @T_2 \rightarrow \mathbb{B}$

.1 $Is\,(x) \triangleq$
.2 $\exists\, y : @T_1 \cdot x = y$;

472.0 $IsFunctionDef : ValFnOpDef \rightarrow \mathbb{B}$

.1 $IsFunctionDef\,(d) \triangleq$
.2 $\exists\, fd : FunctionDef \cdot fd = d$;

473.0 $IsPattern : PatternBind \rightarrow \mathbb{B}$

.1 $IsPattern\,(x) \triangleq$
.2 $\exists\, p : Pattern \cdot p = x$;

474.0 $Is\text{-}Expr : Expr \mid Call \rightarrow \mathbb{B}$

.1 $Is\text{-}Expr\,(x) \triangleq$
.2 $\exists\, e : Expr \cdot e = x$;

475.0 $IsType : Syn \rightarrow \mathbb{B}$

.1 $IsType\,(x) \triangleq$
.2 $\exists\, t : Type \cdot t = x$

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex A
(informative)
Extensions

A specification may use extentions to VDM-SL provided those extensions are defined either by text written the same style as this part of ISO/IEC 13817 or by defining transformations that translate the extended specification language into the specification language defined in this part of ISO/IEC 13817.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B
(informative)
Tool Conformity

A tool conforming with the requirements of this this part of ISO/IEC 13817 shall

a) be able to determine whether a specification is in error (the specification is meaningless or the specification is inconsistent), is an undefined specification or is a consistent specification and report this to the system;

b) either be a fully conforming tool and

— accept all the features of clause 10 (The Interchange Concrete Syntax) and determine any errors as specified in clause 12 (The Syntax Mapping);

NOTE — This implies that all fully conforming tools are required to accept all the features of the language specified in clause 10 (The Interchange Concrete Syntax).

— if the tool accepts an extension of the specification language, be accompanied by a document that separately describes, in the same style as the requirements specified in this part of ISO/IEC 13817, any extensions accepted by the tool or any restrictions on the language accepted by the tool and be capable of running in a mode that treats all extensions as errors;

or be a subset conforming tool and

— report to the system those parts specified in clause 10 (The Interchange Concrete Syntax) to which it conforms or be accompanied by documentation describing the same;

— report to the system those parts specified in clause 12 (The Syntax Mapping) to which it conforms or be accompanied by documentation describing the same;

NOTE — This implies that a subset conforming tool that accepts a subset of the language is required to define that subset.

NOTE — This implies that a subset conforming tool should not accept any extensions to the language except for modules (see f.)

c) report to the system those parts, if any, specified in clause 14 (The Static Semantics) by the function *wf_Spec* to which it conforms or be accompanied by documentation describing the same;

NOTE — See clause B.1 below for examples of how this is to be stated.

NOTE — This implies that a subset conforming tool that accepts a subset of the language must give the same semantics for that subset as a tool that accepts the full language.

d) report to the system those parts, if any, when it deviates from the features specified in clause 8 (The Dynamic Semantics) by the function *SemSpec* and have an accompanying document which provides details of the features specified in clause 8 (The Dynamic Semantics) by the function *SemSpec*, which it has implemented; and

e) a fully conforming tool or a subset conforming tool may accept extensions to the specification language to handle modules (see Annex C). The module extensions accepted by a tool must be documented as described above in Annex A.

B.1 Semantic conformity of tools

The table below gives examples of how a tool is to describe what it has implemented in terms of static semantics. It also shows possible levels of implementation.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Part of standard implemented (refers to clause 14 (The Static Semantics) of the standard)	Brief description
wf_Spec(*mk-DEF*({})) and *wf_Spec*(*mk-POS*({}))	This covers the use of identifiers not defined or not visible. It also checks that function types are used correctly in recursive type equations, set types, map types, the state and the arguments to imperative functions.
wf_Spec(*mk-DEF*({LOCALCONTEXT})) and *wf_Spec*(*mk-POS*({LOCALCONTEXT})) *wf_Spec*(*mk-DEF*({LOCALCONTEXT, TOTALFUNC})) and *wf_Spec*(*mk-POS*({LOCALCONTEXT, TOTALFUNC}))	Checks that a type of an expression is valid when the types of identifiers in that expression are given. Checks for the well-formedness of total functions without pre-conditions (explicit or hidden) in terms of the types of the formal parameters, the body and the result of the function.
wf_Spec(*mk-DEF*({LOCALCONTEXT, TOTALFUNC, UNIONCLOSE})) and *wf_Spec*(*mk-POS*({LOCALCONTEXT, TOTALFUNC, UNIONCLOSE}))	Uses the fact that if an expression holds for two types it holds for the union of those types.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex C
(informative)
Modules

C.1 Overview

This annex describes the features expected of a structuring mechanism for VDM-SL specifications. It also lists work in this area for VDM-SL and related specification languages thay will influence the Standard for a VDM-SL module structure.

The formal specification of any computer system would benefit from a structuring mechanism. Most modern programming languages have a mechanism for packaging the components of the system in such a manner that implementation detail can be hidden, and only the interface of a package is available to other components. The latest generation of languages, the object-oriented languages such as C^{++} and Eifel, take packaging even further and these languages allow the sharing of common implementation strategies. The temptation is to import these ideas directly into specification languages, replacing the idea of hiding implementation detail with that of hiding specification detail.

However there are two problems with this approach. The first is that the packaging mechanism used by programming languages is mainly concerned in hiding implementation detail by controlling the name-space. In a programming language identifiers denote, for example, variables, constants and procedures; and by hiding the identifier, access to the associated entity is denied. In this way implementation detail — the use of variables, constants and procedures — can be hidden from other components of the system. Though a similar mechanism would work for specification languages, a more sophisticated mechanism is more appropriate.

C.2 Requirements of Modularization

C.2.1 Language Facilities

The main requirements of a specification language module facility are:

— the textual structuring of the specification — syntactic separation;

— recorded dependencies — explicit import;

— support for information hiding — explicit export; and

— reusability within and between specifications — generic parameterization.

C.2.1.0.1 Syntactic Separation

A modularization scheme should allow the specification to be structured and presented in such a manner that the reader can understand a large system. Named units of specification should be supported by specification languages.

C.2.1.0.2 Explicit Import

The environment in which the body of a module is to be interpreted should be determined from a collection of references to other modules.

C.2.1.0.3 Explicit Export

It should be possible to restrict the use of construct defined in a module.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

C.2.1.0.4 Parameterization

It should be possible to instantiate a module with specific properties.

C.2.2 Semantics

The main incentive to adding a module structure to a specification language is to provide a facility to partition a specification into separate components (modules), that would allow separate reasoning about each individual component. This facility would be useful for several reasons. Firstly, it would be possible to establish libraries of specifications which could be reused as solutions to different problems. Secondly, the size of each component could be kept sufficiently small so that reasoning about individual components will be simplified — and if it is easy to reason about, it will be easy to understand. There is also, for certain types of system, the possibility of refining one or more modules to executable code independently of the rest of the specification.

It should be emphasized that modularisation is a mechanism for organising specifications; and does not imply a computational model, for example concurrency or object-orientation. Nor does it necessarily define a packaging mechanism for the implementation.

The semantics should be defined in such a way that the meaning of an individual module may be determined, since this determines separate reasoning.

C.3 Extant Approaches to Structuring Specifications

There are many actual approaches to providing a module structure to specification languages that could influence VDM-SL. These are briefly described below[1)]

C.3.1 The Syntactic Approach

This proposal for adding modules to the VDM specification language is to provide a form of "macro expansion" — the module structure is removed by syntactic transformations. There is a renaming convention for identifiers used in each (separate) module; the state components of the modules are "smashed" together to produce a flat state. The renaming convention together with the flat state allow the module walls to be removed leaving a flat VDM-SL specification which can be given semantics in the usual manner.

C.3.2 The Z Approach

The Z approach to modularization is provided by the schema calculus. It provides an elegant way of structuring a specification and providing textual separation to make the specification more understandable; however it provides no hiding mechanisms at all. The semantics for this facility are provided by the flat language, the Z specification is expanded by a macro replacement strategy to provide a flat specification and thus the semantics can be deduced.

C.3.3 The Manchester Approach (Fitzgerald & Jones)

The work of Fitzgerald and Jones is aimed at determining what modular structuring facilities are appropriate to model-oriented specification. It based on two main principles: that the choice of structuring mechanisms should be justified by practical necessity and that such mechanisms should allow formal reasoning to proceed separately for each module in a specification.

Several alternative modular structures for a specification of a database system are presented in FJ90. The structuring facilities used are tentative and have no formal semantics. Responses to the challenge presented in that paper can be

[1)] Input to some of this annex was provided by various researchers in the area

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

found in Fei90 and Den92 where the same structuring problem is addressed in COLD-K, VVSL, RSL, Z and Extended ML.

This work is taken further in Fit91a, where a series of case studies motivates the definition of a collection of specification modularization facilities. Their meaning is given in terms of structuring the logical theories associated with individual modules, with the intentiion of allowing formal reasoning to proceed separately for each. Module import, information hiding via export clauses and a form of parameterisation are supported. The thesis also describes the application of these structuring facilities to a VDM-SL-like flat language. A later paper (Fit91b) presents a condensed version of a case study applying these techniques.

C.3.4 The VVSL Approach (Middelburg)

The approach to modular structuring adopted for VVSL, a specification language which incorporates the version of VDM-SL used in Jon90 and Fit91a, deviates somewhat from established approaches. Firstly, the meaning of a module is a theory presentation. It has this in common with the approach of the Larch Shared Language (GH86). In other approaches, its meaning is usually more abstract – viz. a theory or a model class. Secondly, the origins of names are taken into account in the treatment of name clashes in the composition of modules. It has this in common with the approach of Clear (BG80). In other approaches, name clashes are usually treated in an ad hoc way.

The mathematical basis used for the modular structuring facilities of VVSL consists of an algebraic model for modularization of specifications, called Description Algebra (Jon89), and a variant of classical lambda calculus, called $\lambda\pi$-calculus (Fei89), for parameterization of specifications. A comprehensive exposition of this mathematical basis, which is not overly mathematical, is given in Mid92. Mathematically precise definitions can be found in Mid93 as well as in Jon89 and Fei89.

The semantics of VVSL describes the meaning of modularly structured specifications as terms from the instance of $\lambda\pi$-calculus for a particular subalgebra of DA extended with higher-order generalizations of the operations of that algebra. The building blocks of these terms are the constants of the subalgebra of DA concerned. These constants are essentially theory presentations.

This semantics is a suitable starting-point for devising proof rules that allow theorems about a module to be inherited from the modules from which it has been constructed. Such proof rules naturally suggest general proof strategies which exploit the modular structure of specifications, which matters to the issue of formal correctness proofs of design steps (i.e. verified design). Besides, they enable compositional development of theories about modules, which seems essential to the issue of module reusability. If efficiency is an issue, it seems rarely possible to maintain the modular structure of a specification in the ultimate software system. This justifies the supply of conversion rules which allow to transform a specification to another specification with a different modular structure in a meaning preserving way. The semantics is also a suitable starting-point for devising such conversion rules.

C.3.5 The RAISE Approach

The RAISE approach to modules identified the following requirements for a module structure in a specification language; it must be possible to:

a) decompose specifications/modules into modules;

b) use the module structure to control the name space;

c) use modules to control the 'theory space' — separate reasoning should be possible;

d) develop modules separately using a set of methodological rules that allow integration with known properties;

e) develop modules in a step-wise fashion;

f) distinguish between sharing and copying of sub-modules/theories;

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

g) re-use standard modules with a choice of one of their refinements;

h) do maintenance by redeveloping only some modules;

i) develop into a modular structure from a specification that differs from the original structure of the specification;

j) express simple extensions for example, adding definitions syntactically

As for actual language design, it is important to have orthogonality of module concepts. Expressiveness at the module level is much less important than at the expression level, while confusion about similar module concepts can occur easily.

The RAISE Specification Language (RSL) described in Gro92 is based on the class expression. In its basic form the class expression consists of declarations, which may be of schemes, objects, types, values, variables, channels and axioms. The semantics of a class expression is a class of models. A scheme is a named and possibly parameterised class, and an object is a named arbitrary model from a class. The semantics of RSL are defined in Mil90, and a more informal introduction may be found in BD92.

Scheme parameters are objects and (unlike some languages) there are no limitations on the classes of these objects. Scheme parameterisation can therefore be used for defining reusable modules.

Indexed arrays of objects may be defined: these are very useful for specifying some kinds of system in which arbitrary replication of entities (particularly concurrent processes) is needed. The array indexes may be values of any type.

Different objects (including different objects in arrays) are different models of their class expressions, and names of entities from objects are qualified (prefixed) with the object names. This means that objects allow one to distinguish between copying and sharing.

Apart from basic class expressions, class expressions may be formed by renaming names from a class expression, hiding names in a class expression, extending one class expression with another and by instantiating schemes. Renaming and hiding give control over the name space; extension gives a simple form of inheritance and a syntactic means of adding definitions and getting immediate refinement. Scheme instantiation allows for a richer form of inheritance, since a class expression can contain instantiations of schemes as objects (or arrays of objects), in which case the class expression inherits the entities (with their properties) of the instantiated schemes. This also means that theories about modules can be structured in terms of theories about their components, and that refinement relations between modules decompose into refinement relations between their components, including refinement relations between the modules on which they depend.

Refinement is based semantically on sub-classing of models and restriction. Class expression B refines class expression A if all the models of B, restricted to the signature of A, are models of A. This semantic characterisation of refinement has a convenient and equivalent logical characterisation: class expression B refines class expression A if the signature of B includes that of A and if the theory of A is provable in B. The signature condition is statically decidable, and hence can be checked by a tool. The theory condition produces a set of axioms expressible in RSL, which can be generated by a tool.

This notion of refinement allows separate development of modules because it allows for substitution. If a class contains (depends on) an instantiation of a scheme and the scheme is refined then one can substitute in the dependent class an instantiation of the new scheme for the old, and the resulting new class containing the scheme instantiation will automatically refine the old one. So "horizontal" development (combining modules) composes with "vertical" development (refinement).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex D
(informative)
Cross-references

D.1 Cross References for the Dynamic Semantics

This annex first presents the naming and typesetting conventions which are used in the definition of the dynamic semantics for VDM-SL. Then an alphabetically sorted list of functions and predicates are given. Each function name in this list is prefixed with its number in the dynamic semantics and postfixed by the numbers of functions where it is used.

D.1.1 Naming and Typesetting Conventions Used

Function names are written with capital letters starting each new word (or an abbreviation of the word), e.g. *VerifyValues*. As explained in the previous clause, the definition will contain *Expand*-functions, *Verify*-predicates, and *Eval*-functions. The different auxiliary functions/predicates have (as far as possible) also been given prefixes according to their functional task. Selector functions which take a piece of syntax and select a part of that syntax have their names prefixed with *Sel*. Collector functions, which take a structure of syntax and return a structure with a part of each component of the argument syntax, have their names prefixed by *Col*. Functions which take a piece of syntax and create a new piece of syntax have their names prefixed by *Get*. Those functions which take a syntactic entity and make a semantic entity (which are not directly the semantics of the argument syntax) have their names prefixed by *Make*. Those functions which take a combination of syntactic entities and semantic entities and yields a new semantic entity have their names prefixed by *Comp*. Finally, the functions which take a semantic entity and generate a new semantic entity have their names prefixed by *Gen*. It is also a convention that all definitions that have their names prefixed with *Is* are predicates.

Variable names are written with lower-case letters and underscores, e.g. *env_s*. In order to increase the readability of the definition a number of conventions have been used for variable names as well. In this way it is easier to read the definition when the variable names indicate what kind they are (i.e. to what domain they belong). Following is a list of three groups of conventions which are commonly used as variables[1)]:

Abbreviation	Explanation
ev	Evaluator
l	Loose
lev	Loose evaluator
p	Pattern
pev	Pattern evaluator
tf	Transformer
ob	Object
den	Denotation
d	Domain
env	Environment
_s	Set of ...
_m	Map of ...
_l	List of ...
_p	Pair of ...
mul	Multi-...

In the abstract syntax small capital letters like THIS are used for terminal symbols. In the definition, sets of such terminal symbols have been named by large capital letters such as HERE. The mathematical functions which are assumed to be "built-in" have been underlined such as here. A short definition of these special symbols of our mathematical notation is presented in clause 5.

[1)]The first group contains general abbreviations of different concepts, the second group contains suffixes and the last group contains prefixes.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

When a composite object is split into its components by means of pattern matching only the components which are actually used are mentioned. The components which are not used are indicated by an anonymous 'don't-care' variable labelled by '_' (underscore)[2)].

D.1.2 Listing of Functions/Predicates: Alphabetic (uses)

```
243: AllPairComb              90
 47: ApplyCaseExprAltn        46
 46: ApplyCasesExpr           45 46
120: ApplyCasesStmt           119 120
121: ApplyCaseStmtAltn        120
170: ApplyCurFn               31 170
229: ApplySeq                 54 186 229
236: ApprList                 235 236
163: AssertStateConstancy     154
232: BotEnv                   40 42 43 112 113 114 117 150
 65: CheckMembership          57
164: CheckStateConstancy      37 38
165: CheckTagEnv              25 27 28 37 38 97 98 115 144 151 158 159
                              182 183
231: Choose                   22 231
146: ChopSeqVal               142
145: ChopSetVal               140
196: ColIdList                27 28 155 156 196
195: ColIdSet                 3 25 41 112 113 114 121 123 131 132 195
200: ColInvFns                5
197: ColParList               27 28
198: ColPars                  197 198
193: ColRecs                  153 165 193
201: ColSelMap                172
194: ColTypeList              8 27 28 193 194
199: ColTypeVars              27 29 32 183
240: Compatible               26 56 70 80 81 85 90 91 139 140 142 147
                              153 165 193
171: CompFieldSelectors       153
 72: Compose                  57
173: CompRecordConstructors   153
174: CompRecordModifiers      153
172: CompRecordSel            171 174
  3: DisjointIds              2
123: DoSeqLoop                122 123 124 125
168: EqCurDoms                31 168
 63: EqualOpr                 57
 79: EvalAllOrExistsExpr      39
130: EvalAlways               111
 95: EvalApplyExpr            39
117: EvalAssign               111
  7: EvalBasicType            6
 57: EvalBinaryExpr           39
151: EvalBind                 42 80 81 82 83 85 88 91 113
115: EvalBlock                111
148: EvalBotPat               137 138 139 140 142 147 148
128: EvalCall                 111 114 115 117
 45: EvalCasesExpr            39
```

[2)] This convention is also used in languages like Hope, ML and Prolog.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
119: EvalCasesStmt           111
  8: EvalCompositeType       6 153
 43: EvalDefExpr             39
114: EvalDefStmt             111
 81: EvalExists              79
 82: EvalExistsUniqueExpr    39
133: EvalExit                111
 26: EvalExplDef             5 25 28 41
 38: EvalExplOp              35
 39: EvalExpr                26 27 28 37 38 40 42 43 44 45 47 48 57
                             80 81 82 83 84 85 86 87 88 89 90 91 92
                             93 94 95 96 98 99 107 110 113 114 115 117
                             118 119 122 124 125 126 128 129 133 136
                             143 151 179 180 181
 97: EvalFctTypeInstExpr     39
 96: EvalFieldSelectExpr     39
 17: EvalFnType              6
 80: EvalForAll              79
 15: EvalGeneralMapType      6
102: EvalId                  39
 44: EvalIfExpr              39
118: EvalIfStmt              111
 37: EvalImplOp              36
125: EvalIndexForLoop        111
 16: EvalInjectiveMapType    6
 83: EvalIotaExpr            39
 99: EvalIsExpr              39
 98: EvalLambda              39
 42: EvalLetBeSTExpr         39
113: EvalLetBeSTStmt         111
 40: EvalLetExpr             39
112: EvalLetStmt             111
101: EvalLiteralExpr         39
 41: EvalLocalDef            40 112
 91: EvalMapComprehension    39
 90: EvalMapEnumeration      39
136: EvalMatchVal            134
127: EvalNonDetStmt          111
100: EvalOldId               39
 11: EvalOptionalType        6
134: EvalPattern             26 43 114 139 140 142 143 144 147 150 151
135: EvalPatternId           134
147: EvalPatternList         137 138 141
 32: EvalPolyDef             27
 33: EvalPolyType            32 183
  9: EvalProductType         6
 20: EvalQuoteType           6
 93: EvalRecordConstructor   39
 94: EvalRecordModifier      39
138: EvalRecordPattern       134
132: EvalRecTrapStmt         111
129: EvalReturnStmt          111
 13: EvalSeq0Type            6
 14: EvalSeq1Type            6
 88: EvalSeqComprehension    39
142: EvalSeqConcPattern      134
 87: EvalSeqEnumeration      39
```

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

141: EvalSeqEnumPattern 134
122: EvalSeqForLoop 111
116: EvalSequence 111 116 127
85: EvalSetComprehension 39
143: EvalSetConstrPattern 134
84: EvalSetEnumeration 39
139: EvalSetEnumPattern 134
124: EvalSetForLoop 111
86: EvalSetRange 39
12: EvalSetType 6
140: EvalSetUnionPattern 134
111: EvalStmt 38 112 113 114 115 116 118 121 123 126 130 131 132
89: EvalSubSequence 39
131: EvalTrapStmt 111
92: EvalTupleConstructor 39
137: EvalTuplePattern 134
6: EvalType 5 11 12 13 14 15 16 17 21 22 23 25 27 28 33 37 38 98 99 115 144 151 158 159 180
144: EvalTypeConstrPattern 134
5: EvalTypeDef 4
18: EvalTypeId 6 24
21: EvalTypeList 8 9 21 27 28 97
23: EvalTypeMap 182
22: EvalTypeSet 10 22
19: EvalTypeVar 6
48: EvalUnaryExpr 39
10: EvalUnionType 6
126: EvalWhileLoop 111
152: ExpandImplDefs 2
153: ExpandImplRecTypeDefs 152
154: ExpandImplStateDef 152
160: ExtDomEnv 27 33 159 179
156: ExtExplOpEnv 38
158: ExtImplFnEnv 157
157: ExtImplFnsEnv 30
155: ExtImplOpEnv 37
159: ExtImplPolyFnEnv 29
162: ExtLocDomEnv 115
161: ExtValEnv 31
245: False 29 30 50 62 63 64 65 66 80 81 82 99 101 126 163 164
77: GCD 76
177: GenConstructorFn 173
178: GenPolyApply 29
175: GenRecordModFn 174
176: GenSelectorFn 172
186: GetExpr 83 186
184: GetFnApp 30
192: GetRecConsId 93 173
191: GetRecModId 94 106 174
190: GetRecSelId 96 109 172 175
187: GetStateDom 24 35 36 37 38 163 164
188: GetStateTypeId 24 104 105 128 155 156 163 164 187
189: GetStateValId 105 128 155 156
185: GetType 27 28 32 182 183
225: Injective 16 55 139

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
 59: IntDiv                  57
230: Iota                    5 29 49 59 60 61 77 83 88 115 175 188
  2: IsAModelOf              1
226: IsCont                  5 26 32
227: IsCpo                   10
169: IsCurFunVal             168 169
 78: IsOdd                   76
 73: Iterate                 57
 74: IterateFct              73 74
 75: IterateMap              73 75
 64: LogOpr                  57
108: LookUp                  106 107 109 110 128
109: LookUpFieldRef          108
110: LookUpMapOrSeqRef       108
181: MakeEvalMapSet          29 30
182: MakeFns                 30 41
180: MakeFuncAbs             24
179: MakePolyFnDef           32
183: MakePolyFns             29
 56: MapDi                   48
 70: MapMerge                57
 69: MapOrSeqMod             57
 71: MapToSetOpr             57
 55: MapXx                   48
216: MkTag                   2 3 14 16 20 24 28 37 38 41 49 51 52 53
                             54 55 56 58 59 60 61 67 68 69 70 71 72
                             73 84 85 86 87 88 90 91 92 95 98 101 104
                             105 107 110 116 125 127 137 138 145 146
                             148 151 153 154 155 156 158 159 162 163
                             164 165 167 168 169 170 177 178 180 184
                             185 186 189 190 191 192 193 194 195 196
                             197 198 200 202 203 204 205 206 207 214
                             215 216 219 220 221 222 223 224 234 244
                             245
103: Modify                  106 107 115 117 128
106: ModifyFieldRef          103
107: ModifyMapOrSeqRef       103
105: ModifyState             104
104: ModifyValueId           103
235: NonDetIter              126 132
150: NonErrPat               47 121 123 131 132
 50: Not                     48
 58: NumBin                  57
 62: NumComp                 57
 61: NumMod                  57
 60: NumRem                  57
 76: OddDenom                73
241: Partition               80 81 85 91
242: Permutations            127
 34: PolyVals                32
239: PropE                   27 28 42 45 47 82 83
237: PropErr                 32 41 106 107
238: PropS                   113 117 119 121 122 123 124 125 128 131
                             132
149: PropToLPatEval          139 140 142
167: ResLocId                115
166: Restore                 112 113 114 121 123 131 132
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
228: Reverse                122 228
205: SelBody                26
214: SelExtVarInfId         163
213: SelExtVarInfMode       163
211: SelInvExpr             200
210: SelInvId               5 200
206: SelKind                27 28
215: SelPattern             83
208: SelPolyFn              32
203: SelPost                29 30
202: SelPre                 27 28 29 30
204: SelRes                 30
212: SelSel                 201
209: SelShape               5 153 200
207: SelTypeVars            27
  1: SemSpec
 68: SeqConc                57
 54: SeqDi                  48
 53: SeqXx                  48
 67: SetBin                 57
 52: SetDi                  48
233: SetToSeq               124 233
 51: SetXx                  48
220: ShowRecTagVal          94 96 99 106 109
217: ShowTag                3 6 39 99 103 108 111 114 115 117 134 151
                            185 188 198 202 203 204 219 220 221 222
                            223 224
234: StepsToSeq             125 234
223: StripMapTagVal         55 56 69 70 71 72 73
224: StripNumTagVal         49 58 59 60 61 62 73 86 89 125
219: StripRecTagVal         176 177
221: StripSeqTagVal         53 54 68 69 89 122 147
222: StripSetTagVal         51 52 56 65 66 67 71 124 139 143
218: StripTag               99 101 163 190 199 219 220 221 222 223
                            224
 66: SubSet                 57
244: True                   5 29 30 31 37 38 42 44 50 62 63 64 65 66
                            80 81 82 83 85 88 91 99 101 113 118 163
 49: UNum                   48
 31: VerifyCurFn            27 28
 28: VerifyExplFns          2 40 112
 35: VerifyExplOps          2
 27: VerifyExplPolyFns      2
 30: VerifyImplFns          2 40 41 112
 36: VerifyImplOps          2
 29: VerifyImplPolyFns      2
 24: VerifyStateDef         2
  4: VerifyTypes            2
 25: VerifyValues           2
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

D.2 Cross References for the Concrete Syntax

all expression, 182
always statement, 185
and, 182
applicator, 189
apply, 183, 189
arithmetic abs, 181
arithmetic divide, 182
arithmetic infix operator, 189
arithmetic integer division, 182
arithmetic minus, 182
arithmetic mod, 182
arithmetic multiplication, 182
arithmetic plus, 182
arithmetic prefix operator, 189
arithmetic rem, 182
assign statement, 184
assignment definition, 184

basic type, 178
binary expression, 181
binary operator, 181
bind, 185
bind list, 185
block statement, 184
boolean literal, 188
bracketed expression, 180
bracketed type, 178

call statement, 185
cases expression, 180
cases expression alternative, 180
cases expression alternatives, 180
cases statement, 184
cases statement alternative, 184
cases statement alternatives, 184
character, 186
character literal, 188
combinator, 189
comment, 188
composite type, 178
composition, 182, 189
compound delimiter, 188
connective, 190

dcl statement, 184
def bind, 180
def expression, 180
def statement, 184
definition block, 178
delimiter, 188
delimiter character, 186
digit, 186
discretionary type, 178
distinguished letter, 186
distributed map merge, 181
distributed sequence concatenation, 181
distributed set intersection, 181
distributed set union, 181
document, 178

elseif expression, 180
elseif statement, 184
equal, 182
equals definition, 184
error, 179
error list, 179
evaluator, 189
exceptions, 179
exists expression, 182
exists unique expression, 182
exit statement, 185
explicit function definition, 179
explicit operation definition, 179
exponent, 188
expression, 180
expression list, 180
externals, 179

field, 178
field list, 178
field reference, 183
field select, 183, 189
finite power set, 181
floor, 181
function definition, 179
function definitions, 179
function type, 178
function type instantiation, 183, 189

general map type, 178
greater than, 182
greater than or equal, 182
Greek letter, 186

identifier, 188
identifier type pair, 179
identity statement, 185
if expression, 180
if statement, 184
implicit function definition, 179
implicit operation definition, 179
imply, 182
in set, 182
index for loop, 184
initialization, 179
injective map type, 178
invariant, 179
invariant initial function, 179
iota expression, 183
is basic type, 183
is basic type expression, 183
is defined type expression, 183

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

is expression, 183
iterate, 182, 189

keyword, 188
keyword letter, 186

lambda expression, 183
less than, 182
less than or equal, 182
let be expression, 180
let be statement, 184
let expression, 180
let statement, 184
local definition, 184
logical equivalence, 182
logical infix operator, 190
logical prefix operator, 190

map comprehension, 183
map domain, 181
map domain restrict by, 182
map domain restrict to, 182
map enumeration, 183
map infix operator, 189
map inverse, 189
map inverse expression, 181
map merge, 182
map or sequence modify, 182
map or sequence reference, 183
map prefix operator, 189
map range, 181
map range restrict by, 182
map range restrict to, 182
map type, 178
maplet, 183
match value, 185
mode, 179
multiple bind, 185
multiple set bind, 185
multiple type bind, 185

name, 183
name list, 183
nil literal, 188
nondeterministic statement, 184
not, 181
not equal, 182
not in set, 182
numeral, 188
numeric literal, 188

old name, 183
operation definition, 179
operation definitions, 179
operation type, 179
optional type, 178
or, 182
other character, 186
others expression, 180
others statement, 184

parameter types, 179
parameters, 179
parameters list, 179
partial function type, 178
pattern, 185
pattern bind, 185
pattern identifier, 185
pattern list, 185
pattern type pair, 179
pattern type pair list, 179
plain letter, 186
prefix expression, 181
product type, 178
proper subset, 182

quantified expression, 182
quote literal, 188
quote type, 178

record constructor, 183
record modification, 183
record modifier, 183
record pattern, 185
recursive trap statement, 185
relation, 190
relational infix operator, 190
return statement, 185

separator, 188
seq conc pattern, 185
seq enum pattern, 185
seq type, 178
seq0 type, 178
seq1 type, 178
sequence comprehension, 183
sequence concatenate, 182
sequence elements, 181
sequence enumeration, 183
sequence for loop, 184
sequence head, 181
sequence indices, 181
sequence infix operator, 189
sequence length, 181
sequence prefix operator, 189
sequence tail, 181
set bind, 185
set cardinality, 181
set comprehension, 183
set difference, 182
set enum pattern, 185
set enumeration, 183
set for loop, 184
set infix operator, 189
set intersection, 182

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

set prefix operator, 189
set range expression, 183
set relational operator, 190
set type, 178
set union, 182
set union pattern, 185
state definition, 179
state designator, 183
statement, 184
subsequence, 183, 189
subset, 182
symbolic literal, 188

tagged type definition, 178
text literal, 188
total function type, 178
trap, 185
trap statement, 185
traps, 185
tuple constructor, 183
tuple pattern, 185
type, 178
type bind, 185
type bind list, 185
type definition, 178
type definitions, 178
type name, 178
type variable, 178
type variable identifier, 178
type variable list, 179

unary expression, 181
unary minus, 181
unary operator, 181
unary plus, 181
union type, 178
untagged type definition, 178

value definition, 179
value definitions, 179
var information, 179

while loop, 184

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

D.3 Cross References for the Abstract Syntax

AllExpr, 202
AlwaysStmt, 206
Apply, 203
AssignDef, 205
AssignStmt, 205

BasicType, 196
BinaryExpr, 201
BinaryOp, 202
Bind, 207
BindList, 207
BlockStmt, 205
BoolLit, 208
BracketedExpr, 200
BracketedType, 196

Call, 206
CaseAltn, 201
CaseExprAlternatives, 201
CasesExpr, 201
CasesStmt, 205
CasesStmtAlternatives, 205
CaseStmtAltn, 205
CharLit, 209
CompositeType, 196

DclStmt, 205
DefBind, 200
DefExpr, 200
DefinitionBlock, 196
DefStmt, 205
DiscretionaryType, 197
Document, 196

ElsifExpr, 200
ElsifStmt, 205
EqDef, 205
Error, 199
ExistsExpr, 202
ExistsUniqueExpr, 202
ExitStmt, 206
ExplFnDef, 198
ExplOprtDef, 198
Expr, 200
ExprList, 199

FctTypeInst, 203
Field, 196
FieldList, 196
FieldRef, 204
FieldSelect, 203
FnType, 197
FunctionDef, 198
FunctionDefinitions, 198

GeneralMapType, 197

Id, 208
IdentStmt, 206
IdType, 198
IfExpr, 200
IfStmt, 205
ImplFnDef, 198
ImplOprtDef, 199
IndexForLoop, 206
Initialization, 197
InitId, 208
InjectiveMapType, 197
Invariant, 197
InvId, 208
InvInitFn, 197
IotaExpr, 202
IsBasicType, 204
IsBasicTypeExpr, 204
IsDefTypeExpr, 204
IsExpr, 204
IsId, 208

Lambda, 204
LetBeSTExpr, 200
LetBeSTStmt, 205
LetExpr, 200
LetStmt, 205
Literal, 208
LocalDef, 205

MapComprehension, 203
MapEnumeration, 203
MapInverseExpr, 201
Maplet, 203
MapOrSeqRef, 204
MapType, 197
MatchVal, 207
MkId, 208
Mode, 199
MultBind, 207
MultSetBind, 207
MultTypeBind, 208

Name, 204
NameList, 204
NilLit, 208
NonDetStmt, 206
NumLit, 208

OExceptions, 199
OExprList, 199
OExternals, 199
OldName, 204
OPatList, 207
OPatTypePairList, 198
OperationDef, 198
OperationDefinitions, 198
OptionalType, 197

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

OpType, 199
OthersExpr, 201
OthersStmt, 205
OTypeVarList, 198

Parameters, 198
ParametersList, 198
ParameterTypes, 198
PartialFnType, 197
PatList, 207
Pattern, 207
PatternBind, 207
PatternId, 207
PatTypePair, 198
PostId, 208
PrefixExpr, 201
PreId, 208
ProductType, 197

QuantExpr, 202
QuoteLit, 209
QuoteType, 196

RecordConstructor, 203
RecordModification, 203
RecordModifier, 203
RecordPattern, 207
RecTrapStmt, 206
ReturnStmt, 206

Seq0Type, 197
Seq1Type, 197
SeqComprehension, 203
SeqConcPattern, 207
SeqEnumeration, 203
SeqEnumPattern, 207
SeqForLoop, 206
SeqType, 197
SetBind, 207
SetComprehension, 203
SetEnumeration, 202
SetEnumPattern, 207
SetForLoop, 206
SetRange, 203
SetType, 197
SetUnionPattern, 207
StateDef, 197
StateDesignator, 204
Stmt, 204
SubSequence, 203

TaggedTypeDef, 196
TextLit, 209
TotalFnType, 197
Trap, 206
Traps, 206
TrapStmt, 206
TupleConstructor, 203
TuplePattern, 207
Type, 196
TypeBind, 207
TypeBindList, 208
TypeDef, 196
TypeDefinitions, 196
TypeName, 197
TypeVar, 197
TypeVarId, 197

UnaryExpr, 201
UnaryOp, 201
UnionType, 197
UnTaggedTypeDef, 196

ValueDef, 198
ValueDefinitions, 198
ValueId, 208
VarInf, 199

WhileLoop, 206

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

D.4 Cross References for the Syntax Mapping

AllExpr, 202
AllExpr2AllOrExistsExpr, **241**
AlwaysStmt, 206
AlwaysStmt2Always, 246, **250**
Apply, 203
Apply2Apply, 238, **244**
AssignDef, 205
AssignStmt, 205
AssignStmt2Assign, 246, **248**

BasicType, 196
BinaryExpr, 201
BinaryExpr2BinaryExpr, 238
BinaryOp, 202
Bind, 207
Bind2Bind, 240–242, 247, **252**, 253
BindList, 207
BindList2BindSet, 241–243, **252**
BlockStmt, 205
BlockStmt2BlockOrSequence, 246, **248**, 248, 250, 251
BoolLit, 208
BracketedExpr, 200
BracketedType, 196

Call, 206
Call2Call, 246–248, **250**
CaseAltn, 201
CaseAltn2CaseAltn, 240, **240**
CaseExprAlternatives, 201
CasesExpr, 201
CasesExpr2CasesExpr, 238, **240**
CasesStmt, 205
CasesStmt2CasesStmt, 246, **248**
CasesStmtAlternatives, 205
CaseStmtAltn, 205
CaseStmtAltn2CaseStmtAltn, **249**, 249
CharLit, 209
CompositeType, 196
CompositeType2CompositeType, 222, **222**, 222

DclStmt, 205
DefBind, 200
DefExpr, 200
DefExpr2DefExpr, 238, 240, **240**, 240
DefinitionBlock, 196
DefinitionBlocks2ExplFnDefMap, 217, **218**
DefinitionBlocks2ExplOpDefMap, 217, **219**
DefinitionBlocks2ExplPolyFnDefMap, 217, **218**
DefinitionBlocks2ImplFnDefMap, 217, **218**
DefinitionBlocks2ImplOpDefMap, 217, **219**
DefinitionBlocks2ImplPolyFnDefMap, 217, **218**
DefinitionBlocks2StateDef, 217, **220**
DefinitionBlocks2TypeDefMap, 217, **217**
DefinitionBlocks2ValDefMap, **219**
DefStmt, 205
DefStmt2DefStmt, 246, **247**

DiscretionaryType, 197
Document, 196
Document2Definitions, **217**

ElsifExpr, 200
ElsifStmt, 205
EqDef, 205
Error, 199
ExistsExpr, 202
ExistsExpr2AllOrExistsExpr, **241**
ExistsUniqueExpr, 202
ExistsUniqueExpr2ExistsUniqueExpr, 238, **241**
ExitStmt, 206
ExitStmt2ExitStmt, 246, **251**
ExplFnDef, 198
ExplFnDef2ExplFnDef, 226, 228, **228**, 229, 231, 232, 235
ExplOprtDef, 198
ExplOprtDef2ExplOpDef, 232, **233**
Expr, 200
Expr2Expr, 225, 226, 236, 237, **237**, 238–241, 243, 244, 246–252
ExprList, 199

FctTypeInst, 203
FctTypeInst2FctTypeInst, 238, **244**
Field, 196
Field2Field, **222**, 222, **222**
FieldList, 196
FieldRef, 204
FieldSelect, 203
FieldSelect2FieldSelect, 238, **244**
FnType, 197
FnType2FnType, 222, **223**, 228
FunctionDef, 198
FunctionDef2ExplFnDefMap, 227, **227**, 239, 247
FunctionDef2ExplPolyFnDefMap, 225, **225**
FunctionDef2ImplFnDefMap, 231, **231**, 239, 247
FunctionDef2ImplPolyFnDefMap, 228, 229, **229**
FunctionDefinitions, 198
FunctionDefinitions2ExplFnDefMap, 218, **227**
FunctionDefinitions2ExplPolyFnDefMap, 218, **225**
FunctionDefinitions2ImplFnDefMap, 219, **231**
FunctionDefinitions2ImplPolyFnDefMap, 218, **228**

GeneralMapType, 197
GeneralMapType2GeneralMapType, 222, **223**
GetUnusedId, **216**, 221, 226, 230, 233, 235, 244

Id, 208
Id2Id, 221, 222, 224, 226, 228–233, 235, 236, 238, 243–246, 248–252, **253**
IdentStmt, 206
IdType, 198
IfExpr, 200
IfExpr2IfExpr, 238, 240, **240**
IfStmt, 205

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

IfStmt2IfStmt, 246, **248**, 248
ImplFnDef, 198
ImplFnDef2ImplFnDef, 229, **229**, 231
ImplOprtDef, 199
ImplOprtDef2ImplOpDef, 235, **235**
IndexForLoop, 206
IndexForLoop2IndexForLoop, 246, **249**
Initialization, 197
InitId, 208
InjectiveMapType, 197
InjectiveMapType2InjectiveMapType, 222, **223**
Invariant, 197
InvId, 208
InvInitFn, 197
InvInitFn2Lambda, **221**, 221, **221**, 224
IotaExpr, 202
IotaExpr2IotaExpr, 238, **241**
IsBasicType, 204
IsBasicTypeExpr, 204
IsBasicTypeExpr2IsExpr, 238, **245**
IsDefTypeExpr, 204
IsDefTypeExpr2IsExpr, 238, **245**
IsExpr, 204, **238**
IsId, 208
IsPolymorphic, 225, **225**, 225, 227–229, 231, 239, 247

Lambda, 204
Lambda2Lambda, **244**
LetBeSTExpr, 200
LetBeSTExpr2LetBeSTExpr, 238, **239**
LetBeSTStmt, 205
LetBeSTStmt2LetBeSTStmt, 246, **247**
LetExpr, 200
LetExpr2LetExpr, 221, 238, **239**
LetStmt, 205
LetStmt2LetStmt, 246, **247**
Literal, 208
LocalDef, 205

MakeDisjExcps, **236**, 236, **236**, 237
MakeLambda, **226**, 226, 228
MakeLetExpr, 221, **221**, 221, 230, 244
MakeOpHeading, 233, **233**, 236
MakePostDisjExcps, **236**, 236
MapComprehension, 203
MapComprehension2MapComprehension, 238, **243**
MapEnumeration, 203
MapEnumeration2MapEnumeration, 238, **242**
MapInverseExpr, 201
MapInverseExpr2UnaryExpr, 238, **241**
Maplet, 203
Maplet2Maplet, 242, **242**, 243
MapOrSeqRef, 204
MapType, 197
MatchVal, 207
MkId, 208
Mode, 199

MultBind, 207
MultSetBind, 207
MultTypeBind, 208

Name, 204
Name2Id, 244
NameList, 204
NilLit, 208
NonDetStmt, 206
NonDetStmt2NonDetStmt, 246, **250**
NumberOfParsMatchesFnType, 226, **227**, 227
NumberOfParsMatchesOpType, 232, **232**, 232
NumLit, 208

OExceptions, 199
OExpr2TrueOrExpr, 228, 233, **237**, 240, 242, 243, 247
OExprList, 199
OExternals, 199
OldName, 204
OldName2OldId, 238, **245**
OPatList, 207
OPatTypePairList, 198
OperationDef, 198
OperationDef2ExplOpDefMap, **232**, 232, **232**
OperationDef2ImplOpDefMap, 234, **234**
OperationDefinitions, 198
OperationDefinitions2ExplOpDefMap, 219, **232**
OperationDefinitions2ImplOpDefMap, 219, 220, **234**
OptionalType, 197
OptionalType2OptionalType, 222, **223**
OpType, 199
OthersExpr, 201
OthersStmt, 205
OTypeVarList, 198
OTypeVarList2TypeVarList, **227**, 228, 230

Parameters, 198
ParametersList, 198
ParameterTypes, 198
PartialFnType, 197
PatList, 207
Pattern, 207
Pattern2Pattern, 225, 226, 236, 240, 244, 249, 251, **251**, 252
PatternBind, 207
PatternBind2Pattern, 240, 247, 249, **252**
PatternId, 207
PatternId2PatternId, **251**, 251, **251**
PatTypePair, 198
PostId, 208
PrefixExpr, 201
PrefixExpr2UnaryExpr, 238, **241**
PreId, 208
ProductType, 197
ProductType2ProductType, 222, **223**, 230, 244

QuantExpr, 202
QuoteLit, 209

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

QuoteType, 196
QuoteType2QuoteType, 222, **222**

RecordConstructor, 203
RecordConstructor2RecordConstructor, 238, **243**
RecordModification, 203
RecordModifier, 203
RecordModifier2RecordModifier, 238, **243**
RecordPattern, 207
RecordPattern2RecordPattern, 251, **252**
RecTrapStmt, 206
RecTrapStmt2RecTrapStmt, 246, **251**
ReturnStmt, 206
ReturnStmt2ReturnStmt, 246, **250**

Seq0Type, 197
Seq0Type2Seq0Type, 222, **223**
Seq1Type, 197
Seq1Type2Seq1Type, 222, **223**
SeqComprehension, 203
SeqComprehension2SeqComprehension, 238, **242**
SeqConcPattern, 207
SeqConcPattern2SeqConcPattern, 251, **252**
SeqEnumeration, 203
SeqEnumeration2SeqEnumeration, 238, **242**
SeqEnumPattern, 207
SeqEnumPattern2SeqEnumPattern, 251, **252**
SeqForLoop, 206
SeqForLoop2SeqForLoop, 246, **249**
SeqType, 197
SetBind, 207
SetComprehension, 203
SetComprehension2SetComprehension, 238, **242**
SetEnumeration, 202
SetEnumeration2SetEnumeration, 238, **242**
SetEnumPattern, 207
SetEnumPattern2SetEnumPattern, 251, **251**
SetForLoop, 206
SetForLoop2SetForLoop, 246, **249**
SetRange, 203
SetRange2SetRange, 238, **242**
SetType, 197
SetType2SetType, 222, **223**
SetUnionPattern, 207
SetUnionPattern2SetUnionPattern, 251, **251**
SplitOPatTypePairList, 229, **230**, 230, **230**, 231, 235
StateDef, 197
StateDef2StateDef, 220, **224**
StateDesignator, 204
StateDesignator2StateDesignator, 246, **246**, 246, 248, 250
Stmt, 204
Stmt2Stmt, 233, **246**, 247–249, 251
SubSequence, 203
SubSequence2SubSequence, 238, **242**

TaggedTypeDef, 196
TextLit, 209
TotalFnType, 197
Trap, 206
Traps, 206
TrapStmt, 206
TrapStmt2TrapStmt, 246, **250**
TupleConstructor, 203
TupleConstructor2TupleConstructor, 238, **243**
TuplePattern, 207
TuplePattern2TuplePattern, 236, 244, 251, **252**
Type, 196
Type2Type, 221–223, 225, 226, 228, 233, 244, 248, 252
TypeBind, 207
TypeBindList, 208
TypeDef, 196
TypeDefinition2TypeDefMap, 220, **220**, 220, 224
TypeDefinitions, 196
TypeDefinitions2TypeDefMap, 217, **220**
TypeName, 197
TypeName2TypeId, 222, **223**
TypeVar, 197
TypeVar2TypeVar, 222, **224**, 227
TypeVarId, 197

UnaryExpr, 201
UnaryOp, 201
UnionType, 197
UnionType2UnionType, 222, **222**, 222
UnTaggedTypeDef, 196

ValueDef, 198
ValueDef2ValDefMap, **225**, 225, **225**, 236
ValueDefinitions, 198
ValueDefinitions2ValDefMap, 219, **225**, 239, 247
ValueId, 208
VarInf, 199
VarInfs2ExtVarInfs, **233**, 233, **233**

WhileLoop, 206
WhileLoop2WhileLoop, 246, **249**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

D.5 Cross References for the Static Semantics

NOTE: Bold formula numbers denote defining occurrences of functions, plain formula numbers denote applied occurrences.

AssignDefSynComp, 408, **414**

BindSynComp, 408, **410**
BuildsOn, 92, **123**, 124

CallSynComp, 408, **415**
CanMatch, 190, **191**
CardofSet, **80**, 332
CasesStmt2CasesStmt, 299, **301**
CheckPairDisjoint, 68, **69**
CheckPairFunctionDisjoint, 69, **71**
CheckPairNonMatchDisjoint, 69, **70**
CheckProof, 54, **55**, 57–60
CheckProofBasic, 55, **56**
CheckProofDistUnion, 55, **61**
CheckProofInv, 55, **59**
CheckProofName, 55, **58**
CheckProofStructural, 55, **57**
CheckProofTypeVar, 55, **62**
CheckProofUnion, 55, **60**
Compatible2Maps, 243, **244**
CompatibleMapLetBindings, 265, **266**
CompatibleMapSet, 211, **212**
ComplCP, 171, **172**
ComplPCP, 319, **320**

DefNamesInAssignDef, 381, 398, **399**
DefNamesInAssignDefs, 380, **398**
DefNamesInBind, 371, 372, 380, 394, 395, **395**
DefNamesInDef, 106, 136, 357, 390, **391**
DefNamesInDefBind, 373, 396, **397**
DefNamesInDefBinds, 371, **396**
DefNamesInDefs, 136, 365, 371, 380, **390**
DefNamesInEqDef, 383, 400, **401**
DefNamesInEqDefs, 380, **400**
DefNamesInParTypes, 366, **402**
DefNamesInPat, 369, 380, 391, 392, **393**, 394, 395
DefNamesInPatBind, 380, **394**, 397, 401
DefNamesInPats, 366, 371, 380, **392**, 393, 395, 402
DistributeTypeR, 54, **95**
DoesNotBuildOn, 92, 121, 122, **124**

EmptyFnOverlap, 71, **72**
EmptyOpOverlap, 69, **74**
Exprs2Expr, 270, 307, **431**
ExtractFieldRs, 129, **131**
ExtractFnPostDefFromImplDef, 454, **457**
ExtractFnPreDefFromExplDef, 454, **455**
ExtractFnPreDefFromImplDef, 454, **456**
ExtractFnPrePostDefs, 98, **454**, 454
ExtractInitDef, 465, **466**
ExtractInitDefs, 98, **465**
ExtractInvDefs, 98, **462**, 462
ExtractInvFromStDef, 462, **464**
ExtractInvFromTpDef, 462, **463**, 464
ExtractOpPostDefFromImplDef, 458, **461**
ExtractOpPreDefFromExplDef, 458, **459**
ExtractOpPreDefFromImplDef, 458, **460**
ExtractOpPrePostDefs, 98, 458, **458**
ExtractTypeMap, 114, 116, **128**, 128, 366, 391, 467
ExtractTypeR, 119, 128, 129, **129**, 130, 131, 367, 419
ExtractTypeRs, 129, **130**

FindDepOrdering, 136, **153**
FreeNamesInAssignDef, 381, **382**
FreeNamesInAssignDefs, 380, 381, **381**
FreeNamesInBind, 371, 372, 375, **376**, 376, 380
FreeNamesInDef, 357, 365, **366**
FreeNamesInDefBind, 373, **374**
FreeNamesInDefBinds, 371, **373**, 373
FreeNamesInDefs, **365**, 371, 380
FreeNamesInEqDef, 383, **384**
FreeNamesInEqDefs, 380, **383**, 383
FreeNamesInExceptions, 366, **387**
FreeNamesInExpr, 366, 369–371, **371**, 372, 374, 376, 378, 380, 385–387, 423
FreeNamesInExprOrCall, 380, 382, 384, **386**
FreeNamesInExprs, **370**, 372, 386
FreeNamesInExternals, 366, **388**
FreeNamesInInvInitFn, 368, **369**
FreeNamesInParTypes, 366, **389**
FreeNamesInPat, 366, 369, 375–377, **378**, 380
FreeNamesInPatBind, 374, **375**, 380, 384
FreeNamesInPats, 366, 371, 376, **377**, 378, 380, 389
FreeNamesInSpecExpr, 371, **372**
FreeNamesInStateDesignator, 380, 385, **385**, 386
FreeNamesInStmt, 366, 379, **380**, 423
FreeNamesInStmts, **379**, 380
FreeNamesInType, **367**, 376, 382, 388, 389
FreeNamesInTypeR, 366, 367, **368**

GenExprSynComp, 408, **411**
GetOrExpr, 432, 433, **434**, 434
GetToken, 455–457, 459–461, 463, 466, **470**

HasMatchingSubtype, 163, **166**
HasNonTrivialPreCond, 111, 163, **167**

InducedPartitionOrder, 149, **151**
InScope, 143, 145, 146, **179**, 180, 185, 186, 188, 190, 253–256, 259, 262, 265, 276, 284, 289–291, 299, 302–304, 310, 312
InTotalFnScope, **180**, 276
InType, 143, 146, 168, **177**, 186, 189, 231, 253–256, 284, 290, 298, 305
Is, 37, 43, 69, 160, 275, 421, 423, 441, 442, 458, 462, **471**
Is-Expr, 386, **474**
Is-x-EmptyType, 55, 73, 87, **88**, 89
Is-x-OverlappingSubtype, 64, **65**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Is-x-Subtype, 52, 53, **54**, 65
IsApplicationType, 270, **271**
IsAssertedFromDef, 141, 143, 145, 146, 159, 162, **163**
IsAssertedVisEnv, **162**, 162
IsAsymmetric, 153, **154**
IsBindSetMerge, 345, 347, 348, **353**
IsBoundTo, **37**, 38, 41, 146, 273, 285
IsBoundToDomTypeR, **39**, 58, 70
IsBoundToValTypeR, **38**, 136, 279, 280
IsCurriedFn, 141, **421**, 421
IsDisjoint, 66, **67**, 145, 244, 276, 305
IsDisjointnessRel, 67, **68**
IsEmptyAux, 88, 89, **89**
IsEmptyType, 56, 63, **87**, 91, 92, 143, 275
IsEquivalent, **82**, 86, 91
IsEquivToUnionTypeR, 61, 63, **63**
IsEssentialSubtype, **53**, 83
IsEssEquivTypeR, **83**, 274
IsFiniteTypeR, 63, 92, **92**, 344
IsFlatTypeR, 121, **122**, 213, 259, 264, 265
IsFnRep, 141, **142**
IsFullSubtype, **52**, 82–85
IsFunctionDef, 110, **472**
IsFunctionTypeR, 71, **77**, 249, 250
IsGTE2, 250, **251**
IsIntersectionType, **85**, 197, 239, 351
IsLetExpandedParTypes, 143, 146, **422**
IsLetExpansion, 145, 274, 421, 422, **423**
IsMapTypeR, **76**, 209–212, 241, 243–250
IsNonEmptyEssEquivTypeR, **84**, 163, 295
IsNonEmptySet, **81**, 202, 342, **343**
IsOneValueType, 57, 86, **90**, 92, 212
IsOneValueTypeAux, 90, 91, **91**
IsOrderingOf, 136, **155**
IsOverlapping, 61, 65, **66**, 231
IsPartiallyOrdered, 149, 155, **156**
IsPattern, 375, 394, **473**
IsPermutationOf, 306, **364**
IsSafeAssertedFromDef, 158, **159**
IsSafeAssertedFromDefs, 157, **158**, 158
IsSafeAssertedVisEnv, 136, **157**
IsSeqTypeR, **78**, 206–208, 240, 263
IsSubEnvLoc, 179, 180, **181**
IsSubTVE, 320, **321**
IsSubtypeR, **64**, 84, 166, 191, 218, 220, 249, 260, 262, 268, 269, 295, 321, 327, 328, 339, 344
IsSubValEnv, 321, 327, **328**, 342, 344
IsTagOfType, 125, **126**
IsTransitivelyClosed, 361, **362**
IsTrueType, 186, **187**, 290
IsTveSeqUnion, 322, **323**
IsTveUnion, 322, 323, **324**
IsType, 419, **475**
IsTypeOfEmptyFn, 72, **73**, 75
IsTypeOfEmptyOp, 74, **75**
IsTypeRSubExpr, 126, **127**, 368
IsTypeRwithTagInEnv, 121, **125**, 268, 269, 277
IsUnionComp, 93, **93**, 94
IsUnionComponent, **94**, 95, 178, 181, 286, 291–295, 297, 309, 310, 312
IsVenvMerge, 138, **351**, 352
IsVenvSetMerge, 319, 352, **352**, 353
IsVenvUnion, 324, **350**
IsVisEnvMerge, 135, 137, **138**, 158, 162
IsVisibleTypeName, 39, **41**, 121
IsVisibleTypeVar, **40**, 121
IsZero, 250, **252**

LefthandSide, 106, **107**
LengthofSeq, **79**, 334

MakeFnEntry, 163, **164**
MakePartialFnTypeR, 160, **161**, 164
MapDomInInds, 241, **242**

NegWfClass, 38, **49**, 63, 66, 69, 71, 89, 91, 124, 143, 145, 146, 157, 160, 186, 250, 264, 275, 290, 342, 344
NoLooseInvariants, 98, **109**
NoLoosenessInDepValFnDefs, 109, **110**
NoLoosenessInExpr, 111, **112**
NoLoosenessInPat, 111, **113**
NoLoosenessInValFnDef, 110, **111**
NoNameClash, 98, **106**
NoUnsatisfiableTypes, 98, **108**

OldNewIds, 468, **469**, 469
ONInBindsSubst, 438, **450**
ONInBindSubst, 437, 438, 441, 451, **453**
ONInDefBindsSubst, 437, **440**
ONInDefsSubst, **439**
ONInExprOrCallSubst, 439, **442**
ONInExprsSubst, **436**, 438, 442
ONInExprSubst, 436, **437**, 437, 438, 440, 442, 443, 445, 449, 452, 453, 461
ONInLocalDefsSubst, 437, **444**
ONInLocalDefSubst, 444, **445**
ONInMultBindSubst, 450, **452**
ONInOExprSubst, **443**, 445
OnInParListSubst, **448**
ONInParTypesSubst, 445, **446**
ONInPatBindSubst, 439, 440, **441**
ONInPatsSubst, 437, 446, **447**, 448, 449, 452
ONInPatternSubst, 441, 445, 447, **449**, 449, 453
ONInSpecExprSubst, 437, **438**
ONInTypeBindsSubst, 438, **451**
OrderedDefGroups, 132, **149**
OType2Types, **430**, 459

ParameterTypes2Pats, 424, **425**, 456, 457, 460, 461
ParameterTypes2Pattern, 422, **424**
ParameterTypes2Type, 165, 422, **427**, 456
ParameterTypes2Types, 427, **428**, 457, 460, 461
Partition, 149, **152**
Pats2Pat, 145, 274, 421, 424, **426**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

PatSynComp, 408, **417**
PostOExceptions2Post, 146, **433**
PreAlwaysTrue, 143, 145, 146, **169**, 275, 276
PreNeverTrue, 143, 145, 146, **168**, 276
PreOExceptions2Pre, 146, **432**

ReflExtension, **363**, 406
RemoveCaseOther, 190, **192**
RemoveFromEnv, **44**, 136

SafeEnvEntry, 159, **160**, 160
SameValue, **86**, 244, 264, 331, 338, 351
SatisfiableStateInitialisation, 114, **117**
Shape, **96**, 202, 332, 334, 342
SimplifyCaseStmtAlternatives, 299, **300**, 300
SimultaneousDefRel, 149, **150**
SpecExprSynComp, 408, **412**
StateDesignatorSynComp, 408, **416**
StateName, 114, **115**, 458
StatePats, 458, **468**
StateTypeMap, 114, **116**, 467
StateVarNames, **35**, 146, 423
StateVisEnv, 35, **36**, 145, 146
StmtSynComp, 408, **413**
Subsume, 175, **176**, 287, 291–295, 297, 302–305, 307–310, 312–314
SubsumeExpr, **175**, 183
SubsumeTVE, 326, **327**
SynComp, 407, **408**
SynComps, 111, **405**, 467
SynSubComp, 406, 407, **407**

TransClosure, 153, 356, 358, **361**, 406
TransReflSynSubComp, 127, 405, **406**
TransTypeRDep, 123, **358**
TypeRDefBasis, 360, 420, **420**
TypeRDep, 358, **359**
TypeRSubExprDep, 359, **360**
TypeRSynComp, 408, **418**
TypeRTypeRel, 408, **419**
Types2Type, 274, 427, **429**, 457, 459–461
TypeSubst, 94, 273, 435, **435**, 435

UnionCloseExprCP, 171, **178**
UnionClosePCP, 319, 322, **322**
UniqueVal, 256, **257**
UpdateEnv, **43**, 135–137, 141, 143, 145, 146, 185, 186, 188, 190, 253–256, 259, 262, 265, 275, 284, 289–291, 293, 299, 302–304, 310, 312
UpdateVisEnv, **42**, 43, 137
UsedIds, 98, **467**

ValFnOpDefSynComp, 408, **409**
ValFnOpTpDefDep, 136, 356, **357**
ValFnOpTpTransDefDep, 108, 110, 149, **356**

wf-AllExpr, 182, **253**
wf-AlwaysStmt, 288, **309**
wf-AND, 214, **229**
wf-Apply, 182, **270**
wf-AssertedType, **120**, 163
wf-AssignDef, 293, **295**
wf-AssignStmt, 288, **297**
wf-BinaryExpr, 182, **214**
wf-Bind, 186, 255, 256, 290, **341**
wf-BindList, 253, 254, 259, 265, **345**
wf-BlockStmt, 288, **293**
wf-BracketedExpr, 182, **184**
wf-Call, 288, 296, **307**
wf-CallOrExpr, 292, 295, **296**, 297
wf-CasesExpr, 182, **190**
wf-CasesStmt, 288, **299**
wf-COMPOSE, 214, **249**
wf-CompPatn, **319**, 332–337
wf-DefBind, 139, **144**
wf-DefExpr, 182, **188**
wf-DefGroup, 135, **136**
wf-DefStmt, 288, **291**
wf-EQ, 214, **226**
wf-EqDef, 291, **292**
wf-EQUIV, 214, **232**
wf-ExistsExpr, 182, **254**
wf-ExistsUniqueExpr, 182, **255**
wf-ExitStmt, 288, **313**
wf-ExplFnDef, 139, **141**
wf-ExplOprtDef, 139, **145**
wf-Expr, 140, 143, 144, 146, 168, 171, **183**, 184–186, 188–191, 231, 253–256, 259, 262, 265, 276, 284, 287, 296, 298, 299, 302, 303, 305, 307, 308, 313, 331, 338, 342
wf-ExprBasic, **182**, 183
wf-FctTypeInst, 182, **273**
wf-FieldRef, 285, **286**
wf-FieldSelect, 182, **272**
wf-FLOOR, 193, **197**
wf-FnBody, 275, **276**
wf-IdentStmt, 288, **314**
wf-IfExpr, 182, **189**
wf-IfStmt, 288, **298**
wf-ImplFnDef, 139, **143**
wf-ImplOprtDef, 139, **146**
wf-IMPLY, 214, **230**
wf-IndexForLoop, 288, **304**
wf-INSET, 214, **233**
wf-INTDIV, 214, **219**
wf-IotaExpr, 182, **256**
wf-IsBasicTypeExpr, 182, **278**
wf-IsDefTypeExpr, 182, **277**
wf-ITERATE, 214, **250**
wf-Lambda, 182, **274**
wf-LambdaR, 141, 274, **275**
wf-LetBeSTExpr, 182, **186**
wf-LetBeSTStmt, 288, **290**
wf-LetExpr, 182, **185**
wf-LetStmt, 288, **289**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

wf-Literal, 182, **281**
wf-MapComprehension, 182, **265**
wf-MAPDISTRMERGE, 193, **211**
wf-MAPDOM, 193, **209**
wf-MAPDOMRESTRBY, 214, **246**
wf-MAPDOMRESTRTO, 214, **245**
wf-MapEnumeration, 182, **264**
wf-MapInverseExpr, 182, **213**
wf-MAPMERGE, 214, **243**
wf-MAPORSEQMOD, 214, **241**
wf-MapOrSeqRef, 285, **287**
wf-MAPRNG, 193, **210**
wf-MAPRNGRESTRBY, 214, **248**
wf-MAPRNGRESTRTO, 214, **247**
wf-MatchVal, 325, **331**
wf-MultBind, 345, **346**
wf-MultSetBind, 346, **347**
wf-MultTypeBind, 346, **348**
wf-Name, 182, **279**
wf-NE, 214, **227**
wf-NonDetStmt, 288, **306**
wf-NonStrictLOGOPRT, 228–230, **231**
wf-NonTpDef, 137, **139**
wf-NOT, 193, **198**
wf-NOTINSET, 214, **234**
wf-NUMABS, 193, **196**
wf-NUMBINMINUS, 214, **216**
wf-NUMBINPLUS, 214, **215**
wf-NUMDIV, 214, **218**
wf-NUMGE, 214, **225**
wf-NUMGT, 214, **224**
wf-NUMLE, 214, **223**
wf-NumLit, 281, **282**
wf-NUMLT, 214, **222**
wf-NUMMINUS, 193, **195**
wf-NUMMOD, 214, **221**
wf-NUMMULT, 214, **217**
wf-NUMPLUS, 193, **194**
wf-NUMREM, 214, **220**
wf-OExternals, 146, **147**
wf-OldName, 182, **280**
wf-OR, 214, **228**
wf-OrderedDefGroups, 132, 135, **135**
wf-OrderedDefs, 136, **137**, 137, 188
wf-ParameterTypes, 163, **165**
wf-Pattern, 166, 190, 191, 292, 299, 300, 303, 310, 319, **326**, 329, 338, 339, 342, 344
wf-PatternBasic, **325**, 326
wf-PatternBind, 140, 144, 302, **329**
wf-PatternId, 325, **330**
wf-PredInScope, 259, 262, 265, **284**
wf-PrefixExpr, 182, **193**
wf-PROPERSUBSET, 214, **236**
wf-RecordConstructor, 182, **268**
wf-RecordModifier, 182, **269**
wf-RecordPattern, 325, **337**
wf-RecTrapStmt, 288, **311**
wf-ReturnStmt, 288, **308**
wf-SCompExpr, **171**, 194–211, 213, 215–227, 232–241, 243, 245–250, 258, 260, 261, 263, 264, 267–270, 272, 277, 278, 304
wf-SeqComprehension, 182, **262**
wf-SEQCONC, 214, **240**
wf-SeqConcPattern, 325, **335**
wf-SEQDISTRCONC, 193, **208**
wf-SEQELEMS, 193, **206**
wf-SeqEnumeration, 182, **261**
wf-SeqEnumPattern, 325, **334**
wf-SeqForLoop, 288, **302**
wf-SEQHEAD, 193, **203**
wf-SEQINDICES, 193, **207**
wf-SEQLEN, 193, **205**
wf-SEQTAIL, 193, **204**
wf-Sequence, 293, 294, **294**, 306
wf-SetBind, 262, 341, **342**, 347
wf-SETCARD, 193, **199**
wf-SetComprehension, 182, **259**
wf-SetConstrPattern, 329, **338**
wf-SETDIFFERENCE, 214, **238**
wf-SETDISTRINTERSECT, 193, **202**
wf-SETDISTRUNION, 193, **201**
wf-SetEnumeration, 182, **258**
wf-SetEnumPattern, 325, **332**
wf-SetForLoop, 288, **303**
wf-SETINTERSECT, 214, **239**
wf-SETPOWER, 193, **200**
wf-SetRange, 182, **260**
wf-SETUNION, 214, **237**
wf-SetUnionPattern, 325, **333**
wf-Spec, **98**
wf-StateDesignator, **285**, 286, 287, 297, 307
wf-Stmt, 145, **288**, 289–291, 293, 294, 298, 299, 302–304, 309–312
wf-SubSequence, 182, **263**
wf-SUBSET, 214, **235**
wf-TextLit, 281, **283**
wf-Traps, 311, **312**, 312
wf-TrapStmt, 288, **310**
wf-TupleConstructor, 182, **267**
wf-TuplePattern, 325, **336**
wf-Type, **119**, 120, 163, 165, 273, 274, 295, 339, 344
wf-TypeBind, 341, **344**, 348
wf-TypeConstrPattern, 329, **339**
wf-TypeMap, 114, **118**
wf-TypeR, 118, 119, 121, **121**
wf-TypeStateDefsInInitialEnv, 98, **114**
wf-ValFnOpDefs, 98, **132**, 185, 289
wf-ValueDef, 139, **140**
wf-VarInf, 147, **148**
wf-WhileLoop, 288, **305**

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex E
(informative)
Bibliography

[ACJ72] C. D. Allen, D. N. Chapman, and C. B. Jones. A formal definition of ALGOL 60. Technical Report 12.105, IBM Laboratory Hursley, August 1972.

[AeWO] D. Andrews, R. Henry (ed), D. Ward, et al. ISO/IEC 10514 : 1996 Information technology — Programming languages — Modula-2.

[AH82] Derck Andrews and Wolfgang Henhapl. 7: Pascal. In C. B. Jones D. Bjørner, editor, *Formal Specification & Software Development*, pages 175–251. phi, 1982.

[AI91] Derek Andrews and Darell Ince. *Practical Formal Methods with VDM.* McGraw Hill, 1991.

[ANS76] ANSI/ECMA Programming Language Standard: PL/I, 1976.

[BBH+74] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory Vienna, December 1974.

[BD92] D. Bolignano and M. Debabi. Higher order communicating processes with value-passing, assignment and return of results. In *Proceedings of the ISAAC'92 Conference*, 1992.

[Be78] D. Bjørner and C. B. Jones (eds.). *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science.* Springer-Verlag, 1978.

[Be82] D. Bjøner and C. B. Jones (eds.). *Formal Specification and Software Development.* Prentice-Hall International, 1982.

[Bek84] H. Bekič, editor. *Programming Languages and Their Definition: Selected Papers of H. Bekič*, volume 177 of *Lecture Notes in Computer Science.* Springer-Verlag, 1984.

[BG80] R. M. Burstall and J. A. Goguen. *The Semantics of Clear, a Specification Language*, pages 292–332. Springer-Verlag, 1980. LNCS 86.

[BJ82] D. Bjørner and C. B. Jones. *Formal Specification & Software Development.* Prentice Hall International, 1982.

[BO80] D. Bjørner and O. N. Oest, editors. *Towards a Formal Description of Ada*, volume 98 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[Daw91] John Dawes. *The VDM-SL Reference Guide.* Pitman, 1991.

[dBS69] J. W. de Bakker and D. Scott. A theory of programs. Manuscript notes for IBM Seminar, Vienna, August 1969.

[Den92] B. T. Denvir, editor. *Formal Aspects of Computing Journal, Volume 4, Number 1, Special Issue on Structuring Specifications.* Springer Verlag, 1992.

[End72] H. B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, New York, 1972.

[Fei89] L. M. G. Feijs. The calculus $\lambda\pi$. In M. Wirsing and J. A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, volume 394 of *Lecture Notes in Computer Science*, pages 307–328. Springer Verlag, 1989.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

[Fei90] L. M. G. Feijs. Norman's Database Modularised in COLD-K. In J. A. Bergstra and L. M. G. Feijs, editors, *Algebraic Methods II: Theory, Tools and Applications*, volume 490 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[Fit91a] J. S. Fitzgerald. *Modularity in Model-Oriented Formal Specifications and its Interaction with Formal Reasoning*. PhD thesis, Dept. of Computer Science, University of Manchester, UK, 1991. Available as Technical Report UMCS 91-11-2 from Dept. of Computer Science, University of Manchester, UK.

[Fit91b] J. S. Fitzgerald. Reasoning about a modular model-oriented formal specification. In *Proc. Intl. Workshop on Specifications of Database Systems, University of Glasgow 1991*, Workshops in Computer Science. Springer Verlag, 1991.

[FJ90] J. S. Fitzgerald and C. B. Jones. Modularizing the Formal Description of a Database System. In D. Bjørner, C.A.R. Hoare, and H. Langmaack, editors, *VDM '90: VDM and Z – Formal Methods in Software Development*, volume 428 of *Lecture Notes in Computer Science*. Springer Verlag, 1990.

[GH86] John V. Guttag and James J. Horning. Report on the Larch shared language. *Science of Computer Programming*, 6:103–134, 1986.

[Gro92] Raise Language Group, editor. *The RAISE Specification Language*. Prentice Hall, 1992.

[HJ70] W. Henhapl and C. B. Jones. On the interpretation of GOTO statements in the ULD. Technical Report LN 25.3.065, IBM Laboratory, Vienna, March 1970.

[JJLM91] C. B. Jones, K. D. Jones, P. A. Lindsay, and R. Moore. *mural: A Formal Development Support System*. Springer-Verlag, 1991.

[JL71] C. B. Jones and P. Lucas. Proving correctness of implementation techniques. In E. Engeler, editor, *A Symposium on Algorithmic Languages*, volume 188 of *Lecture Notes in Mathematics*, pages 178–211. Springer-Verlag, 1971.

[Jon70] C. B. Jones. A technique for showing that two functions preserve a relation between their domains. Technical Report LR 25.3.067, IBM Laboratory, Vienna, April 1970.

[Jon72] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. In *SIGPLAN Notices, Volume 7 Number 1*, pages 150–169. ACM, January 1972.

[Jon80] C. B. Jones. *Software Development - A Rigorous Approach*. Prentice Hall International, 1980.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.

[Jon90] C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.

[JS90] C. B. Jones and R. Shaw. *Case Studies in VDM*. Prentice Hall International, 1990.

[Jon89] H. B. M. Jonkers. Description algebra. In M. Wirshing and J. A. Bergstra, editors, *Algebraic Methods: Theory, Tools and Applications*, pages 283–305. Springer-Verlag, 1989. LNCS 394.

[Kle52] S. C. Kleene. *Introduction to Mathematics*. North Holland, 1952. Republished in 1957, 59, 62, 64, 71.

[Luc68] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory Vienna, June 1968.

[McC61] J. McCarthy. A basis for a mathematical theory of computation. In *Western Joint Computer Conference*, 1961. Then published in: Computer Programming and Formal Systems (P. Braffort, D. Hirstberg eds.) North Holland 1967, 33–70.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

[Mid92] C. A. Middelburg. Modular structuring of VDM specifications in VVSL. *Formal Aspects of Computing*, 4(1):13–47, 1992.

[Mid93] C. A. Middelburg. *Logic and Specification – Extending VDM-SL for Advanced Formal Specification.* Computer Science Series: Research and Practice 1. Chapman & Hall, 1993.

[Mil90] R. Milne. The semantic foundations of the raise specification language. Technical report, STC Technology Ltd, 1990.

[Mon86] Brian Q. Monahan. Formal definition of the translation from the abstract syntax into its semantical counterpart. Doc. no. 12, March 1986.

[Mon87] Brian Q. Monahan. A type model for vdm. In Airchinnigh Bjø rner, Jones and Neuhold, editors, *VDM '87 VDM – A Formal Method at Work*, pages 210–236. VDM-Europe, Springer-Verlag LNCS 252, 1987.

[TW90] Andrzej Tarlecki and Morten Wieth. A Naive Domain Universe for VDM. In Dines Bjørner, C. A. R. Hoare, and Hans Langmaack, editors, *VDM '90 VDM and Z – Formal Methods in Software Development*, pages 552–579. VDM Europe, Springer-Verlag, April 1990.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

© ISO/IEC

ICS 35.060

Descriptors: data processing, computer sofware, artificial languages, programming languages, specification languages.

Price based on 399 pages

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale