

Mathematics & Computer Science

A Formal Definition of VDM-SL

IFAD

Technical University of Delft
Technical University of Denmark
The University of Leicester

D Andrews (Editor)

Technical Report No. 1998/9

Department of Mathematics
& Computer Science
University of Leicester
University Road
Leicester LE1 7RH
England



Leicester
University

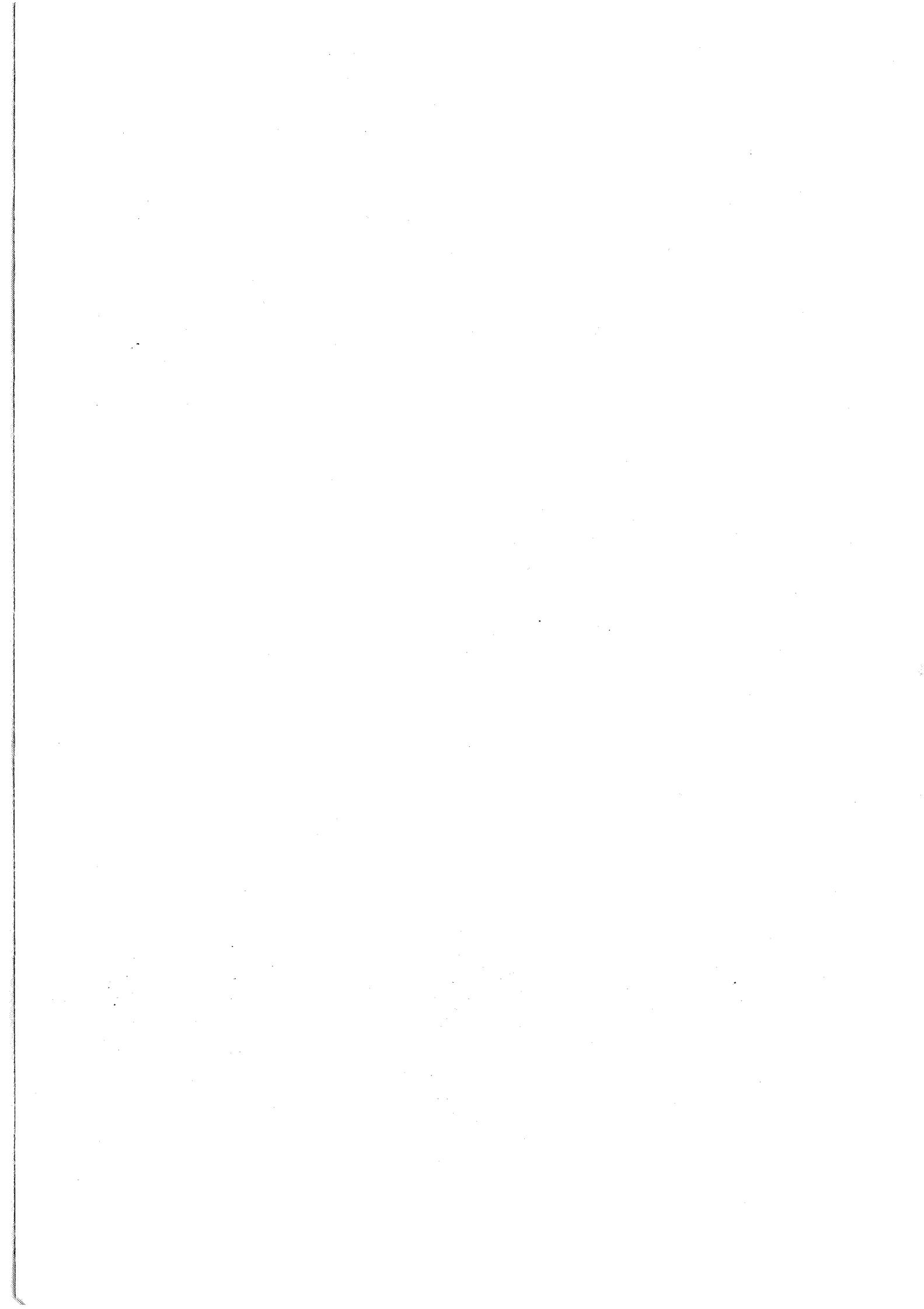
A Formal Definition of VDM-SL

June 1998

A Joint Technical Report from:
IFAD
Technical University of Delft
Technical University of Denmark
The University of Leicester

D. J. Andrews Leicester University (editor)
H. Bruun Technical University of Denmark
F. Damm Technical University of Denmark
J. Dawes ICL
B. S. Hansen Technical University of Denmark
P. G. Larsen IFAD
G. Parkin NPL
N. Plat Technical University of Delft
H. Toetenel Technical University of Delft

©1996



Foreword

This joint report from the Danish Institute for Applied Computer Science (IFAD), the Technical Universities of Delft and Denmark and the University of Leicester contains the background and technical material used in the production of the ISO Standard [LHP⁺96] that defines the specification language part of the Vienna Development Method; this language is called VDM-SL. The joint report is an amalgamation of published reports produced during the period of the project. The material of the original reports has been updated and incorporates various suggestions and corrections that were made by the participants and reviewers of the project — these changes have improved the style and technical correctness of the formal definitions used to define VDM-SL.

The Vienna Development Method

The *Vienna Development Method* (VDM) is a formal method for the description and development of software systems. VDM consists of two components:

1. A mathematically based *specification language* in which specifications can be written in a very abstract and formal form.
2. A *method* describing a process for the development of an implementation of a software system using an arbitrary programming language from a formal specification. The method consists of a series of development steps by which a VDM specification is transformed into an implementation (an executable program) and of a number of proof obligations that must be discharged to justify each development step. Each step generates a new specification which is less abstract (more concrete) and closer to the implementation than the previous one [Din89]. However, no guidelines as to how the development steps should be selected are present in the VDM method. It is only suggested that the level of abstraction should be reduced. The process of moving between the development steps is known as refinement.

The original specification language, Meta-IV, evolved into several different variants. These variants reflect two different styles in the use of VDM: the Danish school and the English school¹. The difference between these schools is primarily caused by different application areas: respectively, 'systems software' and 'algorithm and data refinement'. The main difference between the two schools is that the Danish school focused on modelling large software systems using an explicit style, and thus used the metalanguage as an abstraction mechanism to obtain a better understanding of the complexity of the systems, whereas the English school has focused more on the correctness aspects of software systems. The English school used a more implicit style (using pre- and post-conditions) in order to ease the verification of the systems. However, the different variants also reflect the idea that VDM has a pragmatic approach to its use; the meta-language has been extended with properties that were needed for specific applications.

A Brief History of the Project

The early version of the VDM specification language (META-IV) was developed at the IBM Laboratory in Vienna where it was used to define the programming language PL/I. Work was also done to develop

¹An Irish school of VDM, based on the Danish approach, has also developed [Air87a, Air87b]; this school uses a purely functional notation together with a comprehensive theory of data types.

a compiler from the definition. Other programming languages were also formally defined using this early version of the specification language:

- definitions of minimal BASIC and parts of FORTRAN and APL were undertaken at IBM Hursley in the '70s;
- a formal definition of ALGOL 60 was written by Wolfgang and Cliff Jones at the Vienna Laboratory [HJ78]; and
- Ada and Pascal were specified in the early '80s [BO80, AH82].

The first description of this version of the Vienna Development Method (VDM) and its specification language META-IV can be found in [BJ78].

During the late seventies and early eighties, Cliff Jones developed those aspects of VDM not specifically related to language specification and compiler development, and the first book on what is now generally thought of as the English school of VDM is [Jon80]. This book and the original work that described the specification language [BJ78] have now been superseded: the language description work is best accessed in [Be82] and the non-language work is best seen in [JS90] and in [AI91].

The specification aspects of VDM were taken up in the eighties by the STL laboratory in Harlow and, partly because of their industrial push, the BSI (British Standards Institution) was persuaded to establish a working group whose task was to produce a BSI Standard for the specification language part of the Vienna Development Method — the specification language VDM-SL. The working group started work to harmonize the different variants to produce a Standard for Meta-IV in the mid-eighties [Sen87, And88, PL92, Par94]. In the early nineties the standardization work was taken up by the International Organization for Standardization (ISO) to define an ISO Standard [LHP⁺93, LHP⁺95, LHP⁺96] for the specification language.

The standardization activity has not been easy because of the different requirements of those interested in using the specification language of VDM to define programming languages and those who were more interested in using the specification language in software development. Language definition used the specification language to provide formal semantics of the programming language; software development involved specifying, developing and writing software using implicit specification, data reification and operation decomposition. It is to the credit of the Standards Committee that they have managed to combine the requirements of both types of user and come up with a Standard that embraces a wide scope of technical ideas.

STL was responsible for funding the first formal semantics of VDM-SL and this work was carried out at Manchester University by Brian Monahan [Mon87]. These semantics were used as the starting point of the formal description of the specification language produced as part of the standardization effort. Starting with the STL work, a new formal description of VDM-SL was written because of the need to merge the two aspects of the specification language. The result is the formal description of the dynamic semantics of VDM-SL that forms the first part of this report.

The success of the work to merge the two styles of specification language can be seen from the results: though the Standard Language was based on the STL work on defining a language for general software development, VDM-SL has been used in the ISO Standard for the programming language Modula-2 [AHW⁺96].

The Dynamic Semantics

The first version of the dynamic semantics was written by Michael Meincke Arentoft and Peter Gorm Larsen as a masters thesis in the early part of 1988. The approach was heavily based on [Mon85] and [Bea88]. A full model resulting from this work was circulated towards the end of 1988, and a presentation given at an open BSI/VDM meeting at the VDM '88 symposium by Peter Gorm Larsen. The thesis did not give the complete semantics for VDM-SL; the missing parts of the definition were completed during the latter part of that year. This part of the project was funded by the Department of Computer Science at the Technical University of Denmark (TUD) and by a grant from the private Danish Trane Foundation. During this initial period, contact with the BSI/VDM standardization panel was kept by

Morten Wieth whose participation at the panel meetings was financed by the Danish Technical Research Foundation (STVF).

In 1989 work at Leicester University and the Technical University of Denmark together with John Dawes of ICL produced a full abstract and concrete syntax for VDM-SL, however this work changed the original abstract syntax used for the dynamic semantics and meant that a revision of the semantics was required—this was completed in early 1989.

In January 1989 at a VDM-Europe meeting it was decided that a review of the embryo formal definition of VDM-SL would be required to improve its correctness. With this aim in mind a Review Board² was formed to check both the mathematical foundations and the specification itself. Meetings of the review board occurred in March, August and December of that year. The results of these reviews led to a new version of the dynamic semantics. At the IFIP world congress in 1989 a presentation was given by Peter Gorm Larsen on providing the formal semantics for the VDM specification language [LAMB89]. There was, however, a technical problem with the domain part of the early versions of the dynamic semantics. Wiesław Pawłowski and Peter Gorm Larsen worked together for three months to resolve these difficulties and adopted a domain universe that was based on work proposed in [TW90]. Andrzej Borzyszkowski also joined this team for a month to concentrate on the semantics of the non-deterministic statements of the specification language.

The Static Semantics

In 1991 work was started on the static semantics for VDM-SL. This work was undertaken by Hans Bruun, Flemming M. Damm and Bo Stig Hansen working at the Technical University of Denmark (TUD). The foundations for this work were presented in two papers at VDM'91 [DBH91, BHD91]. A second review board³ was convened to review both the dynamic and static semantics. This board had its first meeting at TUD in September 1991 and its second meeting in The Hague in May 1992. In 1992 the first version of the static semantics for the full VDM-SL language was produced and checked by members of the review board plus Andrzej M. Borzyszkowski and Wiesław Pawłowski. In 1993 minor revisions were made to the static semantics and included in the ISO Standard Committee Draft which was submitted to ISO in December 1993 [LHP⁺93].

During 1994 intensive work on proving that the static semantics were correct with respect to the dynamic semantics resulted in the discovery of several errors in both the static semantics and also in the dynamic semantics. The correctness checking of the static semantics was completed in 1995 and the final version of the static semantics became part of the ISO Draft International Standard published in September of 1995 [LHP⁺95].

The Syntax Mapping

In 1991 work started at Delft University of Technology to define a formal transformation as a mapping from the VDM-SL Outer Abstract Syntax (used for the static semantics) and the Core Abstract Syntax (used for the dynamic semantics). This mapping provides an essential link between the VDM-SL static and dynamic semantics. This ‘syntax mapping’ was written by Nico Plat and Hans Toetenel. The work was reviewed by Andy Lovering which led to many significant improvements. The bulk of the work was finished in 1992. In 1993 the syntax mapping was also mechanically checked which revealed a few more minor errors.

The Structure of the Report

This report is a collection of the final versions of the various documents that provide a formal definition of VDM-SL. It has incorporated the input from the two EC review boards and the work of various reviewers.

²The members of the first review board were Derek Andrews, Cristoph Blaue, Andrzej Blikle (Chairman), Tim Denvir, Clive Jervis, Cliff Jones, Hans Langmaack, Brian Monahan, Nick North, and Jimmi Pettersson.

³The members of the second review board were Derek Andrews, Cristoph Blaue, Tim Denvir, Jean Goubault, Cliff Jones, Kees Middelburg (Chairman), Søren Prehn, and Hans Toetenel.

The work to produce both this report and the ISO Standard was coordinated and managed at Leicester University. The sources of the material used to construct this report are given below:

- Chapters 1–4 are from [Lar92]
- Chapter 5 is from [PT92]
- Chapters 6–7 are from [ABH⁺92]
- Chapters 8–9 are from [BHD⁺95]

Chapter 1 covers the basic mathematical notation (used as the meta-language in the definition of the dynamic semantics).

Chapter 2 defines the core abstract syntax which the semantic definitions are based on.

Chapter 3 constructs the domain universe and presents the semantic meta-domains that are build from it.

Chapter 4 contains the definition of the dynamic semantics of VDM-SL.

Chapter 5 gives the syntax mapping between the outer abstract syntax and the core abstract syntax used in the definition of the dynamic semantics.

Chapter 6 gives the mathematical concrete syntax and outer abstract syntax.

Chapter 7 describes the interchange concrete syntax that uses an ‘ASCII’ character set.

Chapter 8 defines the domains used by the static semantics.

Chapter 9 contains the definition of the static semantics.

VDM resources

A wide range of publications about VDM and VDM-SL and software tools that support both the method and the specification language are available. A VDM bibliography is available that contains more than 600 references [Lar94]. This bibliography is available at <ftp://ftp.ifad.dk/pub/vdm/vdm.bib.gz>. Several tutorial text books exist [JS90, AI91, LBC90]; a reference manual of VDM-SL [Daw91] and a book about proof in VDM [BFPLR94]. Another source of information is the VDM Forum [Fit94] which is an e-mail based newsgroup concerning VDM. More information about VDM can be accessed on the World Wide Web on the Internet via the URL link <http://www.ifad.dk/vdm/vdm.html>. A number of tools which support VDM-SL have also been developed [BFM89, JSS91, ELL94].

Acknowledgements

The work to produce this report and the ISO International Standard was supported by the following institutions and organizations:

The Institute of Applied Computer Science (IFAD)
The Technical University of Denmark
The Technical University of Delft

The University of Leicester
The University of Manchester
The British Standards Institute (BSI)
The RAISE project from the CEC ESPRIT-II program
CEC Directorate-General 11
The Danish Technical Research Foundation (STVF)
Standard Telecommunication Laboratories (STL)

The following people also provided input of one form or another to the dynamic semantics report and the ISO International Standard: Poul Bøgh Lassen, René Elmstrøm and Michael Andersen at IFAD.

Proof reading of earlier versions of the reports that make up this document has been done by many people, but the authors would especially like to thank Anne-Katrine Lundebye at the International Science Park Odense (ISPO) and Keith Hooper of Waikato University in Hamilton, New Zealand.

Contents

1	The Dynamic Semantics of VDM-SL	1
1.1	Introduction	1
1.2	Basic Mathematical Notation	1
1.3	The Core Abstract Syntax	1
1.4	The Domain Universe	2
1.5	Loose Specification	3
1.5.1	Looseness in Functions	3
1.5.2	Looseness in Operations	3
1.6	The Semantic Definition Style	4
1.7	Semantic Properties for VDM-SL	5
1.7.1	The logic at the VDM-SL level	5
1.7.2	Recursive Definitions	5
1.7.3	Polymorphism	6
1.7.4	Partial Functions	6
2	Basic Mathematical Notation	8
2.1	Logic Notation	8
2.2	Basic Set Theory	8
2.3	Cartesian Products	9
2.4	Binary Relations and Functions	10
2.5	Finite Sequences	11
2.6	Finite Mappings	12
2.7	Ordinal Numbers	13
2.8	Definition by Transfinite Induction	14
2.9	Cardinality and Cardinal Numbers	14
2.10	Structured Expressions	14
2.11	Semantic Function and Predicate Definitions	16
2.12	Use of Recursion	17
3	Core Abstract Syntax	18
3.1	Document	18
3.2	Definitions	18
3.2.1	Type Definitions	18
3.2.2	State Definition	20
3.2.3	Value Definitions	20
3.2.4	Function Definitions	21
3.2.5	Operation Definitions	21
3.3	Expressions	22
3.3.1	Local Binding Expressions	23
3.3.2	Conditional Expressions	23
3.3.3	Unary Expressions	23
3.3.4	Binary Expressions	24
3.3.5	Quantified Expressions	24

3.3.6	Iota Expression	24
3.3.7	Set Expressions	25
3.3.8	Sequence Expressions	25
3.3.9	Map Expressions	25
3.3.10	Tuple Constructor	26
3.3.11	Record Expressions	26
3.3.12	Apply Expressions	26
3.3.13	Lambda Expression	27
3.3.14	Is Expression	27
3.3.15	Literals	27
3.3.16	Identifiers	27
3.4	State Designators	28
3.5	Statements	29
3.5.1	Local Binding Statements	29
3.5.2	Block and Assignment Statements	29
3.5.3	Conditional Statements	30
3.5.4	Loop Statements	30
3.5.5	Non-Deterministic Sequences	31
3.5.6	Call and Return Statements	31
3.5.7	Exception Handling Statements	31
3.6	Patterns and Bindings	32
4	Dynamic Semantic Domains	34
4.1	The Domain Universe	34
4.1.1	Basic Definitions	34
4.1.2	A Universe of Complete Partial Orders	37
4.1.3	A Universe of Domains	38
4.2	The Semantic Domains	40
4.2.1	Basic Semantic Domains	40
4.2.2	Extended Semantic Domains	42
4.2.3	Semantic Domains for Evaluation Functions	43
4.2.4	Auxiliary Semantic Domains	45
5	The Dynamic Semantics	46
5.1	Document	47
5.2	Definitions	48
5.2.1	Type Definitions	49
5.2.2	State Definition	57
5.2.3	Value Definitions	57
5.2.4	Function Definitions	63
5.2.5	Operation Definitions	69
5.3	Expressions	73
5.3.1	Local Binding Expressions	75
5.3.2	Conditional Expressions	77
5.3.3	Unary Expressions	79
5.3.4	Binary Expressions	83
5.3.5	Quantified Expressions	94
5.3.6	Iota Expression	99
5.3.7	Set Expressions	99
5.3.8	Sequence Expressions	104
5.3.9	Map Expressions	106
5.3.10	Tuple Constructor	108
5.3.11	Record Expressions	109
5.3.12	Apply Expressions	110
5.3.13	Lambda Expression	112

5.3.14	Is Expression	113
5.4	State Designators	114
5.5	Statements	118
5.5.1	Underlying Theory For Statements	118
5.5.2	The Statement Evaluation Functions	120
5.5.3	Local Binding Statements	120
5.5.4	Block and Assignment Statements	123
5.5.5	Conditional Statements	126
5.5.6	Loop Statements	127
5.5.7	Non-Deterministic Sequences	131
5.5.8	Call and Return Statements	131
5.5.9	Exception Handling Statements	132
5.6	Patterns and Bindings	136
5.6.1	Patterns	137
5.6.2	Bindings	145
5.7	Auxiliary Functions and Predicates	145
5.7.1	Expansion Functions	145
5.7.2	Functions for Extending the Environment	148
5.7.3	Functions and Predicates to deal with the State	152
5.7.4	Functions and Predicates to deal with Curried Functions	154
5.7.5	Compute Functions	155
5.7.6	Generate Functions	157
5.7.7	Make Functions	158
5.7.8	Get Functions	160
5.7.9	Collector Functions	164
5.7.10	Selector Functions	167
5.7.11	Tag-processing Functions	169
5.7.12	General Functions and Predicates	171
6	The Mathematical Concrete Syntax and Outer Abstract Syntax	178
6.1	Document	178
6.2	Definitions	178
6.2.1	Type Definitions	178
6.2.2	State Definition	182
6.2.3	Value Definitions	183
6.2.4	Function Definitions	183
6.2.5	Operation Definitions	184
6.3	Expressions	186
6.3.1	Bracketed Expressions	187
6.3.2	Local Binding Expressions	187
6.3.3	Conditional Expressions	188
6.3.4	Unary Expressions	189
6.3.5	Binary Expressions	190
6.3.6	Quantified Expressions	193
6.3.7	Iota Expression	194
6.3.8	Set Expressions	194
6.3.9	Sequence Expressions	195
6.3.10	Map Expressions	195
6.3.11	Tuple Constructor Expression	195
6.3.12	Record Expressions	196
6.3.13	Apply Expressions	196
6.3.14	Lambda Expression	196
6.3.15	Is Expressions	197
6.3.16	Names	197

6.4	State Designators	198
6.5	Statements	198
6.5.1	Local Binding Statements	199
6.5.2	Block and Assignment Statements	200
6.5.3	Conditional Statements	200
6.5.4	Loop Statements	201
6.5.5	Nondeterministic Statement	202
6.5.6	Call and Return Statements	202
6.5.7	Exception Handling Statements	202
6.5.8	Identity Statement	203
6.6	Patterns and Bindings	203
6.6.1	Patterns	203
6.6.2	Bindings	204
6.7	Lexical Specification	205
6.7.1	General	205
6.7.2	Characters	206
6.7.3	Symbols	208
6.8	Operator Precedence	210
6.8.1	The Family of Combinators	211
6.8.2	The Family of Applicators	211
6.8.3	The Family of Evaluators	211
6.8.4	The Family of Relations	212
6.8.5	The Family of Connectives	212
6.8.6	The Family of Constructors	213
6.8.7	Grouping	213
6.8.8	The Type Operators	213
7	The Interchange Concrete Syntax	214
7.1	Introduction	214
7.2	Lexis	214
7.3	Symbols	214
8	The Syntax Mapping	218
8.1	Structure and Style of the Definition	218
8.1.1	Division into Modules	218
8.1.2	Pre-conditions in the VDM-SL Definition of the Syntax Mapping	219
8.1.3	Transformation of a Document to CAS.Definitions	219
8.1.4	Notational Conventions	223
8.2	Syntaxes and Auxiliary Functions	223
8.2.1	Module “OAS”	223
8.2.2	Module “CAS”	223
8.2.3	Module “GetUnusedId”	224
8.3	The Syntax Mapping Functions	224
8.3.1	Document	225
8.3.2	Definitions	228
8.3.3	Expressions	246
8.3.4	State Designators	255
8.3.5	Statements	256
8.3.6	Patterns and Bindings	261
8.3.7	Lexical Specification	263

9 The Background to the Static Semantics	265
9.1 Abstract Syntaxes, Dynamic Semantics and Consistency	265
9.2 Rôle of the Static Semantics	265
9.3 Overview of the Definition	268
9.3.1 Notation and Style	268
9.3.2 Type Representations	268
9.3.3 Well-formedness Classifications	268
9.3.4 Type Relations	269
9.3.5 Characteristic Predicates	269
9.3.6 Relaxation of Characteristic Predicates	270
9.3.7 Well-formedness Predicates	270
10 The Static Semantic Domains	272
10.1 Type Representations	272
10.1.1 Special Subclasses of Type Representations	274
10.2 Environments	274
10.2.1 Accessing Environments	275
10.2.2 Updating Environments	275
10.3 Well-formedness Classifications	276
10.4 Type Relations	276
10.4.1 Subtypes	276
10.4.2 Overlapping Subtypes, Disjoint Types and Overlapping Types	282
10.4.3 Auxiliary Type Relations and Functions	285
10.5 Extended Abstract Syntax	290
11 The Static Semantics	291
11.1 Documents	291
11.1.1 Auxiliary Well-formedness Requirements	293
11.2 Definitions	294
11.2.1 Type and State Definitions	294
11.2.2 Value, Function, and Operation Definitions	298
11.3 Expressions	308
11.3.1 Expression Characteristic Predicates	308
11.3.2 Relaxations and Restriction of Predicates	309
11.3.3 Well-Formedness of Expressions	313
11.3.4 Bracketed Expression	314
11.3.5 Local Binding Expressions	314
11.3.6 Conditional Expressions	315
11.3.7 Unary Expressions	317
11.3.8 Binary Expressions	323
11.3.9 Quantified Expressions	337
11.3.10 Iota Expression	337
11.3.11 Set Expressions	338
11.3.12 Sequence Expressions	339
11.3.13 MapExpressions	340
11.3.14 Tuple Constructor	341
11.3.15 Record Expressions	342
11.3.16 Apply Expressions	343
11.3.17 Lambda Expressions	344
11.3.18 Is Expressions	346
11.3.19 Names	346
11.3.20 Literals	346
11.3.21 Auxiliary Well-formedness Predicates	347
11.4 State Designators	347
11.5 Statements	348

11.5.1 Local Binding Statements	349
11.5.2 Block and Assignment Statements	350
11.5.3 Conditional Statements	352
11.5.4 Loop Statements	354
11.5.5 Non-Deterministic Statement	355
11.5.6 Call and Return Statements	356
11.5.7 Exception Handling Statements	356
11.5.8 Identity Statements	358
11.6 Patterns and Bindings	359
11.6.1 Patterns	359
11.6.2 Bindings	364
11.6.3 Value Environment	367
11.7 Auxiliary Functions	368
11.7.1 Dependency Relations	368
11.7.2 Syntax Transformations	383
11.7.3 Substitutions	385
11.7.4 Indirectly Defined Functions	390
11.7.5 Classification Functions	393
A Cross References for the Dynamic Semantics	399
A.1 Naming and Typesetting Conventions Used	399
A.1.1 Listing of Functions/Predicates: Alphabetic (uses)	400
B The Concrete and Abstract Syntax	406
C The Syntax Mapping	411
D The Static Semantics	413

List of Figures

8.1 Structure of the syntax mapping	218
9.1 Rejected and accepted subsets of specifications.	267

List of Tables

6.1 Character set	207
7.1 Interchange syntax: representation of symbols	215

Chapter 1

The Dynamic Semantics of VDM-SL

1.1 Introduction

This preamble gives a short overview of the definition of the dynamic semantics for VDM-SL, and the foundations on which it is based. The overview is mainly structured in the same way as the rest of this report. First the basis is dealt with, and then an overview of the semantic properties of the language is given.

The main purpose of this preamble is to give the reader an understanding of the reasons behind the choices which have been made in the definition of the dynamic semantics for VDM-SL. Therefore, it will also be mentioned whether other alternatives are technically feasible¹.

1.2 Basic Mathematical Notation

The ‘meta-language’ used for the definition of the dynamic semantics is based on ZF set theory (see e.g. [End77]). In, for example [Spi88], the formal semantics of the Z notation is given using a subset of Z as a meta-language. This meta-circularity is well-known from programming languages, beginning with a LISP interpreter written in LISP (see [M⁺62]). This is possible for programming languages because a compiler only has to describe compilation (and not logical equality). In the case of a specification language like VDM-SL, which must be able to deal with non-termination (bottom) as a normal value, one needs to use an expressive logical language (such as ZF set theory). Since VDM-SL is almost as expressive as ZF set theory, one could consider using a subset of VDM-SL to describe its semantics. However, if one wants to be fully formal there are subtle problems which make meta-circular definitions problematic. Therefore we have chosen to use mathematics where we have fixed our notation in chapter 2.

1.3 The Core Abstract Syntax

The core abstract syntax (CAS) has been introduced in order to simplify the definition of the dynamic semantics, and in order to use a denotational approach. This means that there is a standard abstract syntax (OAS) from which the CAS will automatically be derived. These two abstract syntaxes are related by a formal transformation that is presented in [PT92].

The main advantage of the CAS for the dynamic semantics is that all definitions are grouped together according to their kind. In addition, in all places where the order in the concrete syntax is without importance the ordering has been abstracted away (e.g. sequences to sets).

¹It may require a certain insight into the problems in order to fully understand the various alternatives on the basis of the short descriptions given here (a more thorough introduction to VDM-SL can be found in [Daw91]). However, references to the literature which enable the reader to get a deeper understanding of the problems will be provided in these cases.

1.4 The Domain Universe

Based on the ‘set-level’ basic mathematical notation a number of cpo² operators have been defined. These are used to construct a domain universe which provides denotations of all values expressible in VDM-SL. The construction of the domain universe for VDM-SL is “naive” in the sense that, although it is based on complete partial orders, it does not involve a sophisticated theory with full reflexive domains like the domain theory in the style of Scott (see e.g. [Sco76]). Thus, in order to achieve a consistent domain universe which enables a least fixed point semantics of VDM-SL definitions, we need to restrict the values which can be used in VDM-SL.

The Domain Universe for VDM-SL is strongly inspired by [TW90], which again is related to [BT83]. Except for notational differences the changes can be listed as:

- Generalized Cartesian product instead of binary Cartesian product has been used.
- Only the smashed³ versions of product, record, and mapping spaces have been used.
- Lists have been added (they are essentially the same as the tuples produced by the generalized Cartesian product, but they have been given different tags).
- The TOKEN type has been added because it was omitted in [TW90].
- The continuous function space has been replaced by the full function space. In this way the domain universe additionally provides non-continuous functions.

The kind of domains which can be used in VDM-SL are:

- Basic domains such as Booleans, and some numeric values, characters, and tokens are also available.
- Finite sets where the elements must be ‘flat’. The reason for only taking flat domains into consideration is that for non-flat domains there is no natural way of constructing a cpo of finite subsets. To understand the kind of problems that arise when one wants to ‘extend’ a non-trivial ordering on elements to an ordering on sets of elements, one can consult [Plo76].
- Smashed tuples. We could also have the full Cartesian products. However, this would require having an extra cross product operator at the concrete syntax level, which can be used in recursive type definitions in a ‘non-essential’ way (cf. [TW90]).
- Finite (smashed) lists. Because of the domain universe structure we do not have domains of finite and infinite lists such as presented in e.g. [Sch86].
- Smashed records. As for tuples, we could introduce an extra record operator into the concrete syntax and prohibit using it in an essential recursive way when writing type definitions (cf. [TW90]).
- Finite (smashed) maps where the elements in the domain must be flat. There is no technical problem in allowing a non-smashed map operator, which is already present in [TW90]. However, if the domain is non-flat, the domain universe construction does not work.
- Function space, except that recursion can only be defined “over” but not “through” it⁴.

Finally, it should be noted that the domain universe construction contains a notion of union-compatible cpo’s. It is strongly believed that because of explicit tagging within the construction of the domain universe the user cannot define types which are not union-compatible. However, this has not been formally proven. A number of other things have been proven concerning the domain universe in [TW90], but we have chosen not to include any of these proofs in this report.

²cpo means “Complete Partial Order”.

³By ‘smashed’ we mean that all partially undefined values (compound values including bottom) are considered to be identical to bottom.

⁴This means that it is possible to make domain equations like $D = D \times D \mid \mathbb{N} \rightarrow \mathbb{N}$ but not equations like $D = D \rightarrow D \mid \mathbb{N}$ which requires Scott domain theory with full reflexive domains.

1.5 Loose Specification

Formal specification languages have properties not usually found in programming languages. One of these is the possibility of specifying constructs that denote a choice. Such ‘loose specification’ arises because a specification generally needs to be stated at a higher level of abstraction than that of the final implementation. When loose specification is used, the question of how to interpret this looseness is often ignored. However, this interpretation is important, especially if a specification is to be proven to implement another specification. The implementation relation relies on the interpretation of looseness. Thus, this interpretation is especially interesting in connection with the proof rules for the specification language. However, the implementation relation and the proof rules have not yet been formulated for the entire VDM-SL language which is described here.

There are (at least) two different ways of interpreting loose specification. We have termed these: ‘underdeterminedness’⁵ (allowing several different implementations) and ‘nondeterminism’ (allowing nondeterministic implementations). The difference between the two lies in the time at which the looseness is resolved. If a loosely specified construct is interpreted as underdetermined, it is decided by the implementer at implementation time which functionality to use. Since this choice is static the final implementation becomes deterministic. However, if a loosely specified construct is interpreted as nondeterministic, it is possible to delay this choice until execution time. In this way the actual functionality is chosen dynamically at run-time. This means that the final implementation may be nondeterministic. Ideally a specifier would like to be able to choose freely between these different interpretations for all functions and operations.

In VDM-SL we have chosen to interpret function definitions as underdetermined, while operation definitions are interpreted as nondeterministic⁶. The difference between functions and operations is that operations may operate on a (specified) state space. Thus, functions are applicative while operations may be imperative.

It is natural that implicit function and operation definitions can contain looseness. However, in VDM-SL it is also possible to let explicit function and operation definitions be loosely specified. Expressions and pattern matchings can be loosely specified in this language. Thus, when there is looseness in an expression or a pattern its interpretation depends upon whether it is used inside a function or an operation.

1.5.1 Looseness in Functions

In a number of cases it is convenient to leave as much freedom to the implementor of a specification as possible, and still ensure ‘referential transparency’⁷. In VDM-SL we have chosen to enable a user to do this by means of functions. A syntactic function definition which contains looseness is interpreted as being underdetermined (i.e. it will denote the set of mathematical functions which satisfy the specification). Each of these mathematical functions can be used as implementation of the function specification). This means that the relation stating that a program is an implementation of a specification — the *implementation relation* — is simply the set membership relation on the respective denotations. This model for underdeterminedness is in some cases (where the data the function is operating upon is refined) too restrictive (see [HJ89]). Keeping the notion of referential transparency the only way of avoiding this restriction is by using ‘multi-functions’ (returning sets of possible results). The major problem with using this approach in VDM-SL is that the user of VDM-SL would have to operate directly with the sets (even though many of them would be singleton sets corresponding to deterministic functions). Thus, we have chosen to live with the restriction we have imposed here, because the operation part enables the user to deal with this restriction (with the loss of the referential transparency though).

1.5.2 Looseness in Operations

While the concept of underdeterminedness is strongly connected to that of specification, the concept of nondeterminism is applicable to specifications as well as programs. Nondeterminism in programs has mostly been discussed in connection with concurrency. However, it is often also desirable to abstract

⁵In the literature ‘underdeterminedness’ has also been called ‘under-specification’.

⁶The complexity of the semantics with an arbitrary combination of loose specification is given in [Wie89].

⁷This means that a function will always return the same result for the same argument.

the description of references that the program makes to input values not controlled by the user (e.g. the time of the day, or the currently available memory size of the computer). This means, that in terms of the input values chosen to be considered, the program can behave in a nondeterministic fashion (even in cases without concurrency). In VDM-SL we have chosen to enable the user to describe this by means of operations. Operations are modelled as relations between input value, input state, output state and output value. In addition, each element in such a relation contains information about which mode the operation is returned in. This is caused by the operational flavor of operations which can report error situations by an exit mode, or indicate whether a normal value is returned.

However, it is not sufficient to say that looseness in operations is interpreted as nondeterminism. A number of additional choices must also be made. Most of these choices relate to the implementation relation of operations, and even though the main purpose of this report is simply to provide a semantics which can find the set of models which a given syntactic specification denotes, we will also briefly discuss the different additional choices which must be made in connection with nondeterminism. The chosen kinds are indicated by using an *italics* font.

The first choice one must make is between tight and loose nondeterminism (see [Par79]). Tight nondeterminism prescribes that a valid implementation must be able to produce all of the results that are possible according to the specification; the implementation must exhibit the same degree of nondeterminism as the specification. Loose nondeterminism allows the implementation to be ‘less nondeterministic’ than the specification, by only demanding that all values that can be produced by the implementation must be allowed by the specification. In VDM-SL *loose nondeterminism* is the most appropriate choice.

When a programming language is given semantics it is a natural assumption that all choices have a finite number of possibilities (this is also called bounded nondeterminism). However, for a specification language there will in general be an infinite number of possibilities, which means that *unbounded nondeterminism* is a natural choice.

Looking at a certain observation level one can talk about weak nondeterminism (also called internal nondeterminism) or strong nondeterminism. Weak nondeterminism means that at a certain level of abstraction the system is essentially deterministic, even though components at a lower level are nondeterministic. *Strong nondeterminism* enables nondeterminism at any level (this is especially important at the outermost level), and this is used for VDM-SL.

The question of non-termination shows new aspects when nondeterminism is introduced. The nondeterminism can either be ‘erratic’, ‘angelic’, or ‘demonic’ depending upon how non-termination is handled. If it is erratic, all possible values (both defined and undefined) are taken into account. If it is angelic, the undefinedness will be avoided if possible, while undefinedness always will be chosen (if possible) if it is demonic. Since we want to be able to specify all possible behaviours we choose the *erratic* kind of semantics of nondeterminism.

Finally, it is necessary to choose between a singular or plural binding of variables. This choice can be seen as a question of whether or not identifiers can be bound to a kind of ‘choice value’ for which the ‘real’ value is chosen later on⁸. If singular binding is chosen two occurrences of an identifier will also denote the same value in a given scope, whereas plural bindings would enable the two occurrences to denote different values. We have chosen the *singular binding* because we believe that it corresponds better to our intuition about the meaning of such expressions.

Having made these choices we have fully described which kind of nondeterminism we are dealing with.

1.6 The Semantic Definition Style

When giving semantics to programming languages it has been customary to specify an elaboration function for each of the syntactic domains. These elaboration functions yield the denotation for the corresponding domains. This means that the denotation of the syntactic domain is *explicitly* constructed.

Another approach places the emphasis on the *relation* between the specification and its denotation. A number of semantic domains are built on top of the domain universe, and the top-level semantic domain, ENV_{PURE} , corresponds to all possible models for a VDM-SL specification. In this implicit style

⁸This strategy is similar to a ‘call by name’ approach where the actual identifiers are not ‘evaluated’ before they are needed.

the syntactic constructs are related to an already given model, such that every possible model (from ENV_{PURE}) can be tested against the specification.

We have chosen to use the implicit style of definition as done in [Hoa85] and [Mon85]. We use an *IsAModelOf*-predicate (in [Hoa85] it is called *sat*) for testing whether a given model is logically associated with a syntactic specification.

The implicit style has been used mainly because the explicit style requires an order in which the definitions must appear. VDM-SL does not have this property and it would require many changes to the language if this were to be done. It would also mean that all kinds of mutual recursion would have to be grouped together, for example, implicitly and explicitly defined functions. Another reason for choosing the implicit style was that it seemed to be easier to deal with when loose specification is present.

1.7 Semantic Properties for VDM-SL

Having sketched the foundations on which the definition of the dynamic semantics is based we will in this section give an overview of important semantic properties of VDM-SL.

1.7.1 The logic at the VDM-SL level

In the classical (two valued) logic every hypothesis is either true or false. Nevertheless, it is possible to write meaningless expressions in VDM-SL which are neither true nor false. Therefore we have to use a ‘three-valued’ logic to be able to describe the semantics of all expressions. There are several ways of extending the traditional two-valued logic to the three valued case (see [BKB88] or [Che86] for an overview). Without going into the different alternatives here, note that we have simply used a candidate originally published in [Kle38] and [Kle52]. In [Jon90] and [CJ90] it is called LPF (Logic for Partial Functions).

It can be useful to consider an operational view of this logic. Imagine that all operands are evaluated in parallel. As soon as a result is available for an operand, it is considered; if the single result uniquely determines the overall result (e.g. if one operand is false for a conjunction), evaluation of all other operands can be stopped and the (determined) result is returned. This is an operational explanation of the logic which illustrates that the logic needs unbounded parallelism to be ‘executed’.

1.7.2 Recursive Definitions

When one is dealing with recursive definitions they will generally not have a unique solution. This is also true if we do not deal with looseness (see [Sto77] or [Sch86] for an introduction). It is possible to find the ‘least’ solution if a recursive definition satisfies certain continuity criteria, and a constructive functional exists for the definition which makes it possible to ‘calculate’ the object to be defined. This theory is known as least fixed point semantics for recursive definitions. These principles are used for most of the explicit definitions in this report, while no solution to give the implicit definitions such a semantics has been found. The strategies can be summarised as:

Mutually recursive type definitions have been given a least fixed point semantics, and the invariants of the definitions are included in the fixed point iteration. No looseness is allowed here so this is reasonable standard.

Mutually recursive explicit function definitions (both monomorphic and polymorphic) have been given a least fixed point semantics. However, since functions can be loosely specified (interpreted as underdeterminedness) the standard way cannot be used directly. In order to get an idea about how this is dealt with, it is necessary to explain a little about how the semantics of loose expressions is structured.

The semantics of an expression which forms the body of an explicit function is a set of expression evaluators. Each of these evaluators perform a choice for all loose parts of the body expression. Thus, each of these evaluators is deterministic and the standard way of least fixed point semantics can be followed. The mutually recursive explicit function definitions are treated in this way, and will therefore have a set of least fixed points corresponding to different choices of underdeterminedness.

Mutually recursive value definitions have been given a least fixed point semantics. However, since expressions in value definitions can be loosely specified the same strategy as above is used.

Mutually recursive implicit function definitions have been given a kind of all fixed point semantics (meaning that all solutions fulfilling the equations are used as denotations). This has been done because there generally does not exist a constructive way of producing the least fixed point.

Mutually recursive implicit operation definitions have also been given a kind of all fixed point semantics (see above).

Mutually recursive explicit operation definitions are given what can informally be called a ‘reasonable’ fixed point semantics. It is in fact quite close in spirit to the least fixed point semantics. However, since no appropriate ordering has been found between operation denotations, this is not, technically, a genuine least fixed point semantics.

In general, a VDMthis report will consist of a collection of different kinds of definitions (value definitions, function definitions, etc.). A denotation of such a collection is such that, when we restrict our view to some kind of definitions, the strategy corresponding to this kind (see the description above) is used in the context of the denotations of the other definitions. For example, if:

values ... $f(x)$...

is a value definition, and:

functions

$f(a : A) b : B$
pre $P(a, f, x)$
post $Q(a, b, f, x)$

is an implicit function definition, where the two definitions are mutually recursive, then the correct denotations are those for x and f , such that:

- Given x , f is any fixed point of its defining equation as an implicit function (all fixed point semantics).
- Given f , x is the least fixed point of its defining equation as a value (least fixed point semantics).

The difficulty here is that the denotations have not been constructed, but defined implicitly instead.

1.7.3 Polymorphism

In VDM-SL it is possible to define polymorphic functions. The kind of polymorphism used can be described as ‘first order explicit parametric polymorphism’ (see [CW85]). Thus polymorphic functions cannot take other polymorphic functions as arguments. Thereby problems concerned with staying within the set-theoretic framework are avoided (see [Rey84]). Applications of user-defined polymorphic functions must be provided with explicit instantiation of type variables.

Furthermore, it can be noted that polymorphic “types” are not types themselves, in the sense that one cannot introduce a polymorphic type by means of a type definition. The problem with allowing this, is how to cope with recursive type definition of polymorphic type variables. The domain universe can possibly be extended such that this could be permitted.

1.7.4 Partial Functions

Most functions which are defined in VDM-SL are total (with respect to the arguments which satisfy their pre-condition). For such function definitions it holds that if the pre-condition is fulfilled, the mathematical functions which the definition denotes should behave according to the body of the definition, but the result must not be undefinedness (bottom). If the body of the definition does yield bottom, the function definition is inconsistent, i.e. it has no models.

Since it is not always possible to statically write down a pre-condition which ensures that the body of the function is defined, there is a need to allow partial functions. Such partial functions are incorporated in VDM-SL. For partial function definitions it holds that if the pre-condition is fulfilled, the mathematical functions which the definition denotes should behave according to the body of the definition, even if the definition says that the result is undefinedness (bottom). Semantically these partial functions have been modelled as total functions, where bottom in the codomain indicates partiality. Thus, we have chosen not to include two different type constructors which can be used in the type definitions, but to use the partiality only at the level of functions instead.

Chapter 2

Basic Mathematical Notation

This chapter 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 chapter the basic mathematical notation used for defining this function is introduced. It will be basic in the sense that the operators defined in this chapter 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 chapter is to fix the notation at the set-theoretic level, which is going to be used in the following chapters. 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 chapter) 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.

2.1 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

2.2 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 section 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 intersection* 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 , $\text{IP}(A)$ denotes the powerset of A , i.e. the set of all subsets of A . Similarly, $\text{IF}(A)$ 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 $\{a_1, a_2, \dots, a_n\}$. 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 $\varphi : I \rightarrow A$ such that $\varphi(i) = a_i$ for all $i \in I$. The set of all such families will be denoted by $I \rightarrow A$. 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, 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 section about cardinal numbers (see 2.9).

Finally, some basic domains (such as Booleans (**IB**), the natural numbers (**IN**), integers (**Z**), the rational numbers (**Q**), reals (**IR**)) are used and appropriate operators on them (such as plus (+), minus (-), multiplication, (\times), 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. abs, the absolute value).

2.3 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, \dots, A_n is defined by:

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

Here the *tupling* operator $(-, \dots, -)$ has the following property:

$$\forall a_1, a_1' \in A_1, \dots, \forall a_n, a_n' \in A_n \cdot$$

$$(a_1, \dots, a_n) = (a_1', \dots, a_n') \Leftrightarrow (a_1 = a_1') \wedge \dots \wedge (a_n = a_n')$$

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

For an n -ary Cartesian product $\bigtimes_{i=1}^n A_i$ there is a family of n *projections* $\{\pi_1, \dots, \pi_n\}$ such that for all $(a_1, \dots, a_n) \in \bigtimes_{i=1}^n A_i$ and for all $k = 1, \dots, n$:

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

The tupling operator $(-, \dots, -)$ and the projection functions have the following property:

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

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.

2.4 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 , $A \simeq B$ denotes the set of all *partial functions* from A to B . This set is defined by:

$$A \simeq B = \{f \mid f \in \text{IP}(A \times B) \cdot \forall (a, b_1), (a, b_2) \in f \cdot b_1 = b_2\}.$$

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

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

A function $f: A \simeq B$ 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 $A \rightarrow B \subseteq A \simeq B$ defined by:

$$A \rightarrow B = \{f \mid f : A \simeq B \cdot \delta_0(f) = A\}.$$

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

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

Functions are defined in typed λ -notation¹. 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 $\lambda x \in A. e \in A \rightarrow B$ and denotes the function:

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

2.5 Finite Sequences

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

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

where ω is the first infinite ordinal, i.e. the set of all natural numbers (see also section 2.7). Let $[]$ denote the empty sequence.

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

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

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

$$[a_1, \dots, a_n]$$

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

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

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

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 $\{1, \dots, \underline{\text{len}}(s)\}$. Thus there is the (partial) *projection* function

$$-(_) : \mathbb{L}(A) \times \omega \rightsquigarrow A$$

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 $f : \omega \rightsquigarrow A$, $D \in \mathbb{F}(\omega)$ such that $D \subseteq \delta_0(f)$, and P is a unary predicate on ω , then:

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

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 $I = \{1, \dots, \underline{\text{card}}(D')\}$, $D' = \{i \in D \mid P(i)\}$, 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:

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

$\underline{\text{hd}} : \mathbb{L}_1(A) \rightarrow A$
 $\underline{\text{hd}}(s) = s(1)$
 $\underline{\text{tl}} : \mathbb{L}_1(A) \rightarrow \mathbb{L}(A)$
 $\underline{\text{tl}}(s) = [s(i) \mid i \in \{2, \dots, \underline{\text{len}}(s)\}]$
 $\underline{\text{inds}} : \mathbb{L}(A) \rightarrow \text{IF}(\omega)$
 $\underline{\text{inds}}(s) = \{1, \dots, \underline{\text{len}}(s)\}$
 $\underline{\text{elems}} : \mathbb{L}(A) \rightarrow \text{IF}(A)$
 $\underline{\text{elems}}(s) = \{s(i) \mid i \in \underline{\text{inds}}(s)\}$
 $\underline{\text{join}} : \mathbb{L}(A) \times \mathbb{L}(A) \rightarrow \mathbb{L}(A)$
 $\underline{\text{join}}([a_1, \dots, a_n], [b_1, \dots, b_m]) = [a_1, \dots, a_n, b_1, \dots, b_m]$
 $\underline{\text{Join}} : \mathbb{L}(\mathbb{L}(A)) \rightarrow \mathbb{L}(A)$
 $\underline{\text{Join}}(l_1, \dots, l_n) = \underline{\text{join}}(l_1, \underline{\text{join}}(l_2, \dots \underline{\text{join}}(l_{n-1}, l_n) \dots)).$

2.6 Finite Mappings

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

$$\text{IM}(A, B) = \{m : A \simeq B \mid \delta_0(m) \text{ is finite}\}.$$

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

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

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

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

For $f : A \simeq B$, $A' \in \text{IF}(A)$, 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 $\text{IM}(A, B)$ 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:

dom : $\text{IM}(A, B) \rightarrow \text{IF}(A)$

$$\underline{\text{dom}}(m) = \delta_0(m)$$

rng : $\text{IM}(A, B) \rightarrow \text{IF}(B)$

$$\underline{\text{rng}}(m) = \delta_1(m)$$

restrict : $\text{IM}(A, B) \times \text{IP}(A) \rightarrow \text{IM}(A, B)$

$$\underline{\text{restrict}}(m, s) = \{a \mapsto m(a) \mid a \in (\underline{\text{dom}}(m) \cap s)\}$$

subtract : $\text{IM}(A, B) \times \text{IP}(A) \rightarrow \text{IM}(A, B)$

$$\underline{\text{subtract}}(m, s) = \{a \mapsto m(a) \mid a \in (\underline{\text{dom}}(m) \setminus s)\}$$

merge : $\text{IM}(A, B) \times \text{IM}(A, B) \simeq \text{IM}(A, B)$

$$\delta_0(\underline{\text{merge}}) = \{(m_1, m_2) \mid m_1, m_2 \in \text{IM}(A, B) \cdot \forall a \in (\underline{\text{dom}}(m_1) \cap \underline{\text{dom}}(m_2)) \cdot m_1(a) = m_2(a)\}$$

$$\underline{\text{merge}}(m_1, m_2) = m_1 \cup m_2$$

Merge : $\text{IF}(\text{IM}(A, B)) \simeq \text{IM}(A, B)$

$$\delta_0(\underline{\text{Merge}}) = \{ms \mid ms \in \text{IF}(\text{IM}(A, B))\}$$

$$\quad \forall m_1, m_2 \in ms \cdot \forall a \in \underline{\text{dom}}(m_1) \cap \underline{\text{dom}}(m_2) \cdot m_1(a) = m_2(a)\}$$

$$\underline{\text{Merge}}(ms) = \bigcup ms$$

overwrite : $\text{IM}(A, B) \times \text{IM}(A, B) \rightarrow \text{IM}(A, B)$

$$\underline{\text{overwrite}}(m_1, m_2) = \underline{\text{merge}}(\underline{\text{subtract}}(m_1, \underline{\text{dom}}(m_2)), m_2).$$

A map abbreviation

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

2.7 Ordinal Numbers

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

$$0 \leq 1 \leq 2 \leq \dots$$

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

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

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

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

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

In set theory ([?]) 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:

$$n \leq m \text{ iff } n \in m \text{ or } n = m$$

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 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.

2.8 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 $!: \omega \rightarrow \omega$ 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 , $F : A \rightarrow A$ is a function and $G : B \rightarrow A$ where $B = \{f \mid f : \beta \rightarrow A \text{ for some } \beta < \alpha\}$. Then there exists a unique function $H : \alpha \rightarrow A$ such that:

1. $H(0) = a$.
2. $H(S(\beta)) = F(H(\beta))$ for every β such that $S(\beta) < \alpha$.
3. $H(\beta) = G(H[\beta])$ for every limit ordinal $\beta < \alpha$.

2.9 Cardinality and Cardinal Numbers

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

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 $\omega < \underline{\text{card}}(S)$.

2.10 Structured Expressions

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

The if expression has the usual meaning:

```
if T then e1 else e2 = e1
if F then e1 else e2 = e2.
```

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

```

if P ∧ 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 ∨ Q
then cons
else altn

```

it will correspond to:

```

if ¬P
then if ¬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'
```

can be read as a shorthand for:

```
(λx : X . e')(e).
```

The cases construct:

```

cases e :
  c11, ..., c1n → e1
  ...
  ci1, ..., cil → ei
  ...
  A_SET → ej
  ...
  cm1, ..., cmk → em

```

where the c 's are constant values, has the value e_i iff $e \in \{c_{i1}, \dots, c_{il}\}$. 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: $A_SET = \{c_1, c_2, \dots, c_n\}$.

The let and cases constructs are also used in forms that facilitate the decomposition of tagged values (chapter 5 contains the formal definitions for *MkTag*, *StripTag*, and *ShowTag* if they are required for the understanding of the following). The expression:

let $MkTag(t, x) = e$ **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 $ShowTag(e)$:

...
 $t \rightarrow \text{let } x = StripTag(e) \text{ in } e'$
...

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:

$t :: \text{some_selector} : x$

2.11 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 $Function\text{-definition}$

Annotations to $FunctionName$:

5. Function-annotations
End of annotations

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

5. $FunctionName : Type$

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 section 2.1.

Predicates in this report are not semantically considered as truth-valued functions. However (for example) a predicate *P* will be used as:

```
if P(...)  
then ...  
else ...
```

instead of

```
if P(...) = True()  
then ...  
else ...
```

2.12 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.

Chapter 3

Core Abstract Syntax

The definition of dynamic semantics as specified in chapter 5 is based on the core abstract syntax presented in this chapter.

In this chapter 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.

3.1 Document

Document = Definitions

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

3.2 Definitions

```
Definitions ::= typem      : ValueId  $\xrightarrow{m}$  TypeDef
              expofnm   : ValueId | PreId | PostId  $\xrightarrow{m}$  ExplPolyFnDef
              exmofnm   : ValueId | PreId | PostId | InitId | 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]
```

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.

3.2.1 Type Definitions

```
TypeDef ::= shape : Type
          inv   : InvId  $\times$  Lambda
```

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. $\lambda a : A \cdot \text{true}$). 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.

$$\text{Type} = \text{BasicType} \mid \text{CompositeType} \mid \text{ProductType} \mid \text{UnionType} \mid \text{OptionalType} \mid \text{SetType} \mid \text{SeqType} \mid \text{MapType} \mid \text{FnType} \mid \text{TypeVar} \mid \text{QuoteType}$$

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.

$$\text{BasicType} :: bt : \text{NATONE} \mid \text{NAT} \mid \text{INTEGER} \mid \text{RAT} \mid \text{REAL} \mid \text{BOOLEAN} \mid \text{CHAR} \mid \text{TOKEN} \mid \text{UNIT}$$

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 section 4.2.1).

$$\text{CompositeType} :: id : \text{ValueId} \\ \text{fields} : \text{Field}^*$$

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.

$$\text{Field} :: sel : [\text{ValueId}] \\ \text{type} : \text{Type}$$

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

$$\text{ProductType} :: \text{fields} : \text{Type}^*$$

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.

$$\text{UnionType} :: tps : \text{Type-set}$$

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

$$\text{OptionalType} :: tp : \text{Type}$$

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

$$\text{SetType} :: \text{elemtp} : \text{Type}$$

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*.

$$\text{SeqType} = \text{Seq0Type} \mid \text{Seq1Type}$$

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

$$\text{Seq0Type} :: \text{elemtp} : \text{Type}$$

Seq0Type 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*.

$$\text{Seq1Type} :: \text{elemtp} : \text{Type}$$

Sq1 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 type¹ or an injective map type.

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

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).

3.2.2 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*.

3.2.3 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.

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

3.2.4 Function Definitions

```
ExplPolyFnDef :: tpparms : TypeVar+
    polytp : Type
    pre : Expr
    body : Lambda
    kind : TOTAL | PARTIAL
```

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.

3.2.5 Operation Definitions

Observe that:

1. Operations cannot be Curried but they can take functions as arguments and return functions as result.
2. Operations cannot be polymorphic.

```
ExplOpDef ::= heading : OpHeading
                  body   : Stmt
```

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*.

3.3 Expressions

```
Expr = LetExpr | LetBeSEExpr | 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.

3.3.1 Local Binding Expressions

```

LetExpr::vals : Pattern  $\xrightarrow{m}$  ValDef
explfns : ValueId  $\xrightarrow{m}$  ExplFnDef
implfns : ValueId  $\xrightarrow{m}$  ImplFnDef
in : Expr

```

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.

```

LetBeSTExpr::bind : Bind
st : Expr
in : Expr

```

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*.

```

DefExpr::lhs : Pattern
rhs : Expr
in : Expr

```

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*.

3.3.2 Conditional Expressions

```

IfExpr::test : Expr
cons : Expr
altn : Expr

```

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

```

CasesExpr::sel : Expr
altns : CaseAltn+

```

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

```

CaseAltn::match : Pattern
body : Expr

```

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.

3.3.3 Unary Expressions

```

UnaryExpr::opr : UnaryOp
arg : Expr

```

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

$$\begin{aligned} UnaryOp = & \text{NUMPLUS} \mid \text{NUMMINUS} \mid \text{NUMABS} \mid \text{FLOOR} \mid \\ & \text{NOT} \mid \\ & \text{SETCARD} \mid \text{SETPOWER} \mid \text{SETDISTRUNION} \mid \text{SETDISTRINTERSECT} \mid \\ & \text{SEQHEAD} \mid \text{SEQTAIL} \mid \text{SEQLEN} \mid \text{SEQELEMS} \mid \text{SEQINDICES} \mid \text{SEQDISTRCONC} \mid \\ & \text{MAPDOM} \mid \text{MAPRNG} \mid \text{MAPINVERSE} \mid \text{MAPDISTRMERGE} \end{aligned}$$

UnaryOp is an ‘unary operator’.

3.3.4 Binary Expressions

$$\begin{aligned} BinaryExpr ::= & left : Expr \\ & opr : BinaryOp \\ & right : Expr \end{aligned}$$

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*.

$$\begin{aligned} BinaryOp = & \text{NUMPLUS} \mid \text{NUMMINUS} \mid \text{NUMMULT} \mid \text{NUMDIV} \mid \text{INTDIV} \mid \text{NUMREM} \mid \text{NUMMOD} \mid \\ & \text{NUMLT} \mid \text{NUMLE} \mid \text{NUMGT} \mid \text{NUMGE} \mid \text{EQ} \mid \text{NE} \mid \\ & \text{OR} \mid \text{AND} \mid \text{IMPLY} \mid \text{EQUIV} \mid \\ & \text{INSET} \mid \text{NOTINSET} \mid \text{SUBSET} \mid \text{PROPERSUBSET} \mid \\ & \text{SETUNION} \mid \text{SETINTERSECT} \mid \text{SETDIFFERENCE} \mid \\ & \text{SEQCONC} \mid \text{MAPORSEQMOD} \mid \text{MAPMERGE} \mid \\ & \text{MAPDOMRESTRTO} \mid \text{MAPDOMRESTRBY} \mid \text{MAPRNGRESTRTO} \mid \text{MAPRNGRESTRBY} \mid \\ & \text{COMPOSE} \mid \text{ITERATE} \end{aligned}$$

BinaryOp is a ‘binary operator’.

3.3.5 Quantified Expressions

$$QuantExpr = AllOrExistsExpr \mid ExistsUniqueExpr$$

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

$$\begin{aligned} AllOrExistsExpr ::= & quant : Quantifier \\ & bind : Bind-set \\ & pred : Expr \end{aligned}$$

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 quantification covers, *pred*.

$$Quantifier = \text{ALL} \mid \text{EXISTS}$$

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

$$\begin{aligned} ExistsUniqueExpr ::= & bind : Bind \\ & pred : Expr \end{aligned}$$

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

3.3.6 Iota Expression

$$\begin{aligned} IotaExpr ::= & bind : Bind \\ & pred : Expr \end{aligned}$$

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 $\exists !\text{bind} \cdot \text{pred}$.

3.3.7 Set Expressions

SetEnumeration :: *els* : *Expr**

SetEnumeration is a set enumeration expression which simply consists of a sequence² 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.

3.3.8 Sequence Expressions

SqEnumeration :: *els* : *Expr**

SqEnumeration is a ‘sequence enumeration expression’. It is simply a sequence of element expressions, *els*.

```
SqComprehension ::= elem : Expr
                   bind : Bind
                   pred : Expr
```

SqComprehension 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.

3.3.9 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*.

²The 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).

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.

3.3.10 Tuple Constructor

TupleConstructor::*fields* : *Expr*⁺

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

3.3.11 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*³. The expression takes a record, *rec*, and modifies those fields that are present in the domain of *modifiers* with the corresponding element expressions.

3.3.12 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*

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

FctTypeInst::*id* : *Id*
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*.

³In [Jon86] these are called μ -functions.

3.3.13 Lambda Expression

Lambda::= *parm* : *Par*
 body : *Expr*

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

3.3.14 Is Expression

IsExpr::= *tag* : *ValueId* | *BasicType*
 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*.

3.3.15 Literals

Literal = *NumLit* | *BoolLit* | *CharLit* | *TextLit* | *QuoteLit* | *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* :

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

3.3.16 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).

3.4 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.

3.5 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.

3.5.1 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
```

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*.

3.5.2 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*.

3.5.3 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.

3.5.4 Loop Statements

```

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

```

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*.

3.5.5 Non-Deterministic Sequences

```
NonDetStmt ::= stmts : Stmt+
```

NonDetStmt is a ‘non-deterministic statement’. It consists of a non-empty sequence of statements⁴, *stmts*. It denotes an arbitrary ordering of execution of these statements.

3.5.6 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 nil if 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.

3.5.7 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
```

⁴The 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.

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.

3.6 Patterns and Bindings

Pattern = *PatternId* | *MatchVal* | *SqPattern* | *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.

SqPattern = *SqEnumPattern* | *SqConcPattern*

SqPattern is a ‘sequence pattern’. It is either a ‘sequence enumeration pattern’ or a ‘sequence concatenation pattern’.

SqEnumPattern :: *els* : *Pattern**

SqEnumPattern is a ‘sequence enumeration pattern’. It consists of a sequence of patterns, *els*.

SqConcPattern :: *lp* : *Pattern*
rp : *Pattern*

SqConcPattern 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*

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.

Chapter 4

Dynamic Semantic Domains

This chapter first introduces the domain universe and then defines domain operators — operators which can be used in the definition of the dynamic semantics¹. 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.

4.1 The Domain Universe

The domain universe which is introduced in this section provides denotations for all values expressible in VDM-SL. This section 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 section also defines new domain operators² — operators which will be used in the definition of the dynamic semantics.

4.1.1 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 section reviews the related basic concepts which underlie and motivate the construction of the VDM-SL domain universe.

Complete Partial Orders and Fixed Point Definitions

Definition 4.1.1 (Partial Ordering) A binary relation \sqsubseteq on D is called a *partial ordering* on D iff it is:

1. Reflexive: for all $a \in D$, $a \sqsubseteq a$.
2. Antisymmetric: for all $a, b \in D$, $a \sqsubseteq b$ and $b \sqsubseteq a$ imply $a = b$.
3. Transitive: for all $a, b, c \in D$, $a \sqsubseteq b$ and $b \sqsubseteq c$ imply $a \sqsubseteq c$.

Definition 4.1.2 (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 4.1.3 (Upper bound) An *upper bound* of a set $X \subseteq |A|$ is an element $a \in |A|$ such that for all $x \in X$, $x \sqsubseteq_A a$.

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

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

Definition 4.1.4 (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 4.1.5 (Least element (bottom)) An element $\perp_A = \bigsqcup_A \emptyset$, if it exists, is called the *least element* of A .

Definition 4.1.6 (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. $a_0 \sqsubseteq_A a_1 \sqsubseteq_A \dots$).

Definition 4.1.7 (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 .

Definition 4.1.8 (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 4.1.9 For any cpo A and continuous function $f: |A| \rightarrow |A|$, the least upper bound:

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

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

- $f^0(\perp_A) = \perp_A$, and
- $f^{n+1}(\perp_A) = f(f^n(\perp_A))$ 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 $\text{Y}f$ where Y is a least fixed point operator yielding the least solution.

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 4.1.10 (Lifting) For any set S , the result of *lifting* S is a cpo S_\perp defined by:

- $|S_\perp| = S \cup \{\perp\}$
- for $a_1, a_2 \in |S_\perp|$, $a_1 \sqsubseteq_{S_\perp} a_2$ iff $a_1 = \perp$ 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 4.1.11 (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 4.1.12 (Union) Let \mathcal{A} be a union-compatible family of cpo's with for all A in $\mathcal{A} \perp_A = \perp$. Then the *union*, $\mathcal{U}_{CPO}(\mathcal{A})$, is defined by

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

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

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

Definition 4.1.13 (Finite subsets) Let A be a flat cpo with $\perp = \perp_A$. Then the cpo of its *finite subsets*, $\mathcal{S}_{CPO}(A)$, is defined as follows:

- $|\mathcal{S}_{CPO}(A)| = \mathbb{F}(|A| \setminus \{\perp\}) \cup \{\perp\}$,
- for $s_1, s_2 \in |\mathcal{S}_{CPO}(A)|$, $s_1 \sqsubseteq_{\mathcal{S}_{CPO}(A)} s_2$ iff $s_1 = \perp$ or $s_1 = s_2$.

Definition 4.1.14 (Cartesian product) Let A_1, \dots, A_n be cpo's with $\perp = \perp_{A_1} = \dots = \perp_{A_n}$. Then their *smashed Cartesian product*, $\mathcal{P}_{CPO}(A_1, \dots, A_n)$, is defined by:

- $|\mathcal{P}_{CPO}(A_1, \dots, A_n)| = \bigtimes_{i=1}^n (|A_i| \setminus \{\perp\}) \cup \{\perp\}$
- $\perp \sqsubseteq_{\mathcal{P}_{CPO}(A_1, \dots, A_n)} p$ for all $p \in |\mathcal{P}_{CPO}(A_1, \dots, A_n)|$, and
for $(a_1, \dots, a_n), (a'_1, \dots, a'_n) \in |\mathcal{P}_{CPO}(A_1, \dots, A_n)|$,
 $(a_1, \dots, a_n) \sqsubseteq_{\mathcal{P}_{CPO}(A_1, \dots, A_n)} (a'_1, \dots, a'_n)$ iff $a_1 \sqsubseteq_{A_1} a'_1$ and \dots and $a_n \sqsubseteq_{A_n} a'_n$.

Definition 4.1.15 (Finite Sequences) Let A be a cpo with $\perp = \perp_A$. The cpo of finite sequences of elements of A , $\mathcal{L}_{CPO}(A)$, is defined by

- $|\mathcal{L}_{CPO}(A)| = \mathbb{L}(|A| \setminus \{\perp\}) \cup \{\perp\}$
- $\perp \sqsubseteq_{\mathcal{L}_{CPO}(A)} l$ for all $l \in |\mathcal{L}_{CPO}(A)|$, and
for $l_1, l_2 \in |\mathcal{L}_{CPO}(A)| \setminus \{\perp\}$, $l_1 \sqsubseteq_{\mathcal{L}_{CPO}(A)} l_2$ iff $\underline{\text{len}}(l_1) = \underline{\text{len}}(l_2)$ and
for $i \in \{1, \dots, \underline{\text{len}}(l_1)\}$, $l_1(i) \sqsubseteq_A l_2(i)$.

Definition 4.1.16 (Record space) Let $id \in Id$ be a VDM-SL identifier, and let A_1, \dots, A_n be cpo's with $\perp = \perp_{A_1} = \dots = \perp_{A_n}$. Then the *smashed record cpo*, $\mathcal{R}_{CPO}^{id}(A_1, \dots, A_n)$, is defined by:

- $|\mathcal{R}_{CPO}^{id}(A_1, \dots, A_n)| = (\{id\} \times \bigtimes_{i=1}^n (|A_i| \setminus \{\perp\})) \cup \{\perp\}$
- $\perp \sqsubseteq_{\mathcal{R}_{CPO}^{id}(A_1, \dots, A_n)} r$ for all $r \in |\mathcal{R}_{CPO}^{id}(A_1, \dots, A_n)|$, and
for $(id, a_1, \dots, a_n), (id, a'_1, \dots, a'_n) \in |\mathcal{R}_{CPO}^{id}(A_1, \dots, A_n)|$,
 $(id, a_1, \dots, a_n) \sqsubseteq_{\mathcal{R}_{CPO}^{id}(A_1, \dots, A_n)} (id, a'_1, \dots, a'_n)$ iff $a_1 \sqsubseteq_{A_1} a'_1$ and \dots 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 4.1.17 (Mapping space) Let A be a flat cpo and B be a cpo with $\perp = \perp_A = \perp_B$. Then the *cpo of smashed mappings* from A to B , $\mathcal{M}_{CPO}(A, B)$, is defined as follows:

- $|\mathcal{M}_{CPO}(A, B)| = \mathbb{M}(|A| \setminus \{\perp\}, |B| \setminus \{\perp\}) \cup \{\perp\}$
- $\perp \sqsubseteq_{\mathcal{M}_{CPO}(A, B)} m$ for all $m \in |\mathcal{M}_{CPO}(A, B)|$, and
for $m_1, m_2 \in |\mathcal{M}_{CPO}(A, B)| \setminus \{\perp\}$, $m_1 \sqsubseteq_{\mathcal{M}_{CPO}(A, B)} m_2$ 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 $D_1 \rightarrow D_2$. The definition is like the usual definition for cpo's with the addition of \perp as a new least element.

Definition 4.1.18 (Function space) Let A and B be cpo's with $\perp = \perp_A = \perp_B$. Then the *cpo of functions* from A to B , $\mathcal{F}_{CPO}(A, B)$, is defined by:

- $|\mathcal{F}_{CPO}(A, B)| = (|A| \rightarrow |B|) \cup \{\perp\}$

- $\perp \sqsubseteq_{\mathcal{F}_{CPO}(A,B)} f$ for all $f \in |\mathcal{F}_{CPO}(A,B)|$, and for $f, g \in |\mathcal{F}_{CPO}(A,B)| \setminus \{\perp\}$, $f \sqsubseteq_{\mathcal{F}_{CPO}(A,B)} g$ 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 4.1.19 (Tagging) Let A be a cpo with $\perp = \perp_A$. For any $t \in TAG$, *tagging* A with t yields a cpo $T_{CPO}^t(A)$ defined as follows:

- $|T_{CPO}^t(A)| = \{(t, a) \mid a \in (|A| \setminus \{\perp\})\} \cup \{\perp\}$
- $\perp \sqsubseteq_{T_{CPO}^t(A)} e$ for all $e \in |T_{CPO}^t(A)|$, and
for $(t, a_1), (t, a_2) \in |T_{CPO}^t(A)|$, $(t, a_1) \sqsubseteq_{T_{CPO}^t(A)} (t, a_2)$ iff $a_1 \sqsubseteq_A a_2$.

Here TAG is taken to be an abbreviation:

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

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

4.1.2 A Universe of Complete Partial Orders

In this section 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 section 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 4.1.20 (Cpo universe) The basic cpo's are:

$$\begin{aligned} \text{Bool-cpo} &= T_{CPO}^{\text{'bool'}}(\mathbb{B}_\perp) \\ \text{char-cpo} &= T_{CPO}^{\text{'char'}}(CHAR_\perp) \\ \text{nil-cpo} &= T_{CPO}^{\text{'nil'}}(\{\text{nil}\}_\perp) \\ \text{token-cpo} &= T_{CPO}^{\text{'token'}}(QUOTE_\perp) \\ \text{NUM-CPOS} &= \{T_{CPO}^{\text{'num'}}(\mathbb{N}_\perp), T_{CPO}^{\text{'num'}}(\mathbb{N}_{1\perp}), T_{CPO}^{\text{'num'}}(\mathbb{Z}_\perp), T_{CPO}^{\text{'num'}}(\mathbb{Q}_\perp), T_{CPO}^{\text{'num'}}(\mathbb{R}_\perp)\} \\ \text{QUOTE-CPOS} &= \{T_{CPO}^{\text{'quot'}}(\{q\}_\perp) \mid q \in QUOTE\} \end{aligned}$$

$CHAR$ 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:

- $U_0 = \{\text{Bool-cpo}, \text{char-cpo}, \text{nil-cpo}, \text{token-cpo}\} \cup \text{NUM-CPOS} \cup \text{QUOTE-CPOS} \cup \{\emptyset_\perp\}$
- for any ordinal $\alpha < \omega_1$,

$$\begin{aligned} U_{\alpha+1} = U_\alpha \cup & \{U_{CPO}(\{D_1, \dots, D_n\}) \mid D_1, \dots, D_n \in U_\alpha \wedge D_1, \dots, D_n \text{ is a union-compatible family}\} \\ & \cup \{T_{CPO}^{\text{'set'}}(S_{CPO}(D)) \mid D \in U_\alpha \wedge D \text{ is flat}\} \\ & \cup \{T_{CPO}^{\text{'tuple'}}(\mathcal{P}_{CPO}(D_1, \dots, D_n)) \mid D_1, \dots, D_n \in U_\alpha\} \\ & \cup \{T_{CPO}^{\text{'seq'}}(\mathcal{L}_{CPO}(D)) \mid D \in U_\alpha\} \\ & \cup \{T_{CPO}^{\text{'record'}}(\mathcal{R}_{CPO}^{id}(D_1, \dots, D_n)) \mid id \in Id \wedge D_1, \dots, D_n \in U_\alpha\} \\ & \cup \{T_{CPO}^{\text{'map'}}(\mathcal{M}_{CPO}(D_1, D_2)) \mid D_1, D_2 \in U_\alpha \wedge D_1 \text{ is flat}\} \\ & \cup \{T_{CPO}^{\text{'fun'}}(\mathcal{F}_{CPO}(D_1, D_2)) \mid D_1, D_2 \in U_\alpha\} \\ & \cup \{T_{CPO}^q(D) \mid D \in U_\alpha, q \in QUOTE\}. \end{aligned}$$

- 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 4.1.21 The cpo universe CPO as defined above contains the basic cpo's, has a least element³, and is closed under all the operators on cpo's defined in section 4.1.1 as well as under unions of countably many, union-compatible sets of cpo's. The proof for this proposition is given in [TW90].

4.1.3 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 4.1.22 (VDM domain) A *VDM domain* is a pair:

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

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

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

Construction 4.1.23 (Domain universe) The universe of domains for VDM, DOM , is defined by:

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

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 section 4.1.1. 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 4.1.24 (VDM domain operators) For each of the operators on CPO defined in section 4.1.1, its extension to a *domain operator* on DOM is:

³The least element is \emptyset_\perp , with respect to the traditional subset inclusion ordering taken pairwise both on the carrier set and the partial order.

$$\begin{aligned}
\text{Bool-dom} &= (\text{Bool-cpo}, \{\text{'bool'}\} \times \mathbb{B}) \\
\text{char-dom} &= (\text{char-cpo}, \{\text{'char'}\} \times \text{CHAR}) \\
\text{nil-dom} &= (\text{nil-cpo}, \{\text{'nil'}\} \times \{\text{nil}\}) \\
\text{token-dom} &= (\text{token-cpo}, \{\text{'token'}\} \times \text{QUOTE}) \\
\text{nat-dom} &= (T_{CPO}^{\text{'num'}}(\mathbb{N}_\perp), \{\text{'num'}\} \times \mathbb{N}) \\
\text{natone-dom} &= (T_{CPO}^{\text{'num'}}(\mathbb{N}_{1\perp}), \{\text{'num'}\} \times \mathbb{N}_1) \\
\text{int-dom} &= (T_{CPO}^{\text{'num'}}(\mathbb{Z}_\perp), \{\text{'num'}\} \times \mathbb{Z}) \\
\text{rat-dom} &= (T_{CPO}^{\text{'num'}}(\mathbb{Q}_\perp), \{\text{'num'}\} \times \mathbb{Q}) \\
\text{real-dom} &= (T_{CPO}^{\text{'num'}}(\mathbb{R}_\perp), \{\text{'num'}\} \times \mathbb{R}) \\
\text{QUOTE-DOMS} &= \{(T_{CPO}^{\text{'quot'}}(\{q\}_\perp), \{(\text{'quot'}, q)\}) \mid q \in \text{QUOTE}\} \\
\mathcal{U}(A) &= (\mathcal{U}_{CPO}(\{\mathcal{D}(A) \mid A \in \mathcal{A}\}), \bigcup \{\|A\| \mid A \in \mathcal{A}\}) \\
\mathcal{S}(A) &= (T_{CPO}^{\text{'set'}}(\mathcal{S}_{CPO}(\mathcal{D}(A))), \{\text{'set'}\} \times \mathbb{F}(\|A\|)) \\
\mathcal{P}(A_1, \dots, A_n) &= (T_{CPO}^{\text{'tuple'}}(\mathcal{P}_{CPO}(\mathcal{D}^*(A_1, \dots, A_n))), \{\text{'tuple'}\} \times \bigtimes_{i=1}^n \|A_i\|) \\
\mathcal{L}(A) &= (T_{CPO}^{\text{'seq'}}(\mathcal{L}_{CPO}(\mathcal{D}(A))), \{\text{'seq'}\} \times \mathbb{L}(\|A\|)) \\
\mathcal{R}^{\text{id}}(A_1, \dots, A_n) &= (T_{CPO}^{\text{'record'}}(\mathcal{R}_{CPO}^{\text{id}}(\mathcal{D}^*(A_1, \dots, A_n))), \{\text{'record'}\} \times (\{\text{id}\} \times \bigtimes_{i=1}^n \|A_i\|)) \\
\mathcal{M}(A, B) &= (T_{CPO}^{\text{'map'}}(\mathcal{M}_{CPO}(\mathcal{D}(A), \mathcal{D}(B))), \{\text{'map'}\} \times \mathbb{M}(\|A\|, \|B\|)) \\
\mathcal{F}(A, B) &= (T_{CPO}^{\text{'fun'}}(\mathcal{F}_{CPO}(\mathcal{D}(A), \mathcal{D}(B))), \\
&\quad \{\text{'fun'}\} \times \{f: |A| \rightarrow |B| \mid \forall a \in \|A\| \cdot f(a) \in |B| \wedge \forall a \in (|A| \setminus \|A\|) \cdot f(a) = \perp\}) \\
\mathcal{T}^t(A) &= (T_{CPO}^t(\mathcal{D}(A)), \{t\} \times \|A\|).
\end{aligned}$$

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

Definition 4.1.25 (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 4.1.26 (Direct summand) A cpo A is a *direct summand* of a cpo B , written $A \sqsubseteq_{DS} B$, if for some cpo D such that $|A| \cap |D| = \{\perp\}$, $B = \mathcal{U}(\{A, D\})$.

Definition 4.1.27 (Subdomain) A domain A is a *subdomain* of a domain B , written $A \sqsubseteq_{DOM} B$, if:

- $(|A|, \sqsubseteq_A) \sqsubseteq_{DS} (|B|, \sqsubseteq_B)$, 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:

1. The partial operators (i.e. \mathcal{S}, \mathcal{M} , and \mathcal{U}) are applied only to domains within their domain of definition.
2. 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.

4.2 The Semantic Domains

This section introduces the semantic domains which are used in the definition of the dynamic semantics. It should be noted that the domains defined in this section 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 section.

This section will first present the basic semantic domains which form the basis for the ‘pure’ models which a specification can denote. These models (ENV_{PURE})⁴ 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.

4.2.1 Basic Semantic Domains

The semantic domains presented in this section 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 chapter. 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. $DOM = \{((|A|, \sqsubseteq_A), \|A\|) \mid ((|A|, \sqsubseteq_A) \in CPO, \|A\| \subseteq |A| \setminus \{\perp\})\}$
4. $OPVAL = \{\{(i_v, i_st, o_st, o_v, m) \mid i_v \in \|t_1\| \cup \{\perp\}, i_st, o_st \in (\|st\| \cup \{\perp, \underline{nil}\}), o_v \in (\|t_2\| \cup \{\perp, \underline{nil}\}), m \in MODE\} \mid t_1, t_2, st \in FLATDOM\}$
5. $FLATDOM = \{d \mid d \in DOM \cdot \forall a_1, a_2 \in |d| \cdot a_1 \sqsubseteq_d a_2 \Leftrightarrow a_1 = \perp \vee a_1 = a_2\}$
6. $FLATVAL = \bigcup\{|d| \mid d \in FLATDOM\}$
7. $POLYVAL = \mathbb{L}_1(DOM) \rightarrow VAL$
8. $POLYDOM = \mathbb{L}_1(DOM) \rightarrow DOM$
9. $MODE = \{\underline{cont}, \underline{exit}, \underline{ret}\}$

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

⁴A model for a specification can also be viewed as an environment for its ‘outside’ world and therefore the term ENV_{PURE} is used.

an operation, an output value⁵, and a flag indicating the mode of the operation⁶. Do also notice that bottom, \perp , is allowed as a normal value. This is necessary in order to give an erratic nondeterministic interpretation.

5. $FLATDOM$: all domains which fulfill the flatness criterion (i.e. $FLATDOM \subseteq DOM$)⁷.
6. $FLATVAL$: all flat values (i.e. $FLATVAL \subseteq VAL$).
7. $POLYVAL$: all polymorphic functions⁸. 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

⁵Notice that if an operation does not return a value, this is modelled in the dynamic semantics by the return of nil.

⁶State, input and output values must all be flat (since sets containing non-flat elements are not contained within the domain universe).

⁷All domains involving the function space operator are ruled out in this way.

⁸The kind of polymorphism which is present is called “first order explicit polymorphism”.

4.2.2 Extended Semantic Domains

This section 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. $ENV = ENV_{PURE} \cup IE(MOD_FUN \cup SEL_FUN \cup CONS_FUN \cup STATE_FUN \cup TAG_ENV \cup LOC_ENV)$
11. $MOD_FUN = IE(VAL) \rightarrow RECORD_VAL \rightarrow VAL$
12. $SEL_FUN = RECORD_VAL \rightarrow VAL$
13. $CONS_FUN = IL(VAL) \rightarrow RECORD_VAL \perp$
14. $STATE_FUN = IF(ExtVarInf) \rightarrow (VAL \cup \{\underline{nil}\} \times VAL \cup \{\underline{nil}\}) \rightarrow IB$
15. $TAG_ENV = IE(CompositeType)$
16. $LOC_ENV = IE(DOM)$

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 values⁹. 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 section 3.2.5 in chapter 3.
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 section 3.2.1 in chapter 3.
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

⁹The domain for record values called *RECORD_VAL* is defined in section 4.2.4.

4.2.3 Semantic Domains for Evaluation Functions

This section 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 evaluators¹⁰. 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. $LDef = \text{IP}(Def)$
- 18. $Def = ENV \rightarrow (ENV \cup \{\underline{\text{err}}\})$
- 19. $TEval = ENV \rightarrow (DOM \cup \{\underline{\text{err}}\})$
- 20. $LSEval = \text{IP}(SEval)$
- 21. $SEval = ENV \rightarrow (ENV \times MODE \times (VAL \cup \{\underline{\text{nil}}\}))$
- 22. $LPatEval = \text{IP}(PatEval)$
- 23. $PatEval = VAL \rightarrow ENV \rightarrow (VEnv \cup \{\underline{\text{err}}, \underline{\text{unmatch}}\})$
- 24. $LBindEval = \text{IP}(BindEval)$
- 25. $BindEval = ENV \rightarrow \text{IP}(VEnv \cup \{\underline{\text{err}}\})$
- 26. $VEnv = \text{IE}(VAL)$
- 27. $LEEval = \text{IP}(EEval)$
- 28. $EEval = ENV \rightarrow VAL$
- 29. $LPEval = \text{IP}(PEval)$
- 30. $PEval = ENV \rightarrow POLYVAL$
- 31. $PDomEval = ENV \rightarrow POLYDOM$

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 environments¹¹ 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

¹⁰Notice that this way of modelling looseness implies that definers and evaluators are deterministic.

¹¹The returning value environments could just as well be described with the domain ENV , but for clarity the auxiliary domain $VEnv$ is introduced.

- a type is erroneous).
- 26. $VEnv$: all value environments are mappings from identifiers to values. An auxiliary domain used only for clarity (i.e. $Venv \subseteq ENV$).
 - 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

4.2.4 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 section 4.1.3) supply the values (and the domain) with the appropriate tags (e.g. ‘map’). These auxiliary semantic domains are listed here.

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

Chapter 5

The Dynamic Semantics

This chapter 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 sections 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 definitions¹.

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 descrip-

¹Thus, a model can also be considered as the environment which the collection of definitions provides for the ‘outside’ world.

tion. 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.

5.1 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. $SemSpec : Document \rightarrow IP(ENV_{PURE})$
- 1.1 $SemSpec(doc) \triangleq$
- 1.2 $\{ env \mid env \in ENV_{PURE} \cdot IsAModelOf(env, doc) \}$

Annotations to *SemSpec*:

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 $IsAModelOf(env, defs) \triangleq$
- 2.2 $\text{let } MkTag('Definitions', t, epf, ef, ipf, if, v, eo, io, s) = defs \text{ in}$
- 2.3 $DisjointIds(t, epf, ef, ipf, if, v, eo, io, s, env) \wedge$
- 2.4 $\text{let } exp_env = ExpandImplDfs(t, s)(env) \text{ in}$
- 2.5 $exp_env \neq \underline{\text{err}} \wedge$
- 2.6 $VerifyTypes(t, exp_env) \wedge$
- 2.7 $VerifyExplPolyFns(epf, exp_env) \wedge$
- 2.8 $VerifyExplFns(ef, exp_env) \wedge$
- 2.9 $VerifyImplPolyFns(ipf, exp_env) \wedge$
- 2.10 $VerifyImplFns(if, exp_env) \wedge$
- 2.11 $VerifyValues(v, exp_env) \wedge$
- 2.12 $VerifyExplOps(eo, exp_env) \wedge$
- 2.13 $VerifyImplOps(io, exp_env) \wedge$
- 2.14 $VerifyStateDef(s, exp_env)$

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

```

3.   DisjointIds : Pred( $\mathbb{E}(TypeDef) \times \mathbb{E}(ExplPolyFnDef) \times \mathbb{E}(ExplFnDef) \times$   

 $\mathbb{E}(ImplPolyFnDef) \times \mathbb{E}(ImplFnDef) \times$   

 $\mathbb{M}(Pattern, ValDef) \times \mathbb{E}(ExplOpDef) \times \mathbb{E}(ImplOpDef) \times$   

 $(StateDef \cup \{\underline{nil}\}) \times ENV_{PURE}$  )

3.1 DisjointIds( $m_1, m_2, m_3, m_4, m_5, pm, m_6, m_7, sd, env$ )  $\triangleq$ 
.2   let  $s_1 = \underline{dom}(m_1),$ 
.3      $s_2 = \underline{dom}(m_2),$ 
.4      $s_3 = \underline{dom}(m_3),$ 
.5      $s_4 = \underline{dom}(m_4),$ 
.6      $s_5 = \underline{dom}(m_5),$ 
.7      $s_6 = \underline{dom}(m_6),$ 
.8      $s_7 = \underline{dom}(m_7),$ 
.9      $ps = \bigcup \{ CollIdSet(p) \mid p \in \underline{dom}(pm) \} \text{ in}$ 
.10    let  $set\_l = [s_1, s_2, s_3, s_4, s_5, s_6, s_7, ps],$ 
.11       $st\_ids = \text{if } sd = \underline{nil}$ 
.12        then { }
.13        else let  $MkTag('StateDef', (MkTag('ValueId', stid), \_, \_)) = sd \text{ in}$ 
.14           $\{MkTag('StateTypeId', stid), MkTag('InitId', stid)\}$ 
.15    in
.16       $(\underline{dom}(env) = \bigcup \underline{elems}(set\_l) \cup st\_ids) \wedge$ 
.17       $(\forall i, j \in \underline{inds}(set\_l) \cdot i \neq j \Rightarrow set\_l(i) \cap set\_l(j) = \{\}) \wedge$ 
.18       $(\forall id_1, id_2 \in \underline{dom}(env) \cdot (ShowTag(id_1) = 'StateTypeId' \wedge$ 
.19         $ShowTag(id_2) = 'StateTypeId') \Rightarrow id_1 = id_2 \wedge$ 
.20         $ShowTag(id_1) \in \{ 'ValueId', 'PreId', 'PostId', 'InvId', 'InitId', 'StateTypeId' \})$ 

```

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

5.2 Definitions

A VDM-SL document (or specification) consists of a collection of definitions. This section 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 section 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).

5.2.1 Type Definitions

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

The Verification Predicate for Types

```

4.   VerifyTypes : Pred( $\mathbb{E}(TypeDef)$  × ENV)
4.1  VerifyTypes(tpdef_m, env)  $\triangleq$ 
    .2  if dom(tpdef_m)  $\subseteq$  dom(env)  $\wedge$   $\forall id \in \underline{\text{dom}}(\text{tpdef}_m) \cdot \text{env}(id) \in \text{DOM}$ 
    .3  then let  $\text{env}' = \text{EvalTypeDef}(\text{tpdef}_m)(\text{subtract}(\text{env}, \underline{\text{dom}}(\text{tpdef}_m)))$  in
    .4  if  $\text{env}' = \text{err}$ 
    .5  then F
    .6  else  $\forall id \in \underline{\text{dom}}(\text{tpdef}_m) \cdot \text{env}(id) = \text{env}'(id)$ 
    .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:

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

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

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

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

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

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

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

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

□

Evaluation Functions for Types

```

5.   EvalTypeDef :  $\mathbb{E}(\text{TypeDef}) \rightarrow \text{Def}$ 
5.1  EvalTypeDef(td_m)(env)  $\triangleq$ 
.2   let id_s = dom(td_m) in
.3   let tev_m = { id  $\mapsto$  EvalType(SelShape(td_m(id))) | id  $\in$  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        $\lambda \text{appr} \in \text{DomApprox}_{\underline{\text{err}}}$  .
.9       if appr = err
.10      then err
.11      else let m =
.12        { id  $\mapsto$  let d = tev_m(id)(overwrite(env, appr)),
.13          env' = overwrite(env, appr) in
.14          if d = err  $\vee$  env' = err
.15          then err
.16          else let inv = itf(env')(SelInvId(td_m(id))) in
.17             $\mathcal{I}(\{ e \in \|d\| \mid \text{inv}(e) = \text{True}() \}, d)$ 
.18            | id  $\in$  id_s } in
.19            if  $\exists \text{id} \in \text{id}_s \cdot m(\text{id}) = \underline{\text{err}}$ 
.20            then err
.21            else m in
.22            if IsContDomApproxerr(d_tf)
.23            then let d_m =  $\Upsilon(d_{tf})$  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 ensure that they are not loosely specified.
- .7-.21 A function $d_tf : DomApprox_{\text{err}} \rightarrow DomApprox_{\text{err}}$ is being defined. The set $DomApprox_{\text{err}} = DomApprox \cup \{\text{err}\}$, where $DomApprox = id_s \rightarrow DOM$ (see .2) has a structure of a cpo with an ordering defined by:
- for $f, g \in DomApprox$ $f \sqsubseteq g$ iff $\forall id \in id_s f(id) \sqsubseteq_{DOM} g(id)$
 - $\lambda id : id_s . (\emptyset_{\perp}, \{\}) \sqsubseteq \text{err}$.
- The function $\lambda id : id_s . (\emptyset_{\perp}, \{\})$ is a bottom element. Note that err is incomparable with any element of $DomApprox$ except bottom.
- .22-.26 If d_tf is continuous with respect to the ordering on $DomApprox_{\text{err}}$, 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. $EvalType : Type \rightarrow TEval$
- 6.1 $EvalType(tp)(env) \triangleq$
- 6.2 **cases** $ShowTag(tp) :$
- 6.3 '*BasicType*' $\rightarrow EvalBasicType(tp)(env),$
- 6.4 '*CompositeType*' $\rightarrow EvalCompositeType(tp)(env),$
- 6.5 '*ProductType*' $\rightarrow EvalProductType(tp)(env),$
- 6.6 '*UnionType*' $\rightarrow EvalUnionType(tp)(env),$
- 6.7 '*OptionalType*' $\rightarrow EvalOptionalType(tp)(env),$
- 6.8 '*SetType*' $\rightarrow EvalSetType(tp)(env),$
- 6.9 '*Seq0Type*' $\rightarrow EvalSeq0Type(tp)(env),$
- 6.10 '*Seq1Type*' $\rightarrow EvalSeq1Type(tp)(env),$
- 6.11 '*GeneralMapType*' $\rightarrow EvalGeneralMapType(tp)(env),$
- 6.12 '*InjectiveMapType*' $\rightarrow EvalInjectiveMapType(tp)(env),$
- 6.13 '*FnType*' $\rightarrow EvalFnType(tp)(env),$
- 6.14 '*TypeId*' $\rightarrow EvalTypeId(tp)(env),$
- 6.15 '*TypeVar*' $\rightarrow EvalTypeVar(tp)(env),$
- 6.16 '*QuoteType*' $\rightarrow EvalQuoteType(tp)(env)$

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 $DOM \cup \{\text{err}\}$).
- .3-.16 The different kinds of type are evaluated by means of sub-functions.

End of annotations

7. $EvalBasicType : BasicType \rightarrow TEval$
- 7.1 $EvalBasicType(MkTag('BasicType', tp))(env) \triangleq$
- 7.2 **cases** $tp :$
- 7.3 NATONE $\rightarrow \text{natone-dom},$
- 7.4 NAT $\rightarrow \text{nat-dom},$
- 7.5 INTEGER $\rightarrow \text{int-dom},$
- 7.6 RAT $\rightarrow \text{rat-dom},$
- 7.7 REAL $\rightarrow \text{real-dom},$
- 7.8 BOOLEAN $\rightarrow \text{Bool-dom},$
- 7.9 CHAR $\rightarrow \text{char-dom},$
- 7.10 TOKEN $\rightarrow \text{token-dom},$
- 7.11 UNIT $\rightarrow \text{nil-dom}$

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

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

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

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

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 . Otherwise an error is reported.

End of annotations

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

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. $EvalSetType : SetType \rightarrow TEval$
- 12.1 $EvalSetType(MkTag('SetType', t))(env) \triangleq$
 - .2 **let** $d = EvalType(t)(env)$ **in**
 - .3 **if** $d = \underline{\text{err}} \vee d \notin FLATDOM$
 - .4 **then** $\underline{\text{err}}$
 - .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. $EvalSeq0Type : Seq0Type \rightarrow TEval$
- 13.1 $EvalSeq0Type(MkTag('Seq0Type', t))(env) \triangleq$
 - .2 **let** $d = EvalType(t)(env)$ **in**
 - .3 **if** $d = \underline{\text{err}}$
 - .4 **then** $\underline{\text{err}}$
 - .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

14. $\text{EvalSeq1Type} : \text{Seq1Type} \rightarrow \text{TEval}$

14.1 $\text{EvalSeq1Type}(\text{MkTag}(\text{'Seq1Type'}, t))(env) \triangleq$
 .2 let $d = \text{EvalType}(\text{MkTag}(\text{'Seq0Type'}, t))(env)$ 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 *Seq1Types*, 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. $\text{EvalGeneralMapType} : \text{GeneralMapType} \rightarrow \text{TEval}$

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

Annotations to $\text{EvalGeneralMapType}$:

15. This function gives meaning to *GeneralMapTypes*, 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. $\text{EvalInjectiveMapType} : \text{InjectiveMapType} \rightarrow \text{TEval}$

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

Annotations to $\text{EvalInjectiveMapType}$:

16. This function gives meaning to *InjectiveMapTypes* 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

17. $EvalFnType : FnType \rightarrow TEval$
- 17.1 $EvalFnType(MkTag('FnType', (t_1, t_2)))(env) \triangleq$
 .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. $EvalTypeId : TypeId \rightarrow TEval$
- 18.1 $EvalTypeId(MkTag('TypeId', id))(env) \triangleq$
 .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. $EvalTypeVar : TypeVar \rightarrow TEval$
- 19.1 $EvalTypeVar(MkTag('TypeVar', id))(env) \triangleq$
 .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. $EvalQuoteType : QuoteType \rightarrow TEval$
- 20.1 $EvalQuoteType(MkTag('QuoteType', q))(env) \triangleq$
 .2 $(T_{CPO}^{quot}(\{q\}_{\perp}), \{MkTag('quot', q)\})$

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

21. $EvalTypeList : \mathbb{L}(Type) \rightarrow ENV \rightarrow \mathbb{L}(DOM) \cup \{\underline{err}\}$
 21.1 $EvalTypeList(t_l)(env) \triangleq$
 .2 $\text{if } t_l = []$
 .3 $\text{then } []$
 .4 $\text{else let } d = EvalType(\text{hd}(t_l))(env),$
 .5 $d_l = EvalTypeList(\text{tl}(t_l))(env) \text{ in}$
 .6 $\text{if } d = \underline{err} \vee d_l = \underline{err}$
 .7 $\text{then } \underline{err}$
 .8 $\text{else } \underline{\text{join}}([d], d_l)$

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. $EvalTypeSet : \mathbb{F}(Type) \rightarrow ENV \rightarrow \mathbb{F}(DOM) \cup \{\underline{err}\}$
 22.1 $EvalTypeSet(t_s)(env) \triangleq$
 .2 $\text{if } t_s = \{\}$
 .3 $\text{then } \{\}$
 .4 $\text{else let } t = Choose(t_s) \text{ in}$
 .5 $\text{let } d = EvalType(t)(env),$
 .6 $d_s = EvalTypeSet(t_s \setminus \{t\})(env) \text{ in}$
 .7 $\text{if } d = \underline{err} \vee d_s = \underline{err}$
 .8 $\text{then } \underline{err}$
 .9 $\text{else } \{d\} \cup d_s$

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. $EvalTypeMap : \mathbb{E}(Type) \rightarrow ENV \rightarrow \mathbb{E}(DOM) \cup \{\underline{err}\}$
 23.1 $EvalTypeMap(t_m)(env) \triangleq$
 .2 $\text{let } id_s = \underline{\text{dom}}(t_m) \text{ in}$
 .3 $\text{if } \exists id \in id_s \cdot EvalType(t_m(id))(env) = \underline{err}$
 .4 $\text{then } \underline{err}$
 .5 $\text{else } \{id \mapsto EvalType(t_m(id))(env) \mid id \in id_s\}$

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

5.2.2 State Definition

This subsubsection 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

5.2.3 Value Definitions

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

The Verification Predicate for Value Definitions

```

25.   VerifyValues : Pred(IM(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:

```

 $\lambda n : \mathbb{N} .$ 
  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*):

```

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

$$(1) \ fac_1 = \lambda n. \perp$$

$$(2) \ fac_1 = \lambda n. \left\{ \begin{array}{l} n = 0 \Rightarrow 1 \\ n > 0 \Rightarrow \perp \end{array} \right.$$

$$(3) \ fac_1 = \lambda n. \left\{ \begin{array}{l} n = 0 \Rightarrow 1 \\ n = 1 \Rightarrow 1 \\ n > 1 \Rightarrow \perp \end{array} \right.$$

$$(4) \ fac_1 = \lambda n. \left\{ \begin{array}{l} n = 0 \Rightarrow 1 \\ n = 1 \Rightarrow 1 \\ n = 2 \Rightarrow 2 \\ n > 2 \Rightarrow \perp \end{array} \right.$$

$$(5) \ fac_1 = \lambda n. \left\{ \begin{array}{l} n = 0 \Rightarrow 1 \\ n = 1 \Rightarrow 1 \\ n = 2 \Rightarrow 2 \\ n = 3 \Rightarrow 6 \\ n > 3 \Rightarrow \perp \end{array} \right.$$

⋮

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

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

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:

```

 $fac_2 : \mathbb{N} \rightarrow \mathbb{N}$ 
 $fac_2(n) \triangleq$ 
  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 choose to bind x to 1 while $fac_2(1)$ may choose 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:

```

 $\lambda n : \mathbb{N} .$ 
  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:

```

{  $\lambda env . \lambda n .$  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 by²: $\lambda n . 2^{n+1-k} \times fac(n)$ 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 ($2 \times 2!$), 8 ($2 \times 2 \times 2!$) and 16 ($2 \times 2 \times 2 \times 2!$) in the different models for fac_2 .

□

The Evaluation Function for Value Definitions

The function presented in this subsubsection is made so general that monomorphic function definitions are also handled as a kind of value definitions.

²Note 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.

```

26.   EvalExplDef :  $\text{IM}(\text{Pattern}, (\text{ExplFnDef} \cup \text{ValDef})) \rightarrow \text{LDef}$ 
26.1  EvalExplDef(def_m)  $\triangleq$ 
.2   let  $p\_s = \underline{\text{dom}}(\text{def}_m)$  in
.3   let  $lev\_m = \{ p \mapsto \text{EvalExpr}(\text{SelBody}(\text{def}_m(p))) \mid p \in p\_s \}$  in
.4   let  $ev\_m\_s = \{ m \mid m \in \text{IM}(\text{Pattern}, \text{EEval}) \cdot$ 
.5    $\underline{\text{dom}}(m) = p\_s \wedge$ 
.6    $\forall p \in p\_s \cdot m(p) \in lev\_m(p) \}$  in
.7   let  $plev\_m = \{ p \mapsto \text{EvalPattern}(p) \mid p \in p\_s \}$  in
.8   let  $pev\_m\_s = \{ m \mid m \in \text{IM}(\text{Pattern}, \text{PatEval}) \cdot$ 
.9    $\underline{\text{dom}}(m) = p\_s \wedge$ 
.10   $\forall p \in p\_s \cdot m(p) \in plev\_m(p) \}$  in
.11  {  $\lambda env. \text{let } tf = \lambda appr. ObjApprox_{\underline{\text{err}}}.$ 
.12    if  $appr = \underline{\text{err}}$ 
.13    then  $\underline{\text{err}}$ 
.14    else let  $venv\_s = \{ \text{let } pev = pev\_m(p),$ 
.15       $ev = ev\_m(p),$ 
.16       $env' = \underline{\text{overwrite}}(env, appr) \text{ in}$ 
.17       $pev(ev(env'))(env)$ 
.18       $| p \in p\_s \}$  in
.19      if  $\{ \underline{\text{err}}, \underline{\text{unmatch}} \} \subseteq venv\_s \vee \neg \text{Compatible}(venv\_s)$ 
.20      then  $\underline{\text{err}}$ 
.21      else  $\underline{\text{Merge}}(venv\_s) \text{ in}$ 
.22      if  $\text{IsContObjApprox}_{\underline{\text{err}}}(tf)$ 
.23      then let  $lfp\_m = \Upsilon(tf) \text{ in}$ 
.24        if  $lfp\_m = \underline{\text{err}}$ 
.25        then  $\underline{\text{err}}$ 
.26        else  $\underline{\text{overwrite}}(env, lfp\_m)$ 
.27        else  $\underline{\text{err}}$ 
.28      |  $pev\_m \in pev\_m\_s, ev\_m \in ev\_m\_s \}$ 

```

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 $tf : ObjApprox_{\underline{\text{err}}} \rightarrow ObjApprox_{\underline{\text{err}}}$ is defined. The set, $ObjApprox_{\underline{\text{err}}}$, is a union $ObjApprox \cup \{ \underline{\text{err}} \}$, where $ObjApprox \subseteq DefId \rightarrow VAL$ and $DefId = \bigcup \{ ColIdSet(p) \mid p \in p_s \}$ (see .2 of *EvalLocalDef*). The set, $ObjApprox$, has the following property:
 $\forall id \in DefId \cdot \exists D_{id} \in DOM$.
 $\forall f \in ObjApprox \cdot f(id) \in |D_{id}|$.
- The set $ObjApprox_{\underline{\text{err}}}$ has a structure of a cpo with an ordering \sqsubseteq defined by:
- for $f, g \in ObjApprox$ $f \sqsubseteq g$ iff $\forall id \in DefId \cdot f(id) \sqsubseteq_{D_{id}} g(id)$
 - $\perp \sqsubseteq \underline{\text{err}}$
- The function $\lambda id : DefId \cdot \perp$ is a bottom element. Note that $\underline{\text{err}}$ is incomparable with any element of $ObjApprox$ except bottom.
- .22 – .27 If tf is continuous with respect to the ordering on $ObjApprox_{\underline{\text{err}}}$, then the least fixed point of tf is calculated. If it is different from $\underline{\text{err}}$, 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

5.2.4 Function Definitions

This subsubsection 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 section about value definitions.

Verification Predicates for Function Definitions

```

27.   VerifyExplPolyFns : Pred( $\text{IE}(\text{ExplPolyFnDef}) \times \text{ENV}$ )
27.1  VerifyExplPolyFns(epfd_m, env)  $\triangleq$ 
.2    let id_s = dom(epfd_m) in
.3    if id_s  $\subseteq$  dom(env)  $\wedge$   $\forall id \in id_s \cdot \text{env}(id) \in \text{POLYVAL}$ 
.4    then let ldef = EvalPolyDef(epfd_m) in
.5      let env_s = { def(subtract(env, id_s)) | def  $\in$  ldef } \ { err },
.6      p_lev_f =  $\lambda id \in id_s. \text{EvalExpr}(\text{SelPre}(epfd_m(id)))$  in
.7      let p_ev_f_s = PropE({ p_lev_f }) in
.8        let tp_l_m = { id  $\mapsto$  ColTypeList(ColParList(epfd_m(id))) | id  $\in$  id_s },
.9        fn_d_m = { id  $\mapsto$  EvalType(GetType(epfd_m(id))) | id  $\in$  id_s },
.10       id_l_m = { id  $\mapsto$  ColIdList(ColParList(epfd_m(id))) | id  $\in$  id_s } in
.11        $\exists env' \in env_s .$ 
.12        $\exists p_{\text{ev}}f \in p_{\text{ev}}f_s .$ 
.13        $\forall id \in id_s .$ 
.14        $\forall d_l \in \text{IL}(\text{DOM}) .$ 
.15       len(d_l) = len(ColTypeVars(epfd_m(id)))  $\Rightarrow$ 
.16       ( $\forall i \in \text{inds}(tp_l_m(id)) \cdot \text{CheckTagEnv}(tp_l_m(id)(i))$ )  $\wedge$ 
.17       let env'' = ExtDomEnv(SelTypeVars(epfd_m(id)), d_l) in
.18       if env''  $\neq$  err
.19       then let d_m = { id  $\mapsto$  EvalType(GetType(epfd_m(id)))
.20           | id  $\in$  id_s },
.21           d_l_m = { id  $\mapsto$  EvalTypeList(tp_l_m(id))
.22             (overwrite(env, env'')) |
.23             | id  $\in$  id_s } in
.24           if env'(id)  $\in$  POLYVAL  $\wedge$ 
.25             env'(id)(d_l)  $\in$  FUN_VAL  $\wedge$ 
.26               d_m(id)  $\neq$  err  $\wedge$  env(id)(d_l)  $\in$  ||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                 f_1 = env'(id)(d_l),
.32                 f_2 = env(id)(d_l) in
.33                 VerifyCurFn(id_l, d_l', f_1, f_2, 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 .

- .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 : Pred( $\mathbb{E}(ExplFnDef) \times ENV$ )
28.1  VerifyExplFns( $efd\_m, env$ )  $\triangleq$ 
.2    let  $id\_s = \underline{\text{dom}}(efd\_m)$  in
.3      if  $id\_s \subseteq \underline{\text{dom}}(env) \wedge \forall id \in id\_s \cdot env(id) \in FUN\_VAL$ 
.4      then let  $ldef = EvalExplDef(\{ Mktag('PatternId', id) \mapsto efd\_m(id) \mid id \in id\_s \})$  in
.5        let  $env\_s = \{ def(\underline{\text{subtract}}(env, id\_s)) \mid def \in ldef \} \setminus \{ \underline{\text{err}} \}$ ,
.6           $p\_lev.f = \lambda id \in id\_s. EvalExpr(SelPre(efd\_m(id)))$  in
.7        let  $p\_ev.f\_s = PropE(\{ p\_lev.f \})$  in
.8          let  $tp\_L.m = \{ id \mapsto ColTypeList(ColParList(efd\_m(id))) \mid id \in id\_s \}$ ,
.9             $fn\_d.m = \{ id \mapsto EvalType(GetType(efd\_m(id))) \mid id \in id\_s \}$ ,
.10            $id\_L.m = \{ id \mapsto ColIdList(ColParList(efd\_m(id))) \mid id \in id\_s \}$  in
.11           if  $\forall id \in id\_s. EvalTypeList(tp\_L.m(id))(env) \neq \underline{\text{err}} \wedge fn\_d.m(id) \neq \underline{\text{err}}$ 
.12           then let  $d\_L.m = \{ id \mapsto EvalTypeList(tp\_L.m(id))(env) \mid id \in id\_s \}$  in
.13              $\exists env' \in env\_s .$ 
.14                $\exists p\_ev.f \in p\_ev.f\_s .$ 
.15                  $\forall id \in id\_s .$ 
.16                    $(\forall i \in \underline{\text{inds}}(tp\_L.m(id)) \cdot CheckTagEnv(tp\_L.m(id)(i))) \wedge$ 
.17                     if  $env(id) \in \|fn\_d.m(id)(env)\| \wedge env'(id) \in FUN\_VAL$ 
.18                     then let  $id\_l = id\_L.m(id)$ ,
.19                          $d\_l = d\_L.m(id)$ ,
.20                            $ev = p\_ev.f(id)$ ,
.21                            $kind = SelKind(efd\_m(id))$ ,
.22                            $f_1 = env'(id)$ ,
.23                            $f_2 = env(id)$  in
.24                              $VerifyCurFn(id\_l, d\_l, f_1, f_2, ev, kind, env)$ 
.25                           else F
.26                         else F
.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 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.    $VerifyImplPolyFns : \text{Pred}(\text{IE}(ImplPolyFnDef) \times ENV)$ 
29.1   $VerifyImplPolyFns(ipfd\_m, env) \triangleq$ 
.2    let  $id\_s = \underline{\text{dom}}(ipfd\_m)$  in
.3    if  $id\_s \subseteq \underline{\text{dom}}(env) \wedge \forall id \in id\_s . env(id) \in POLYVAL$ 
.4    then let  $pre\_ev\_m\_s = \text{MakeEvalMapSet}(ipfd\_m)(\lambda(id, ipfd). SelPre(ipfd))$ ,
.5         $post\_ev\_m\_s = \text{MakeEvalMapSet}(ipfd\_m)(\lambda(id, ipfd). SelPost(ipfd))$ ,
.6         $pfn\_m\_s = \text{MakePolyFns}(ipfd\_m)(env)$  in
.7         $pfn\_m\_s \neq \underline{\text{err}}$   $\wedge$ 
.8         $\exists pre\_ev\_m \in pre\_ev\_m\_s .$ 
.9         $\exists post\_ev\_m \in post\_ev\_m\_s .$ 
.10        $\exists pfn\_m \in pfn\_m\_s .$ 
.11        $\forall id \in id\_s .$ 
.12        $\forall d\_l \in \underline{\text{IL}}(DOM) .$ 
.13        $\underline{\text{len}}(d\_l) = \underline{\text{len}}(\text{ColTypeVars}(ipfd\_m(id))) \Rightarrow$ 
.14        $env(id) = pfn\_m(id) \wedge$ 
.15        $\forall ext\_m \in ExtImplPolyFnEnv(ipfd\_m(id))(env)(d\_l) .$ 
.16        $ext\_m \neq \underline{\text{err}}$   $\wedge$ 
.17       let  $env' = \underline{\text{overwrite}}(env, ext\_m)$ ,
.18            $ext = \{ SelRes(ipfd\_m(id)) \mapsto$ 
.19              $\text{GenPolyApply}(env(id), d\_l, Iota(\underline{\text{rng}}(ext\_m))) \}$  in
.20       let  $ext\_env = \underline{\text{overwrite}}(env', ext)$  in
.21       let  $pre = pre\_ev\_m(id)$ ,
.22            $post = post\_ev\_m(id)$  in
.23           if  $pre \notin POLYVAL \vee post \notin POLYVAL$ 
.24           then F
.25           else  $(pre(d\_l)(env') = \text{False}()) \vee$ 
.26              $(pre(d\_l)(env') = \text{True}()) \wedge$ 
.27              $post(d\_l)(ext\_env) = \text{True}()$ 
.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 $\text{IP}(\text{IE}(EEval))$.
- .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 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 : Pred( $\text{IE}(\text{ImplFnDef}) \times \text{ENV}$ )
30.1  VerifyImplFns( $\text{ifd\_m}$ ,  $\text{env}$ )  $\triangleq$ 
    .2  let  $\text{id\_s} = \underline{\text{dom}}(\text{ifd\_m})$  in
        .3  if  $\text{id\_s} \subseteq \underline{\text{dom}}(\text{env}) \wedge \forall id \in \text{id\_s} \cdot \text{env}(id) \in \text{FUN\_VAL}$ 
        .4  then let  $\text{pre\_ev\_m\_s} = \text{MakeEvalMapSet}(\text{ifd\_m})(\lambda(id, \text{ifd}). \text{SelPre}(\text{ifd}))$ ,
            .5   $\text{post\_ev\_m\_s} = \text{MakeEvalMapSet}(\text{ifd\_m})(\lambda(id, \text{ifd}). \text{SelPost}(\text{ifd}))$ ,
            .6   $\text{app\_ev\_m\_s} = \text{MakeEvalMapSet}(\text{ifd\_m})(\lambda(id, \text{ifd}). \text{GetFnApp}(id, \text{ifd}))$ ,
            .7   $\text{ext\_m\_s} = \text{ExtImplFnsEnv}(\text{ifd\_m})(\text{env})$ ,
            .8   $\text{fn\_m\_s} = \text{MakeFns}(\text{ifd\_m})(\text{env})$  in
        .9   $\text{fn\_m\_s} \neq \underline{\text{err}}$   $\wedge$ 
            .10   $\exists \text{pre\_ev\_m} \in \text{pre\_ev\_m\_s} \cdot$ 
            .11   $\exists \text{post\_ev\_m} \in \text{post\_ev\_m\_s} \cdot$ 
            .12   $\exists \text{app\_ev\_m} \in \text{app\_ev\_m\_s} \cdot$ 
            .13   $\exists \text{fn\_m} \in \text{fn\_m\_s} \cdot$ 
            .14   $\forall \text{ext\_m} \in \text{ext\_m\_s} \cdot$ 
            .15   $\forall id \in \text{id\_s} \cdot$ 
            .16   $\text{env}(id) = \text{fn\_m}(id) \wedge$ 
            .17  let  $\text{env}' = \underline{\text{overwrite}}(\text{env}, \text{ext\_m}(id))$  in
            .18  let  $\text{ext} = \{ \text{SelRes}(\text{ifd\_m}(id)) \mapsto \text{app\_ev\_m}(id)(\text{env}') \}$  in
            .19  let  $\text{ext\_env} = \underline{\text{overwrite}}(\text{env}', \text{ext})$  in
            .20   $(\text{pre\_ev\_m}(id)(\text{env}') = \text{False}()) \vee$ 
            .21   $(\text{pre\_ev\_m}(id)(\text{env}') = \text{True}()) \wedge$ 
            .22   $\text{post\_ev\_m}(id)(\text{ext\_env}) = \text{True}()$ 
    .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 $\text{IP}(\text{EEval})$.
- .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).
- .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( $\mathbb{L}(Id)$  × ( $\mathbb{L}(DOM)$  ∪ {err}) ×  $FUN\_VAL$  ×  $FUN\_VAL$  ×  $EVal$  × {TOTAL, PARTIAL} ×  $ENV$ )
31.1 VerifyCurFn( $id\_l, d\_l, f_1, f_2, ev, kind, env$ ) ≡
.2   if  $d\_l \neq \underline{\text{err}}$  ∧  $EqCurDoms(\underline{\text{len}}(d\_l), f_1, f_2)$ 
.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      $\forall v\_l \in v\_l\_s \cdot \text{let } env' = ExtValEnv(id\_l, v\_l) \text{ in}$ 
.6        $env' \neq \underline{\text{err}}$  ∧
.7        $ev(\underline{\text{overwrite}}(env, env')) = True() \Rightarrow$ 
.8        $((kind = PARTIAL \vee ApplyCurFn(f_1, v\_l) \neq \perp) \wedge$ 
.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 \perp .
- .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 \perp if the pre-condition is fulfilled.

End of annotations

Evaluation Functions for Polymorphic Functions

```

32.   EvalPolyDef : IE(ExplPolyFnDef) → LDef
32.1  EvalPolyDef(poly_m) ≡
.2    let id_s = dom(poly_m) in
.3    let lpolyev_m = { id ↦ MakePolyFnDef(SelPolyFn(poly_m(id))) | id ∈ id_s } in
.4    let polyev_m_s = { m ∈ IE(PEval)
.5      | dom(m) = id_s · ∀id ∈ id_s · m(id) ∈ lpolyev_m(id) } in
.6      { λenv.let id_l_m = { id ↦ ColTypeVars(poly_m(id)) | id ∈ id_s },
.7        tp_m = { id ↦ GetType(poly_m(id)) | id ∈ id_s } in
.8        let pfun = λid ∈ id_s.
.9          PolyVals(EvalPolyType(id_l, tp_m(id))(env)) in
.10         let PolyApprox = PropErr({ pfun }) in
.11         let tf = λapp ∈ PolyApprox.
.12           { id ↦ polyev_m(id)(overwrite(env, app)) | id ∈ id_s } in
.13           if IsContPolyApprox(tf)
.14             then let lfp_m = Y(tf) in
.15               overwrite(env, lfp_m)
.16             else err
.17           | polyev_m ∈ polyev_m_s }

```

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* ⊆ *id_s* → 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* → *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 :  $\mathbb{L}_1(Id) \times Type \rightarrow PDomEval$ 
33.1  EvalPolyType(id_l, t) ≡
.2    λenv . λd_l . if (len(d_l) ≠ len(id_l) ) ∨ (len(id_l) ≠ card(elems(id_l))) )
.3      then (Ø⊥, { })
.4      else let newenv = overwrite(env, ExtDomEnv(id_l, d_l)) in
.5        let d = EvalType(t)(overwrite(env, newenv)) in
.6          if d = err
.7            then (Ø⊥, { })
.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. $PolyVals : POLYDOM \rightarrow \text{IP}(POLYVAL)$
- 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

5.2.5 Operation Definitions

This subsubsection presents the verification predicates for the two different kind of operation definitions. In addition the corresponding evaluation functions are given.

Verification Predicates for Operation Definitions

35. $VerifyExplOps : \text{Pred}(\text{IE}(ExplOpDef) \times ENV)$
- 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 $st_d = \text{GetStateDom}(env)$ in
- .7 $st_d \in DOM \wedge$
- .8 $rel \subseteq \{(\|fd_1\| \times \|st_d\| \times \|st_d\| \times (\|fd_2\| \cup \{\text{nil}\}) \times MODE)$
- .9 $\quad | fd_1, fd_2 \in DOM\} \wedge$
- .10 $rel = \text{EvalExplOp}(op_m(id))(env)$
- .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 = dom(op_m) in
.3    if id_s  $\subseteq$  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  $\subseteq \{(\|fd_1\| \times \|st_d\| \times \|st_d\| \times (\|fd_2\| \cup \{\underline{nil}\}) \times MODE)$ 
.9         $| fd_1, fd_2 \in DOM\} \wedge$ 
.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

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  $\neg \text{CheckTagEnv}(\text{env}, \text{tp}) \vee \neg \text{CheckTagEnv}(\text{env}, \text{restp})$ 
      .5  then err
      .6  else let i_d = EvalType(tp)(env),
      .7    o_d = if restp = nil
      .8      then ( $\emptyset_{\perp}$ , { })
      .9      else EvalType(restp)(env),
     .10   st_d = GetStateDom(env) in
     .11   if i_d = err  $\vee$  o_d = err
     .12   then err
     .13   else let st_s = if  $\| \text{st\_d} \| = \{ \}$ 
     .14     then { nil }
     .15     else  $\| \text{st\_d} \| \cup \{ \perp \}$ ,
     .16   o_s = if  $\| \text{o\_d} \| = \{ \}$ 
     .17     then { nil }
     .18     else  $\| \text{o\_d} \| \cup \{ \perp \}$  in
     .19   let r = {(i_v, i_st, o_st, o_v, m)
     .20   | i_v ∈  $\| \text{i\_d} \| \cup \{ \perp \}$ , 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  $\wedge$  e2 ≠ err
     .24   CheckStateConstancy(e2)(ext)(i_st, o_st) = True()  $\wedge$ 
     .25   rid = nil  $\Leftrightarrow$  (o_v = nil  $\wedge$  m = cont)  $\wedge$ 
     .26   rid ≠ nil  $\Leftrightarrow$  (o_v ≠ nil  $\wedge$  m = ret)  $\wedge$ 
     .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  $\Rightarrow$  post_v_s = { True() } } in
     .30   if r = { }  $\vee$ 
     .31    $\exists e \in r \cdot \text{let } (., ., o_{st}, o_v, .) = e \text{ in } o_{st} \notin st_s \vee o_v \notin o_s$ 
     .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 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  $\neg \text{CheckTagEnv}(env, tp) \vee \neg \text{CheckTagEnv}(env, restp)$ 
.4        then err
.5      else let i_d = EvalType(tp)(env),
.6          o_d = if restp = nil
.7            then ( $\emptyset_{\perp}$ , { })
.8            else EvalType(restp)(env),
.9          st_d = GetStateDom(env) in
.10         if i_d = err  $\vee$  o_d = err
.11         then err
.12         else let st_s = if  $\|st_d\| = \{\}$ 
.13           then {nil}
.14           else  $\|st_d\| \cup \{\perp\}$ ,
.15         o_s = if  $\|o_d\| = \{\}$ 
.16           then {nil}
.17           else  $\|o_d\| \cup \{\perp\}$  in
.18         let r = {(i_v, i_st, o_st, o_v, m)
.19           | i_v  $\in \|i_d\| \cup \{\perp\}$ , i_st  $\in st_s$ , o_st  $\in VAL$ ,
.20             o_v  $\in VAL$ , m  $\in MODE$ .
.21         let (env', env'') = ExtExplOpEnv(id, i_v, i_st, o_st, ext)(env) in
.22           CheckStateConstancy(env')(ext)(i_st, o_st) = True()  $\wedge$ 
.23           let pre_v_s = {evpre(env') | evpre  $\in EvalExpr(pre)},
.24             lsev = EvalStmt(body) in
.25             True()  $\in pre_v_s \Rightarrow$ 
.26                $\exists sev \in lsev \cdot sev(env') = (env'', m, o_v)$  } in
.27             if r = { }  $\vee$ 
.28                $\exists e \in r \cdot \text{let } (-, -, o_st, o_v, -) = e \text{ in } o_st \notin st_s \vee o_v \notin o_s$ 
.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 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

5.3 Expressions

This section 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' }

39. $\text{EvalExpr} : \text{Expr} \rightarrow \text{LEEval}$
 39.1 $\text{EvalExpr}(\text{expr}) \triangleq$
 .2 cases $\text{ShowTag}(\text{expr}) :$
 .3 ‘ LetExpr ’ $\rightarrow \text{EvalLetExpr}(\text{expr}),$
 .4 ‘ LetBeSE Expr ’ $\rightarrow \text{EvalLetBeSE Expr}(\text{expr}),$
 .5 ‘ DefExpr ’ $\rightarrow \text{EvalDefExpr}(\text{expr}),$
 .6 ‘ IfExpr ’ $\rightarrow \text{EvalIfExpr}(\text{expr}),$
 .7 ‘ CasesExpr ’ $\rightarrow \text{EvalCasesExpr}(\text{expr}),$
 .8 ‘ UnaryExpr ’ $\rightarrow \text{EvalUnaryExpr}(\text{expr}),$
 .9 ‘ BinaryExpr ’ $\rightarrow \text{EvalBinaryExpr}(\text{expr}),$
 .10 ‘ AllOrExistsExpr ’ $\rightarrow \text{EvalAllOrExistsExpr}(\text{expr}),$
 .11 ‘ ExistsUniqueExpr ’ $\rightarrow \text{EvalExistsUniqueExpr}(\text{expr}),$
 .12 ‘ IotaExpr ’ $\rightarrow \text{EvalIotaExpr}(\text{expr}),$
 .13 ‘ SetEnumeration ’ $\rightarrow \text{EvalSetEnumeration}(\text{expr}),$
 .14 ‘ SetComprehension ’ $\rightarrow \text{EvalSetComprehension}(\text{expr}),$
 .15 ‘ SetRange ’ $\rightarrow \text{EvalSetRange}(\text{expr}),$
 .16 ‘ SeqEnumeration ’ $\rightarrow \text{EvalSeqEnumeration}(\text{expr}),$
 .17 ‘ SeqComprehension ’ $\rightarrow \text{EvalSeqComprehension}(\text{expr}),$
 .18 ‘ SubSequence ’ $\rightarrow \text{EvalSubSequence}(\text{expr}),$
 .19 ‘ MapEnumeration ’ $\rightarrow \text{EvalMapEnumeration}(\text{expr}),$
 .20 ‘ MapComprehension ’ $\rightarrow \text{EvalMapComprehension}(\text{expr}),$
 .21 ‘ TupleConstructor ’ $\rightarrow \text{EvalTupleConstructor}(\text{expr}),$
 .22 ‘ RecordConstructor ’ $\rightarrow \text{EvalRecordConstructor}(\text{expr}),$
 .23 ‘ RecordModifier ’ $\rightarrow \text{EvalRecordModifier}(\text{expr}),$
 .24 ‘ Apply ’ $\rightarrow \text{EvalApplyExpr}(\text{expr}),$
 .25 ‘ FieldSelect ’ $\rightarrow \text{EvalFieldSelectExpr}(\text{expr}),$
 .26 ‘ FctTypeInst ’ $\rightarrow \text{EvalFctTypeInstExpr}(\text{expr}),$
 .27 ‘ Lambda ’ $\rightarrow \text{EvalLambda}(\text{expr}),$
 .28 ‘ IsExpr ’ $\rightarrow \text{EvalIsExpr}(\text{expr}),$
 .29 LITERAL $\rightarrow \text{EvalLiteralExpr}(\text{expr}),$
 .30 ID $\rightarrow \text{EvalId}(\text{expr}),$
 .31 ‘ OldId ’ $\rightarrow \text{EvalOldId}(\text{expr})$

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

5.3.1 Local Binding Expressions

```

40.   EvalLetExpr : LetExpr → LEEval
40.1  EvalLetExpr(MkTag('LetExpr', (vals, explfns, implfns, in)))  $\triangleq$ 
    .2  let ids =  $\bigcup\{\text{ColIdSet}(p) \mid p \in \underline{\text{dom}}(\text{vals})\}$  in
    .3  let lev = {  $\lambda \text{env}.$ let fnenv = restrict(env', dom(explfns)  $\cup$  dom(implfns)  $\cup$  ids) in
    .4    let env'' = overwrite(env, fnenv) in
    .5      if def(env'') = err  $\vee$  BotEnv(def(env''))
    .6        then  $\perp$ 
    .7        else ev(def(env''))
    .8      | def  $\in$  EvalLocalDef(vals, explfns, implfns), ev  $\in$  EvalExpr(in), env'  $\in$  ENV ·
    .9        VerifyExplFns(explfns, env')  $\wedge$  VerifyImplFns(implfns, env')  $\wedge$ 
    .10       VerifyValues(vals, env') } in
    .11     if lev = { }
    .12     then {  $\lambda \text{env}.$   $\perp$  }
    .13     else lev

```

Annotations to *EvalLetExpr*:

- 40. This function gives meaning to *Let* expressions³.
- .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

³Notice 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.

```

41.   EvalLocalDef : IM(Pattern, ValDef) × IE(ExplFnDef) × IE(ImplFnDef) → LDef
41.1  EvalLocalDef(vdm, efdm, ifdm) ≜
    .2  let ids = ⋃{ ColIdSet(p) | p ∈ dom(vdm) } in
    .3  if ids ∩ dom(efdm) ≠ { } ∨
    .4  ids ∩ dom(ifdm) ≠ { } ∨
    .5  dom(efdm) ∩ dom(ifdm) ≠ { }
    .6  then { λenv.err }
    .7  else let fd_m = { MkTag('PatternId', id) ↪ efdm(id)
    .8  | id ∈ dom(efdm) } in
    .9  let mul_ldef =
        { λenv.let funm_s = MakeFns(ifdm)(env) in
        .11  if funm_s = err
        .12  then { }
        .13  else { env'
        .14  | env' ∈ ENV ·
        .15  ∃ funm ∈ funm_s.
        .16  env' = tf(overwrite(env, funm)) ∧
        .17  VerifyImplFns(ifdm)(env') }
        .18  | tf ∈ EvalExplDef(merge(vdm, fdm)) } 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 venvs = ⋃{ bev(env) | bev ∈ EvalBind(bind) } in
    .3  let venvs' = { venv | venv ∈ venvs · venv ≠ err ∧ ¬BotEnv(venv) ∧
    .4  stev(overwrite(env, venv)) = True() } in
    .5  { inev(overwrite(env, venv)) | venv ∈ venvs' }
    .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 envi-

ronment extended with all possible value environments.

- .6 This is done for all possible ‘such that’ predicate evaluators and all possible expression evaluators for the ‘in-expression’.

End of annotations

43. $\text{EvalDefExpr} : \text{DefExpr} \rightarrow \text{LEEVal}$

43.1 $\text{EvalDefExpr}(\text{MkTag}(\text{'DefExpr}', (\text{lhs}, \text{rhs}, \text{in}))) \triangleq$
.2 $\{ \lambda \text{env}. \text{let } \text{venv} = \text{pev}(\text{ev}(\text{env}))(\text{env}) \text{ in }$
.3 $\quad \text{if } \text{venv} \in \{\text{err}, \text{unmatch}\} \vee \text{BotEnv}(\text{venv})$
.4 $\quad \text{then } \perp$
.5 $\quad \text{else } \text{inev}(\text{overwrite}(\text{env}, \text{venv}))$
.6 $\mid \text{pev} \in \text{EvalPattern}(\text{lhs}), \text{ev} \in \text{EvalExpr}(\text{rhs}), \text{inev} \in \text{EvalExpr}(\text{in}) \}$

Annotations to EvalDefExpr :

43. This function gives meaning to definition expressions⁴.
- .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

5.3.2 Conditional Expressions

44. $\text{EvalIfExpr} : \text{IfExpr} \rightarrow \text{LEEVal}$

44.1 $\text{EvalIfExpr}(\text{MkTag}(\text{'IfExpr}', (\text{test}, \text{cons}, \text{altn}))) \triangleq$
.2 $\{ \lambda \text{env}. \text{if } \text{testev}(\text{env}) \notin \text{BOOL_VAL}$
.3 $\quad \text{then } \perp$
.4 $\quad \text{else if } \text{testev}(\text{env}) = \text{True}()$
.5 $\quad \text{then } \text{consev}(\text{env})$
.6 $\quad \text{else } \text{altnev}(\text{env})$
.7 $\mid \text{testev} \in \text{EvalExpr}(\text{test}), \text{consev} \in \text{EvalExpr}(\text{cons}), \text{altnev} \in \text{EvalExpr}(\text{altn}) \}$

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 $\text{True}()$ 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

⁴Notice 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.

45. $\text{EvalCasesExpr} : \text{CasesExpr} \rightarrow \text{LEEeval}$
 45.1 $\text{EvalCasesExpr}(\text{MkTag}(\text{'CasesExpr'}, (\text{sel}, \text{altns}))) \triangleq$
 .2 $\text{PropE}(\{\lambda \text{env}. \text{if } \text{selev}(\text{env}) = \perp$
 .3 $\quad \text{then } \{\perp\}$
 .4 $\quad \text{else } \{ \text{ev}(\text{env}) \mid \text{ev} \in \text{ApplyCasesExpr}(\text{selev}(\text{env}), \text{altns}) \}$
 .5 $\quad \mid \text{selev} \in \text{EvalExpr}(\text{sel}) \})$

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. $\text{ApplyCasesExpr} : \text{VAL} \times \text{IL}(\text{CaseExprAltn}) \rightarrow \text{LEEeval}$
 46.1 $\text{ApplyCasesExpr}(\text{val}, \text{csa.l}) \triangleq$
 .2 $\text{if } \underline{\text{len}}(\text{csa.l}) = 0$
 .3 $\text{then } \{\lambda \text{env}. \perp\}$
 .4 $\text{else } \{ \lambda \text{env}. \text{let } \text{res} = \text{caev}(\text{env}) \text{ in}$
 .5 $\quad \text{if } \text{res} = \underline{\text{unmatch}}$
 .6 $\quad \text{then } \text{c}(\text{env})$
 .7 $\quad \text{else } \text{res}$
 .8 $\quad \mid \text{caev} \in \text{ApplyCaseExprAltn}(\text{val}, \underline{\text{hd}}(\text{csa.l})),$
 .9 $\quad \text{c} \in \text{ApplyCasesExpr}(\text{val}, \underline{\text{tl}}(\text{csa.l})) \}$

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. $\text{ApplyCaseExprAltn} : \text{VAL} \times \text{CaseExprAltn} \rightarrow \text{IP}(\text{ENV} \rightarrow (\text{VAL} \cup \{\underline{\text{unmatch}}\}))$
 47.1 $\text{ApplyCaseExprAltn}(\text{val}, \text{MkTag}(\text{'CaseAltn'}, (\text{match}, \text{body}))) \triangleq$
 .2 $\text{PropE}(\{\lambda \text{env}. \text{let } s = \{ \text{boev}(\text{env})$
 .3 $\quad \mid \text{boev} \in \text{EvalExpr}(\text{body}),$
 .4 $\quad \text{env}' \in \text{NonErrPat}(\text{match}, \text{val}, \text{env}) \} \text{ in}$
 .5 $\quad \text{if } s = \{ \}$
 .6 $\quad \text{then } \{\underline{\text{unmatch}}\}$
 .7 $\quad \text{else } s$
 .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.

End of annotations

5.3.3 Unary Expressions

48. *EvalUnaryExpr* : *UnaryExpr* → *LEEval*
48.1 *EvalUnaryExpr*(*MkTag*('UnaryExpr', (*opr*, *arg*))) ≡
.2 { λ*env*.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 → *SqXx*(*opr*, *val*),
.9 SEQDISTRCONC → *SqDi*(*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

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* ∈ \mathbb{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

Logical Operation

```

50.   Not : VAL → VAL
50.1  Not(val) ≡
    .2  if val ∈ BOOL_VAL
    .3  then if val = True() then False()
    .4  else True()
    .5  else ⊥
    .6

```

Annotations to *Not*:

50. This function perform logical negation of the argument value (if it is a Boolean value).
End of annotations

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',  $\bigcup(ss)$ )
    .8  SETDISTRINTERSECT → if ss = {} then ⊥
    .9
    .10  else MkTag('set',  $\bigcap(ss)$ )
    .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.

End of annotations

Sequence Operations

53. $\text{SeqXx} : \{\text{SEQHEAD}, \text{SEQTAIL}, \text{SEQLEN}, \text{SEQELEMS}, \text{SEQINDICES}\} \times \text{VAL} \rightarrow \text{VAL}$

53.1 $\text{SeqXx}(\text{un_op}, \text{val}) \triangleq$

.2 if $\text{val} \in \text{LIST_VAL}$

.3 then let $\text{seq_val} = \text{StripSeqTagVal}(\text{val})$ in

.4 cases un_op :

.5 SEQHEAD \rightarrow if $\text{seq_val} = []$
 then \perp
 else hd(seq_val),

.6 SEQTAIL \rightarrow if $\text{seq_val} = []$
 then \perp
 else $\text{MkTag}(\text{'seq'}, \underline{\text{tl}}(\text{seq_val}))$,

.7 SEQLEN \rightarrow $\text{MkTag}(\text{'num'}, \underline{\text{len}}(\text{seq_val}))$,

.8 SEQELEMS \rightarrow $\text{MkTag}(\text{'set'}, \underline{\text{elems}}(\text{seq_val}))$,

.9 SEQINDICES \rightarrow $\text{MkTag}(\text{'set'}, \underline{\text{inds}}(\text{seq_val}))$

.14 else \perp

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. $\text{SeqDi} : \text{VAL} \rightarrow \text{VAL}$

54.1 $\text{SeqDi}(\text{val}) \triangleq$

.2 if $\text{val} \in \text{LIST_VAL}$

.3 then let $\text{seq_seq_val} = \text{StripSeqTagVal}(\text{val})$ in

.4 if $\underline{\text{elems}}(\text{seq_seq_val}) \subseteq \text{LIST_VAL}$

.5 then let $\text{ll} = \text{ApplySeq}(\text{StripSeqTagVal})(\text{seq_seq_val})$ in

.6 $\text{MkTag}(\text{'seq'}, \underline{\text{Join}}(\text{ll}))$

.7 else \perp

.8 else \perp

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

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
            cases un_op :
      .5  MAPDOM → MkTag('set', dom(map_val)),
      .6  MAPRNG → if  $\forall a \in \underline{\text{dom}}(\text{map\_val}) \cdot \text{map\_val}(a) \in \text{FLAT\_VAL}$ 
                  then MkTag('set', rng(map_val))
                  else ⊥,
      .7  MAPINVERSE → if Injective(m) ∧
                       $\forall a \in \underline{\text{dom}}(m) \cdot m(a) \in \text{FLAT\_VAL}$ 
                      then MkTag('map', { m(d) ↦ d | d ∈ \underline{\text{dom}}(m) })
                      else ⊥
      .8
      .9
      .10
      .11
      .12
      .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
            if  $\forall m \in \text{set\_map\_val} \cdot m \in \text{MAP\_VAL}$ 
            then let mm = { StripMapTagVal(m) | m ∈ set_map_val } in
                  if Compatible(mm)
                  then MkTag('map', Merge(mm))
                  else ⊥
            else ⊥
      .9
      .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

5.3.4 Binary Expressions

```

57.   EvalBinaryExpr : BinaryExpr → LEEval
57.1  EvalBinaryExpr(MkTag('BinaryExpr', (left, opr, right))) ≡
.2    { λenv.let Lv = Lev(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   | Lev ∈ 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

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.

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*:

$l_v \text{ div } r_v$ $l_v \text{ rem } r_v$ $l_v \text{ mod } r_v$

Before presenting the semantics of these operators an example (taken from [Daw91]) is shown to illustrating the signs used:

l_v	r_v	$l_v \text{ rem } r_v$	$l_v \text{ div } r_v$	$l_v \text{ mod } r_v$	$\text{floor}(l_v/r_v)$
+ 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. $\text{IntDiv} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

59.1 $\text{IntDiv}(l_val, r_val) \triangleq$
 .2 if ($l_val \in \text{NUM_VAL}$) \wedge ($r_val \in \text{NUM_VAL}$)
 .3 then let $l_v = \text{StripNumTagVal}(l_val)$,
 $r_v = \text{StripNumTagVal}(r_val)$ 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 $q = \text{Iota}(\{p \in \mathbb{Z} \mid \exists r \in \mathbb{Z} \cdot \underline{\text{abs}}(r) < \underline{\text{abs}}(r_v) \wedge$
 .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

60. $\text{NumRem} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

60.1 $\text{NumRem}(l_val, r_val) \triangleq$

.2 if ($l_val \in \text{NUM_VAL}$) \wedge ($r_val \in \text{NUM_VAL}$)

.3 then let $l_v = \text{StripNumTagVal}(l_val)$,

.4 $r_v = \text{StripNumTagVal}(r_val)$ 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 $r = \text{Iota}(\{ r \in \mathbb{Z} \mid \underline{\text{abs}}(r) < \underline{\text{abs}}(r_v) \wedge r \times l_v \geq 0 \wedge \exists p \in \mathbb{Z} \cdot l_v = r_v \times p + r \})$ in

.9 $MkTag('num', r)$

.10 else \perp

.11 else \perp

.12 else \perp

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. $\text{NumMod} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

61.1 $\text{NumMod}(l_val, r_val) \triangleq$

.2 if ($l_val \in \text{NUM_VAL}$) \wedge ($r_val \in \text{NUM_VAL}$)

.3 then let $l_v = \text{StripNumTagVal}(l_val)$,

.4 $r_v = \text{StripNumTagVal}(r_val)$ 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 $\text{floor} = \text{Iota}(\{ t \in \mathbb{Z} \mid t \leq l_v / r_v \wedge t + 1 > l_v / r_v \})$ in

.9 $MkTag('num', l_v - r_v \times \text{floor})$

.10 else \perp

.11 else \perp

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

Numeric Comparison Operators

62. $\text{NumComp} : \{\text{NUMLT}, \text{NUMLE}, \text{NUMGT}, \text{NUMGE}\} \times \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

62.1 $\text{NumComp}(\text{bin_op}, l\text{-val}, r\text{-val}) \triangleq$

.2 if ($l\text{-val} \in \text{NUM_VAL}$) \wedge ($r\text{-val} \in \text{NUM_VAL}$)

.3 then let $l\text{-v} = \text{StripNumTagVal}(l\text{-val})$,

.4 $r\text{-v} = \text{StripNumTagVal}(r\text{-val})$ in

.5 cases bin_op :

.6 $\text{NUMLT} \rightarrow$ if $l\text{-v} < r\text{-v}$
 .7 then $\text{True}()$
 .8 else $\text{False}()$,

.9 $\text{NUMLE} \rightarrow$ if $l\text{-v} \leq r\text{-v}$
 .10 then $\text{True}()$
 .11 else $\text{False}()$,

.12 $\text{NUMGT} \rightarrow$ if $l\text{-v} > r\text{-v}$
 .13 then $\text{True}()$
 .14 else $\text{False}()$,

.15 $\text{NUMGE} \rightarrow$ if $l\text{-v} \geq r\text{-v}$
 .16 then $\text{True}()$
 .17 else $\text{False}()$

.18 else \perp

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

Equality Operations

63. $\text{EqualOpr} : \{\text{EQ}, \text{NE}\} \times \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

63.1 $\text{EqualOpr}(\text{opr}, l\text{-v}, r\text{-v}) \triangleq$

.2 if ($l\text{-v} = \perp \vee r\text{-v} = \perp \vee (l\text{-v} \notin \text{FLAT_VAL} \vee r\text{-v} \notin \text{FLAT_VAL})$)

.3 then \perp

.4 else cases opr :

.5 $\text{EQ} \rightarrow$ if $l\text{-v} = r\text{-v}$
 .6 then $\text{True}()$
 .7 else $\text{False}()$

.8 $\text{NE} \rightarrow$ if $l\text{-v} = r\text{-v}$
 .9 then $\text{False}()$
 .10 else $\text{True}()$

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

Logical Operations

```

64.   LogOpr : { OR,AND,IMPLY,EQUIV } × VAL × VAL → VAL
64.1  LogOpr(opr, l_v, r_v) ≡
.2    cases opr :
.3      OR   → if l_v ∈ BOOL_VAL⊥ ∧ r_v ∈ BOOL_VAL⊥
.4          then if (l_v = True()) ∨ (r_v = True())
.5              then True()
.6              else if (l_v = False()) ∧ (r_v = False())
.7                  then False()
.8                  else ⊥
.9                  else ⊥,
.10     AND  → if l_v ∈ BOOL_VAL⊥ ∧ r_v ∈ BOOL_VAL⊥
.11        then if (l_v = False()) ∨ (r_v = False())
.12            then False()
.13            else if (l_v = True()) ∧ (r_v = True())
.14                then True()
.15                else ⊥
.16                else ⊥,
.17     IMPLY → if l_v ∈ BOOL_VAL⊥ ∧ r_v ∈ BOOL_VAL⊥
.18        then if (l_v = False()) ∨ (r_v = True())
.19            then True()
.20            else if (l_v = True()) ∧ (r_v = False())
.21                then False()
.22                else ⊥
.23                else ⊥,
.24     EQUIV → if (l_v = True() ∧ r_v = True()) ∨
.25         (l_v = False() ∧ r_v = False())
.26         then True()
.27         else if (l_v = True() ∧ r_v = False()) ∨
.28             (l_v = False() ∧ r_v = True())
.29             then False()
.30             else ⊥

```

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

Logical Set Operations

65. $\text{CheckMembership} : \{\text{INSET}, \text{NOTINSET}\} \times \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

65.1 $\text{CheckMembership}(\text{mem}, \text{val}, \text{set}) \triangleq$
 .2 if $\text{set} \in \text{SET_VAL} \wedge \text{val} \neq \perp$
 .3 then let $s = \text{StripSetTagVal}(\text{set})$ in
 .4 cases mem :
 .5 INSET \rightarrow if $\text{val} \in s$
 .6 then $\text{True}()$
 .7 else $\text{False}()$,
 .8 NOTINSET \rightarrow if $\text{val} \in s$
 .9 then $\text{False}()$
 .10 else $\text{True}()$
 .11 else \perp

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. $\text{SubSet} : \{\text{SUBSET}, \text{PROPERSUBSET}\} \times \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

66.1 $\text{SubSet}(\text{bin_op}, \text{l_val}, \text{r_val}) \triangleq$
 .2 if $(\text{l_val} \in \text{SET_VAL}) \wedge (\text{r_val} \in \text{SET_VAL})$
 .3 then let $\text{l_v} = \text{StripSetTagVal}(\text{l_val})$,
 .4 $\text{r_v} = \text{StripSetTagVal}(\text{r_val})$ in
 .5 cases bin_op :
 .6 SUBSET \rightarrow if $\text{l_v} \subseteq \text{r_v}$
 .7 then $\text{True}()$
 .8 else $\text{False}()$,
 .9 PROPERSUBSET \rightarrow if $\text{l_v} \subset \text{r_v}$
 .10 then $\text{True}()$
 .11 else $\text{False}()$
 .12 else \perp

Annotations to *SubSet*:

66. This function deals with the subset and proper subset operations.
End of annotations

Set Operations

67. $\text{SetBin} : \{\text{SETUNION}, \text{SETDIFFERENCE}, \text{SETINTERSECT}\} \times \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

67.1 $\text{SetBin}(\text{bin_op}, \text{l_val}, \text{r_val}) \triangleq$
 .2 if $(\text{l_val} \in \text{SET_VAL}) \wedge (\text{r_val} \in \text{SET_VAL})$
 .3 then let $\text{l_v} = \text{StripSetTagVal}(\text{l_val})$,
 .4 $\text{r_v} = \text{StripSetTagVal}(\text{r_val})$ in
 .5 cases bin_op :
 .6 SETUNION $\rightarrow \text{MkTag}(\text{'set'}, \text{l_v} \cup \text{r_v})$,
 .7 SETDIFFERENCE $\rightarrow \text{MkTag}(\text{'set'}, \text{l_v} \setminus \text{r_v})$,
 .8 SETINTERSECT $\rightarrow \text{MkTag}(\text{'set'}, \text{l_v} \cap \text{r_v})$
 .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**

Sequence Operation

68. $\text{SeqConc} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$
- 68.1 $\text{SeqConc}(l_val, r_val) \triangleq$
- .2 if ($l_val \in \text{LIST_VAL}$) \wedge ($r_val \in \text{LIST_VAL}$)
- .3 then let $l_v = \text{StripSeqTagVal}(l_val)$,
- .4 $r_v = \text{StripSeqTagVal}(r_val)$ in
- .5 $\text{MkTag}(\text{'seq'}, \underline{\text{join}}(l_v, r_v))$
- .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**

Sequence and Map Modification Operation

69. $\text{MapOrSeqMod} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$
- 69.1 $\text{MapOrSeqMod}(l_val, r_val) \triangleq$
- .2 if ($l_val \in \text{LIST_VAL}$) \wedge ($r_val \in \text{MAP_VAL}$)
- .3 then let $l_v = \text{StripSeqTagVal}(l_val)$,
- .4 $r_v = \text{StripMapTagVal}(r_val)$ in
- .5 if $\underline{\text{dom}}(r_v) \subseteq \underline{\text{inds}}(l_v)$
- .6 then let $den = [\text{if } i \in \underline{\text{dom}}(l_v)$
- .7 then $r_v(i)$
- .8 else $l_v(i)$
- .9 | $i \in \underline{\text{inds}}(l_v)$] in
- .10 if $\perp \in \underline{\text{elems}}(den)$
- .11 then \perp
- .12 else $\text{MkTag}(\text{'seq'}, den)$
- .13 else \perp
- .14 else if ($l_val \in \text{MAP_VAL}$) \wedge ($r_val \in \text{MAP_VAL}$)
- .15 then let $l_v = \text{StripMapTagVal}(l_val)$,
- .16 $r_v = \text{StripMapTagVal}(r_val)$ in
- .17 $\text{MkTag}(\text{'map'}, \underline{\text{overwrite}}(l_v, r_v))$
- .18 else \perp

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

Map Operations

70. $\text{MapMerge} : \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

70.1 $\text{MapMerge}(l_val, r_val) \triangleq$

.2 if ($l_val \in \text{MAP_VAL}$) \wedge ($r_val \in \text{MAP_VAL}$)

.3 then let $l_v = \text{StripMapTagVal}(l_val)$,

.4 $r_v = \text{StripMapTagVal}(r_val)$ in

.5 if $\text{Compatible}(\{ l_v, r_v \})$

.6 then $\text{MkTag}(\text{'map'}, \underline{\text{merge}}(l_v, r_v))$

.7 else \perp

.8 else \perp

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

71. $\text{MapToSetOpr} : \{ \text{MAPDOMRESTRTO}, \text{MAPDOMRESTRBY}, \text{MAPRNGRESTRTO}, \text{MAPRNGRESTRBY} \}$
 $\times \text{VAL} \times \text{VAL} \rightarrow \text{VAL}$

71.1 $\text{MapToSetOpr}(bin_op, l_val, r_val) \triangleq$

.2 if ($l_val \in \text{MAP_VAL}$) \wedge ($r_val \in \text{SET_VAL}$)

.3 then let $l_v = \text{StripMapTagVal}(l_val)$,

.4 $r_v = \text{StripSetTagVal}(r_val)$ in

.5 cases bin_op :

.6 $\text{MAPDOMRESTRTO} \rightarrow \text{MkTag}(\text{'map'}, \underline{\text{restrict}}(l_v, r_v)),$

.7 $\text{MAPDOMRESTRBY} \rightarrow \text{MkTag}(\text{'map'}, \underline{\text{subtract}}(l_v, r_v)),$

.8 $\text{MAPRNGRESTRTO} \rightarrow \text{MkTag}(\text{'map'}, \{ a \mapsto l_v(a) \mid a \in \underline{\text{dom}}(l_v) \cdot l_v(a) \in r_v \}),$

.9 $\text{MAPRNGRESTRBY} \rightarrow \text{MkTag}(\text{'map'}, \{ a \mapsto l_v(a) \mid a \in \underline{\text{dom}}(l_v) \cdot l_v(a) \notin r_v \})$

.10

.11

.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

Compose and Iterate

```

72.   Compose : VAL × VAL → VAL
72.1  Compose(l-v, r-v) ≡
    .2  if l-v ∈ FUN_VAL ∧ r-v ∈ FUN_VAL
    .3  then let MkTag('fun', f) = l-v,
        .4      MkTag('fun', g) = r-v in
        .5      if δ1(g) ⊆ δ0(f)
        .6      then MkTag('fun', f ∘ g)
        .7      else ⊥
    .8  else if l-v ∈ MAP_VAL ∧ r-v ∈ MAP_VAL
    .9  then let m1 = StripMapTagVal(l-v),
    .10     m2 = StripMapTagVal(r-v) in
    .11     if rng(m2) ⊆ dom(m1)
    .12     then MkTag('map', {i ↦ m1(m2(i)) | i ∈ dom(m2)})
    .13     else ⊥
    .14  else ⊥

```

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:
 $f_1 \circ f_2 = f_1(f_2(x))$
 $f_1 \in A \rightarrow B \wedge f_2 \in C \rightarrow D$
- .5 Recall that δ₀ is the domain operator on general functions, and that δ₁ is the range operator on general functions (as described page 10).
- .5 Condition: $D \subseteq A$
.6 Result: $f_1 \circ f_2 \in C \rightarrow B$
- .9-.13 Map composition can be explained by:
 $m_1 \circ m_2 = m_1(m_2(x))$
 $m_1 \in IM(A, B) \wedge m_2 \in IM(C, D)$
- .11 Condition: $D \subseteq A$
.12 Result: $m_1 \circ m_2 \in IM(C, B)$

End of annotations

```

73.   Iterate : VAL × VAL → VAL
73.1  Iterate(l_v, r_v) ≡
.2    if l_v ∈ FUN_VAL ∧ r_v ∈ NUM_VAL
.3    then let MkTag('fun', f) = l_v,
.4        MkTag('num', n) = r_v in
.5        if n ∈ IN ∧ (n ≥ 2 ⇒ δ1(f) ⊆ δ0(f))
.6        then MkTag('fun', IterateFct(f, n))
.7        else ⊥
.8    else if l_v ∈ MAP_VAL ∧ r_v ∈ NUM_VAL
.9    then let m = StripMapTagVal(l_v),
.10       n = StripNumTagVal(r_v) in
.11       if n ∈ IN ∧ (n ≥ 2 ⇒ rng(m) ⊆ dom(m))
.12       then MkTag('map', IterateMap(m, n))
.13       else ⊥
.14    else if l_v ∈ NUM_VAL ∧ r_v ∈ NUM_VAL
.15    then let l_v = StripNumTagVal(l_val),
.16       r_v = StripNumTagVal(r_val) in
.17       if (l_v > 0) ∨
.18           (l_v = 0 ∧ r_v > 0) ∨
.19           (l_v < 0 ∧ OddDenom(r_v))
.20       then MkTag('num', l_vr_v)
.21       else ⊥
.22    else ⊥

```

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:

$$f^n(x) = \underbrace{f(f(f(f \dots f(x) \dots)))}_{n \text{ times}}$$

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 10).

.5 Condition: $B \subseteq A$

.6 Result: $f^n \in A \rightarrow B$

- .9 – .13 Map iteration can be explained by:

$$m^n(x) = \underbrace{m(m(m(m \dots m(x) \dots)))}_{n \text{ times}}$$

where $m \in IM(A, B)$

- .11 Condition: $n \in IN \wedge (n \geq 2 \Rightarrow B \subseteq A)$

.12 Result: $m^n \in IM(A, B)$

- .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

74. $\text{IterateFct} : (A \rightarrow B) \times \mathbb{N} \rightarrow (A \rightarrow B) \cup \{\perp\}$

74.1 $\text{IterateFct}(f, n) \triangleq$
.2 if $n \geq 1 \wedge (n \geq 2 \Rightarrow \delta_1(f) \subseteq \delta_0(f))$
.3 then $f \circ (\text{IterateFct}(f, n - 1))$
.4 else if $n = 0$
.5 then $\lambda x \in \delta_0(f) . x$
.6 else \perp

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 $f^0 = \lambda x \in \delta_0(f) . x$, 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. $\text{IterateMap} : \text{IM}(A, B) \times \mathbb{N} \rightarrow \text{IM}(A, B) \cup \{\perp\}$

75.1 $\text{IterateMap}(\text{map}, n) \triangleq$
.2 if $n \geq 1 \wedge (n \geq 2 \Rightarrow \text{rng}(\text{map}) \subseteq \text{dom}(\text{map}))$
.3 then $\{ i \mapsto \text{map}(\text{IterateMap}(\text{map}, n - 1)(i)) \mid i \in \text{dom}(\text{map}) \}$
.4 else if $n = 0$
.5 then $\{ a \mapsto a \mid a \in \text{dom}(\text{map}) \}$
.6 else \perp

Annotations to *IterateMap*:

75. This auxiliary function performs iteration of the given map, map , the given number of times, n .

.3 $\text{map}^n = \text{map}(\text{map}^{n-1})$
.4 $\text{map}^0 = \{ a \mapsto a \mid a \in \text{dom}(\text{map}) \}$ i.e. the identity map on the domain of map .

End of annotations

76. $\text{OddDenom} : \text{Pred}(\mathbb{R})$

76.1 $\text{OddDenom}(r) \triangleq$
.2 $\exists m, n \in \mathbb{N} \cdot GCD(m, n) = 1 \wedge \text{IsOdd}(n) \wedge$
.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. $GCD : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

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

78. $\text{IsOdd} : \text{Pred}(\mathbb{N})$
 78.1 $\text{IsOdd}(n) \triangleq$
 .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

5.3.5 Quantified Expressions

79. $\text{EvalAllOrExistsExpr} : \text{AllOrExistsExpr} \rightarrow \text{LEEval}$
 79.1 $\text{EvalAllOrExistsExpr}(\text{MkTag}(\text{'AllOrExistsExpr'}, (\text{quant}, \text{bind}, \text{pred}))) \triangleq$
 .2 **cases quant :**
 .3 $\text{ALL} \rightarrow \text{EvalForAll}(\text{bind}, \text{pred}),$
 .4 $\text{EXISTS} \rightarrow \text{EvalExists}(\text{bind}, \text{pred})$

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. $\text{EvalForAll} : \text{IF}(\text{Bind}) \times \text{Expr} \rightarrow \text{LEEval}$
 80.1 $\text{EvalForAll}(\text{bind_s}, \text{pred}) \triangleq$
 .2 $\text{let } pbev_s = \text{Partition}(\{ \text{EvalBind}(b) \mid b \in \text{bind_s} \}) \text{ in}$
 .3 $\{ \lambda \text{env}. \text{let } venv_ss = \{ \text{bev}(\text{env}) \mid \text{bev} \in pbev \} \text{ in}$
 .4 $\text{if } \{ \} \in venv_ss \vee \exists \text{venv_s} \in venv_ss \cdot \underline{\text{err}} \in \text{venv_s}$
 .5 $\text{then if } \{ \} \in venv_ss$
 .6 $\text{then True}()$
 .7 $\text{else } \perp$
 .8 $\text{else let } venvs = \{ \text{Merge}(\text{envs}) \mid \text{envs} \in \text{Partition}(\text{venv_ss}) \cdot$
 .9 $\text{Compatible}(\text{envs}) \} \text{ in}$
 .10 $\text{if } \exists \text{venv} \in venvs \cdot \text{ev}(\underline{\text{overwrite}}(\text{env}, \text{venv})) \notin \text{BOOL_VAL}_\perp$
 .11 $\text{then } \perp$
 .12 $\text{else if } \forall \text{venv} \in venvs \cdot \text{ev}(\underline{\text{overwrite}}(\text{env}, \text{venv})) = \text{True}()$
 .13 $\text{then True}()$
 .14 $\text{else if } \exists \text{venv} \in venvs \cdot \text{ev}(\underline{\text{overwrite}}(\text{env}, \text{venv})) = \text{False}()$
 .15 $\text{then False}()$
 .16 $\text{else } \perp$
 .17 $| \text{ev} \in \text{EvalExpr}(\text{pred}), \text{pbev} \in pbev_s \}$

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*.

- .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 :  $\text{IF}(\text{Bind}) \times \text{Expr} \rightarrow \text{LEEval}$ 
81.1 EvalExists(bind_s, pred)  $\triangleq$ 
     2 let pbev_s = Partition( $\{ \text{EvalBind}(b) \mid b \in bind_s \}$ ) in
     3 {  $\lambda env. \text{let } venv\_ss = \{ \text{bev}(env) \mid \text{bev} \in pbev \}$  in
     4   if  $\{ \} \in venv\_ss \vee \exists venv_s \in venv\_ss \cdot \text{err} \in venv_s$ 
     5   then if  $\{ \} \in venv\_ss$ 
     6     then False()
     7     else  $\perp$ 
     8   else let vonus = { Merge(envs)  $| \text{envs} \in \text{Partition}(venv\_ss) \cdot$ 
     9            $\text{Compatible}(\text{envs}) \}$  in
    10     if  $\exists venv \in vonus \cdot ev(\underline{\text{overwrite}}(env, venv)) \notin \text{BOOL\_VAL}_{\perp}$ 
    11     then  $\perp$ 
    12     else if  $\exists venv \in vonus \cdot ev(\underline{\text{overwrite}}(env, venv)) = \text{True}()$ 
    13     then True()
    14     else if  $\forall venv \in vonus \cdot ev(\underline{\text{overwrite}}(env, venv)) = \text{False}()$ 
    15     then False()
    16     else  $\perp$ 
    17   |  $ev \in EvalExpr(pred), pbev \in pbev_s \}$ 

```

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 *venu_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

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:

$$\text{simple} = \exists! a \in \{1, 2, 3\} \cdot a = 2$$

The set expression $\{1, 2, 3\}$ simply has one expression evaluator:

$$\begin{aligned} \text{EvalExpr}(\{1, 2, 3\}) &= \\ \{\lambda \text{env} . \text{MkTag}(\text{'set'}, \{ \text{MkTag}(\text{'num'}, 1), \text{MkTag}(\text{'num'}, 2), \text{MkTag}(\text{'num'}, 3) \})\} \end{aligned}$$

The bindings for which the predicate ' $a = 2$ ' must be checked are:

$$\begin{aligned} \text{EvalBind}(a \in \{1, 2, 3\}) &= \\ \{\lambda \text{env} . \{ \{ a \mapsto \text{MkTag}(\text{'num'}, 1) \}, \\ \{ a \mapsto \text{MkTag}(\text{'num'}, 2) \}, \\ \{ a \mapsto \text{MkTag}(\text{'num'}, 3) \} \} \end{aligned}$$

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:

$$\text{EvalExpr}(\text{simple}) = \{\lambda \text{env} . \text{MkTag}(\text{'bool'}, \text{T})\}$$

The concrete syntax for the second (and much more complicated) example looks like:

$$\begin{aligned} \text{unique} = \exists! l_1 \curvearrowright l_2 \in (\text{let } s \in \{\{[4], [2, 5], 7\}, \{[2], [3], [7, 4]\}\} \text{ in } s) \cdot \\ \text{let } x \in \{2, 4\} \text{ in } x \in (\text{elems } l_1 \cup \text{elems } l_2) \end{aligned}$$

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 $\text{EvalExpr}(\text{unique}) = \{\lambda \text{env} . \text{True}()\}$. 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:

$$\begin{aligned} \text{EvalExpr}(\text{set}) &= \\ \{\lambda \text{env} . \{ [4], [2, 5], 7 \}, \\ \lambda \text{env} . \{ [2], [3], [7, 4] \} \} \end{aligned}$$

In principle all the set values, the sequence values, and the numeric values should be tagged by 'set', 'seq', and 'num' as is done in the domain universe, but for simplicity all these tags are omitted here⁵ (this will also be done in the following formulae).

⁵As an example of how much more complicated the expressions would look the set from the first expression evaluator given above can be described as:

$$\begin{aligned} \text{MkTag}(\text{'set'}, \{ \text{MkTag}(\text{'seq'}, [\text{MkTag}(\text{'num'}, 4)]), \\ \text{MkTag}(\text{'seq'}, [\text{MkTag}(\text{'num'}, 2), \text{MkTag}(\text{'num'}, 5)]), \\ \text{MkTag}(\text{'num'}, 7) \}) \end{aligned}$$

These two set evaluators can be combined with the sequence concatenation pattern $l_1 \curvearrowright l_2$ in 6 and 9 ways respectively. The semantics of the binding will thus be (see *EvalBind* for an explanation of binding denotations):

```
EvalBind(bind) =
{λenv . { { l1 ↦ [ ], l2 ↦ [4] }, { l1 ↦ [ ], l2 ↦ [2, 5] } },
 λenv . { { l1 ↦ [ ], l2 ↦ [4] }, { l1 ↦ [2], l2 ↦ [5] } },
 λenv . { { l1 ↦ [ ], l2 ↦ [4] }, { l1 ↦ [2, 5], l2 ↦ [ ] } },
 λenv . { { l1 ↦ [4], l2 ↦ [ ] }, { l1 ↦ [ ], l2 ↦ [2, 5] } },
 λenv . { { l1 ↦ [4], l2 ↦ [ ] }, { l1 ↦ [2], l2 ↦ [5] } },
 λenv . { { l1 ↦ [4], l2 ↦ [ ] }, { l1 ↦ [2, 5], l2 ↦ [ ] } },
 λenv . { { l1 ↦ [4], l2 ↦ [2] }, { l1 ↦ [ ], l2 ↦ [3] }, { l1 ↦ [ ], l2 ↦ [7, 4] } },
 λenv . { { l1 ↦ [4], l2 ↦ [2] }, { l1 ↦ [ ], l2 ↦ [3] }, { l1 ↦ [7], l2 ↦ [4] } },
 λenv . { { l1 ↦ [4], l2 ↦ [2] }, { l1 ↦ [ ], l2 ↦ [3] }, { l1 ↦ [7, 4], l2 ↦ [ ] } },
 λenv . { { l1 ↦ [4], l2 ↦ [2] }, { l1 ↦ [3], l2 ↦ [ ] }, { l1 ↦ [ ], l2 ↦ [7, 4] } },
 λenv . { { l1 ↦ [4], l2 ↦ [2] }, { l1 ↦ [3], l2 ↦ [ ] }, { l1 ↦ [7], l2 ↦ [4] } },
 λenv . { { l1 ↦ [4], l2 ↦ [2] }, { l1 ↦ [3], l2 ↦ [ ] }, { l1 ↦ [7, 4], l2 ↦ [ ] } },
 λenv . { { l1 ↦ [2], l2 ↦ [ ] }, { l1 ↦ [3], l2 ↦ [ ] }, { l1 ↦ [ ], l2 ↦ [7, 4] } },
 λenv . { { l1 ↦ [2], l2 ↦ [ ] }, { l1 ↦ [3], l2 ↦ [ ] }, { l1 ↦ [7], l2 ↦ [4] } },
 λenv . { { l1 ↦ [2], l2 ↦ [ ] }, { l1 ↦ [3], l2 ↦ [ ] }, { l1 ↦ [7, 4], l2 ↦ [ ] } }
```

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:

```
EvalExpr(pred) =
{λenv . if l1 ∈ dom(env) ∧ l2 ∈ dom(env)
   then 2 ∈ (elems(env(l1)) ∪ elems(env(l2)))
   else ⊥,
 λenv . if l1 ∈ dom(env) ∧ l2 ∈ dom(env)
   then 4 ∈ (elems(env(l1)) ∪ elems(env(l2)))
   else ⊥ }
```

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 expression with the same binding and the same predicate as the “unique quantified”-expression. Consider:

$$\text{iota} = \iota l_1 \curvearrowright l_2 \in (\text{let } s \in \{\{[4], [2, 5], 7\}, \{[2], [3], [7, 4]\}\} \text{ in } s) \cdot \\ \text{let } x \in \{2, 4\} \text{ in } x \in (\text{elems } l_1 \cup \text{elems } l_2)$$

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))) =
{  $\lambda \text{env} . \text{if } l_1 \in \underline{\text{dom}}(\text{env}) \wedge l_2 \in \underline{\text{dom}}(\text{env})$ 
   $\text{then } \underline{\text{join}}(\text{env}(l_1), \text{env}(l_2))$ 
   $\text{else } \perp$  }

```

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) =
{  $\lambda \text{env} . [2, 5], \lambda \text{env} . [2], \lambda \text{env} . [4], \lambda \text{env} . [7, 4]$  }

```

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  $\rightarrow$  LEEval
82.1  EvalExistsUniqueExpr(MkTag('ExistsUniqueExpr', (bind, pred)))  $\trianglelefteq$ 
     .2  let lbev = EvalBind(bind) in
     .3  PropE({  $\lambda \text{env} . \{ \text{if } b(\text{env}) = \{ \} \vee \underline{\text{err}} \in b(\text{env})$ 
      .4   $\text{then if } \{ \} = b(\text{env})$ 
      .5   $\text{then False}()$ 
      .6   $\text{else } \perp$ 
      .7   $\text{else if } \exists ! \text{venv} \in b(\text{env}) \cdot ev(\underline{\text{overwrite}}(\text{env}, \text{venv})) \notin \text{BOOL\_VAL}$ 
      .8   $\text{then } \perp$ 
      .9   $\text{else if } \exists ! \text{venv} \in b(\text{env}) \cdot ev(\underline{\text{overwrite}}(\text{env}, \text{venv})) = \text{True}()$ 
      .10  $\text{then True}()$ 
      .11  $\text{else False}()$ 
      .12  $| b \in lbev \}$ 
     .13  $| ev \in EvalExpr(pred) \}$ )

```

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.
- .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

5.3.6 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

5.3.7 Set Expressions

```

84.   EvalSetEnumeration : SetEnumeration → LEEval
84.1  EvalSetEnumeration(MkTag('SetEnumeration', els)) ≡
     .2  let el_lev_l = [ EvalExpr(els(i)) | i ∈ inds(els) ] in
     .3  let seq_lev_s = { ev_l | ev_l ∈ LL(EEval) ·
          .4      len(ev_l) = len(el_lev_l) ∧
          .5      ∀i ∈ inds(ev_l) · ev_l(i) ∈ el_lev_l(i) } in
     .6  { λenv . let setden = { ev(env) | ev ∈ elems(ev_l) } in
          .7      if ∀el ∈ setden · el ∈ FLATVAL \ { ⊥ }
          .8      then MkTag('set', setden)
          .9      else ⊥
          .10     | ev_l ∈ 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 = $\{a \mid a \in \{1, \dots, 5\} \cdot a \text{ rem } 2 = 0\}$

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 $a \in \{1, \dots, 5\}$, while the predicate is $a \text{ rem } 2 = 0$.

The set range expression, $\{1, \dots, 5\}$, has one expression evaluator:

```

EvalExpr( $\{1, \dots, 5\}$ ) =
{  $\lambda env . MkTag('set', \{ MkTag('num', 1),
                           MkTag('num', 2),
                           MkTag('num', 3),
                           MkTag('num', 4),
                           MkTag('num', 5) \}) }$ 

```

The predicate will ensure that only even numbers are considered. Thus, the semantics of *simple* is:

```

EvalExpr(simple) =
{  $\lambda env . MkTag('set', \{ MkTag('num', 2), MkTag('num', 4) \}) }$ 

```

The concrete syntax for the second (and much more complicated) example looks like:

```

set = {let e  $\in \{1, 2\}$  in mk-(e, i, l1)
       | i  $\in (\text{let } s \in \{\{4, 8, 3\}, \{5\}\} \text{ in } s), l1  $\curvearrowright$  l2  $\in \{[7, 4], [3], 7\}$  .
           let x  $\in \{4, 6\}$  in i  $\leq x$ }$ 
```

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:

$\text{EvalBind}(i \in (\text{let } s \in \{\{4, 8, 3\}, \{5\}\} \text{ in } s)) =$
 $\{\lambda\text{env} . \{\{i \mapsto 4\}, \{i \mapsto 8\}, \{i \mapsto 3\}\},$
 $\lambda\text{env} . \{\{i \mapsto 5\}\}\}$

while the semantics of the second binding is:

$\text{EvalBind}(l_1 \curvearrowright l_2 \in \{[7, 4], [3]\}, 7) =$
 $\{\lambda\text{env} . \{\{l_1 \mapsto [], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7, 4], l_2 \mapsto []\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7, 4], l_2 \mapsto []\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\}$

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 ($l_1 \curvearrowright l_2$) 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:

$\text{Partition}(\{\text{bind}_1, \text{bind}_2\}) =$
 $\{\{\lambda\text{env} . \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7, 4], l_2 \mapsto []\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 4\}, \{i \mapsto 3\}, \{i \mapsto 8\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7, 4], l_2 \mapsto []\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 5\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [], l_2 \mapsto [7, 4]\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 5\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 5\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7, 4], l_2 \mapsto []\}, \{l_1 \mapsto [], l_2 \mapsto [3]\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 5\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7], l_2 \mapsto [4]\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\}\},$
 $\{\lambda\text{env} . \{\{i \mapsto 5\}\},$
 $\lambda\text{env} . \{\{l_1 \mapsto [7, 4], l_2 \mapsto []\}, \{l_1 \mapsto [3], l_2 \mapsto []\}\}\}$

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:

```

EvalExpr(let  $e \in \{1, 2\}$  in  $mk\text{-}(e, i, l_1)$ ) =
{  $\lambda env$  .if  $i \in \underline{\text{dom}}(env) \wedge l_1 \in \underline{\text{dom}}(env)$ 
  then  $(1, env(i), env(l_1))$ 
  else  $\perp$ ,
 $\lambda env$  .if  $i \in \underline{\text{dom}}(env) \wedge l_1 \in \underline{\text{dom}}(env)$ 
  then  $(2, env(i), env(l_1))$ 
  else  $\perp$  }

```

and then the semantics of the predicate looks like:

```

EvalExpr(let  $x \in \{4, 6\}$  in  $i \leq x$ ) =
{  $\lambda env$  .if  $i \in \underline{\text{dom}}(env)$ 
  then  $env(i) \leq 4$ 
  else  $\perp$ ,
 $\lambda env$  .if  $i \in \underline{\text{dom}}(env)$ 
  then  $env(i) \leq 6$ 
  else  $\perp$  }

```

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 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:

```

EvalExpr(set) =
{  $\lambda env$  . { (1, 3, []), (1, 4, []) },
 $\lambda env$  . { (1, 3, [7]), (1, 4, [7]), (1, 3, []), (1, 4, []) },
 $\lambda env$  . { (1, 3, [7, 4]), (1, 4, [7, 4]), (1, 3, []), (1, 4, []) },
 $\lambda env$  . { (1, 3, []), (1, 4, []), (1, 3, [3]), (1, 4, [3]) },
 $\lambda env$  . { (1, 3, [7]), (1, 4, [7]), (1, 3, [3]), (1, 4, [3]) },
 $\lambda env$  . { (1, 3, [7, 4]), (1, 4, [7, 4]), (1, 3, [3]), (1, 4, [3]) },
 $\lambda env$  . { (1, 3, []), },
 $\lambda env$  . { (1, 5, []), },
 $\lambda env$  . { (1, 5, [7]), (1, 5, []) },
 $\lambda env$  . { (1, 5, [7, 4]), (1, 5, []) },
 $\lambda env$  . { (1, 5, []), (1, 5, [3]) },
 $\lambda env$  . { (1, 5, [7]), (1, 5, [3]) },
 $\lambda env$  . { (1, 5, [7, 4]), (1, 5, [3]) },
 $\lambda env$  . { (2, 3, []), (2, 4, []) },
 $\lambda env$  . { (2, 3, [7]), (2, 4, [7]), (2, 3, []), (2, 4, []) },
 $\lambda env$  . { (2, 3, [7, 4]), (2, 4, [7, 4]), (2, 3, []), (2, 4, []) },
 $\lambda env$  . { (2, 3, []), (2, 4, []), (2, 3, [3]), (2, 4, [3]) },
 $\lambda env$  . { (2, 3, [7]), (2, 4, [7]), (2, 3, [3]), (2, 4, [3]) },
 $\lambda env$  . { (2, 3, [7, 4]), (2, 4, [7, 4]), (2, 3, [3]), (2, 4, [3]) },
 $\lambda env$  . { (2, 5, []), },
 $\lambda env$  . { (2, 5, [7]), (2, 5, []) },
 $\lambda env$  . { (2, 5, [7, 4]), (2, 5, []) },
 $\lambda env$  . { (2, 5, []), (2, 5, [3]) },
 $\lambda env$  . { (2, 5, [7]), (2, 5, [3]) },
 $\lambda env$  . { (2, 5, [7, 4]), (2, 5, [3]) } }

```

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: $\text{let } x \in \{\text{true}, 4, 6\} \text{ in } i \leq x$ an expression evaluator returning bottom is added.

□

```

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_s} 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 ∃venus ∈ Partition(venv_ss) · Compatible(venus) ∧
     .10  pev(overwrite(env, Merge(venus))) ∈ BOOL_VAL
     .11  then ⊥
     .12  else let den = { ev(overwrite(env, Merge(venus)))
     .13  | venus ∈ Partition(venv_ss) · Compatible(venus) ∧
     .14  pev(overwrite(env, Merge(venus))) = 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: $\{ \text{elem} \mid \text{binds} \cdot \text{pred} \}$ 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

```

86.   EvalSetRange : SetRange → LEEval
86.1  EvalSetRange(MkTag('SetRange', (lb, ub))) ≡
.2    {  $\lambda \text{env}$  . let l_val = lb_ev(env),
.3      r_val = rb_ev(env) in
.4      if (l_val ∈ NUM_VAL)  $\wedge$  (r_val ∈ NUM_VAL)
.5      then let l_v = StripNumTagVal(l_val),
.6        r_v = StripNumTagVal(r_val) in
.7        let num_set = {x ∈  $\mathbb{Z}$  | l_v ≤ x ≤ r_v} in
.8        let num_val_s = {MkTag('num', x) | x ∈ num_set} in
.9          MkTag('set', num_val_s)
.10     else ⊥
.11   | lb_ev ∈ EvalExpr(lb), rb_ev ∈ EvalExpr(ub) }

```

Annotations to *EvalSetRange*:

- 86. This function gives meaning to set range expressions. A typical usage is: {1, ..., 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

5.3.8 Sequence Expressions

```

87.   EvalSeqEnumeration : SeqEnumeration → LEEval
87.1  EvalSeqEnumeration(MkTag('SeqEnumeration', els)) ≡
.2    let elev_l = [EvalExpr(els(i)) | i ∈ inds(els)] in
.3    let seq_lev_s = {ev_l | ev_l ∈ IL(EEEval) .
.4      len(ev_l) = len(elev_l)  $\wedge$ 
.5       $\forall i \in \text{inds}(\text{ev}_l) \cdot \text{ev}_l(i) \in \text{elev}_l(i)$  } in
.6      {  $\lambda \text{env}$  . let seqden = [ev_l(i)(env) | i ∈ inds(ev_l)] in
.7        if ⊥  $\notin \text{elems}(\text{seqden})$ 
.8        then MkTag('seq', seqden)
.9        else ⊥
.10      | ev_l ∈ 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

```
88. EvalSeqComprehension : SeqComprehension → LEEeval
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( $\bigcup\{\text{dom}(\text{venv}) \mid \text{venv} \in \text{bev}(\text{env})\}$ ),
.9             set =  $\bigcup\{\text{rng}(\text{venv}) \mid \text{venv} \in \text{bev}(\text{env})\}$  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

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 {λ*env*. 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

5.3.9 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 {λ*env*.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 enumer-

- ation 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

```

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

5.3.10 Tuple Constructor

```

92.   EvalTupleConstructor : TupleConstructor → LEEval
92.1  EvalTupleConstructor(MkTag('tuple', fields)) ≡
      .2  let f_lev_l = [EvalExpr(fields(i)) | i ∈ inds(fields)] in
      .3  let tup_lev = {ev_l | ev_l ∈  $\mathbb{L}(EEval)$  ·
          .4           len(ev_l) = len(f_lev_l) ∧
          .5            $\forall i \in \text{inds}(\text{ev}_l) \cdot \text{ev}_l(i) \in f_{\text{lev}}_l(i)$ } in
      .6  { $\lambda \text{env}$  . let tupden = [ev_l(i)(env) | i ∈ inds(ev_l)] in
          .7           if  $\perp \notin \text{elems}(\text{tupden})$ 
          .8           then MkTag('tuple', tupden)
          .9           else  $\perp$ 
      .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

5.3.11 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 ∈ IL(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)) ∈ IL(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 $\text{IL}(\text{VAL}) \rightarrow \text{RECORD_VAL}_\perp$ is also called *CONS_FUN* in the definition of the extended semantic domains.

End of annotations

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) ∈ ( $\text{IE}(\text{VAL}) \rightarrow \text{RECORD\_VAL} \rightarrow \text{VAL}$ )
      .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 ∈  $\text{IM}(\text{dom}(\text{modifiers}), \text{EEval})$  .
      .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, \perp will be returned because mfn will not be in $\text{dom}(env)$. 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

5.3.12 Apply Expressions

```

95.   EvalApplyExpr : Apply → LEEval
95.1  EvalApplyExpr(MkTag('Apply', (fct, arg))) ≡
.2    {  $\lambda env.let\ a = a\_ev(env),$ 
.3       $f = f\_ev(env)$  in
.4        if  $f \notin (\text{FUN\_VAL} \cup \text{MAP\_VAL} \cup \text{LIST\_VAL}) \vee a = \perp$ 
.5          then  $\perp$ 
.6        else cases f:
.7          MkTag('fun', fn) → if  $a \in \delta_0(fn)$ 
.8            then fn(a)
.9            else  $\perp$ ,
.10         MkTag('map', m) → if  $a \in \text{dom}(m)$ 
.11           then m(a)
.12           else  $\perp$ ,
.13         MkTag('seq', l) → if  $a \in \text{inds}(l)$ 
.14           then l(a)
.15           else  $\perp$ 
.16   |  $f\_ev \in \text{EvalExpr}(fct), a\_ev \in \text{EvalExpr}(arg)$  }

```

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 10).
- .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))) ≡
    { λenv.if ev(env) ∈ RECORD_VAL
      then let tag = ShowRecTagVal(ev(env)) in
        let sel_fn = GetRecSelId(tag, field) in
          if sel_fn ∈ dom(env) ∧ env(sel_fn) ∈ RECORD_VAL → VAL ∧
            ev(env) ∈ δ₀(env(sel_fn))
            then env(sel_fn)(ev(env))
            else ⊥
          else ⊥
      | 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

```

97. EvalFctTypeInstExpr : FctTypeInst → LEEval
97.1 EvalFctTypeInstExpr(MkTag('FctTypeInst', (id, inst))) ≡
    { λenv.let d_l = EvalTypeList(inst)(env) in
      if d_l = err ∨ ∃i ∈ inds(inst) · ¬CheckTagEnv(inst(i))
      then ⊥
      else if id ∈ dom(env) ∧ env(id) ∈ POLYVAL
      then env(id(d_l))
      else ⊥ }

```

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

5.3.13 Lambda Expression

```

98.   EvalLambda : Lambda → LEEval
98.1  EvalLambda(MkTag('Lambda', (MkTag('Par', (id, tp)), body))) ≡
.2    let tev = EvalType(tp) in
.3      let quasi_eval =  $\lambda \text{env} . \text{let } d = \text{tev}(\text{env}) \text{ in}$ 
.4        if d = err ∨  $\neg \text{CheckTagEnv}(\text{env}, \text{tp})$ 
.5          then  $\perp$ 
.6          else  $\lambda \text{ob} \in |d| . \{ \text{if } \text{ob} \in \|d\|$ 
.7            then ev(overwrite(env, {id ↦ ob}))
.8            else  $\perp$ 
.9            | ev ∈ EvalExpr(body) } in
.10   { eeval | eeval ∈ EEEval .
.11      $\forall \text{env} \in \text{ENV} .$ 
.12     (eeval(env) =  $\perp \wedge (\text{tev}(\text{env}) = \text{err} \vee \neg \text{CheckTagEnv}(\text{env}, \text{tp})) \vee$ 
.13     (ShowTag(eeval(env)) = 'Fun'  $\wedge \text{tev}(\text{env}) \neq \text{err} \wedge \text{CheckTagEnv}(\text{env}, \text{tp}) \wedge$ 
.14      $\delta_0(\text{StripTag}(\text{eeval}(\text{env}))) = |\text{tev}(\text{env})| \wedge$ 
.15      $\forall \text{ob} \in |\text{tev}(\text{env})| . \text{StripTag}(\text{eeval}(\text{env}))(ob) \in \text{quasi\_eval}(\text{env})(ob)\}$ 

```

Annotations to *EvalLambda*:

98. This function gives meaning to monomorphic lambda expressions. A *Lambda* expression looks like: $\lambda id : tp . body$ 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* → $\|d\| \rightarrow \text{IP}(\text{VAL})$ 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 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

5.3.14 Is Expression

```

99.   EvalIsExpr : IsExpr → LEEVal
99.1  EvalIsExpr(MkTag('IsExpr', (tag, arg))) ≡
.2    {  $\lambda \text{env}.$  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  {  $\lambda \text{env}.$  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.
- .2-.13 The different kinds of literals are transformed into corresponding basic values.

End of annotations

101. $\text{EvalId} : Id \rightarrow LEEval$
- 101.1 $\text{EvalId}(id) \triangleq$
- .2 $\{ \lambda env . \text{if } id \in \underline{\text{dom}}(env) \wedge env(id) \in VAL$
 - .3 $\quad \text{then } env(id)$
 - .4 $\quad \text{else } \perp \}$

Annotations to EvalId :

101. This function gives meaning to identifiers.
- End of annotations**

102. $\text{EvalOldId} : OldId \rightarrow LEEval$
- 102.1 $\text{EvalOldId}(id) \triangleq$
- .2 $\{ \lambda env . \text{if } id \in \underline{\text{dom}}(env)$
 - .3 $\quad \text{then } env(id)$
 - .4 $\quad \text{else } \perp \}$

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

5.4 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 section contains functions which modify a state designator and look up the value of a state designator in the environment.

103. $\text{Modify} : StateDesignator \times VAL \rightarrow LDef$
- 103.1 $\text{Modify}(sd, val) \triangleq$
- .2 cases $ShowTag(sd)$:
 - .3 ‘ ValueId ’ $\rightarrow \text{ModifyValueId}(sd, val)$,
 - .4 ‘ FieldRef ’ $\rightarrow \text{ModifyFieldRef}(sd, val)$,
 - .5 ‘ MapOrSeqRef ’ $\rightarrow \text{ModifyMapOrSeqRef}(sd, val)$

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

104. $\text{ModifyValueId} : \text{ValueId} \times \text{VAL} \rightarrow \text{LDef}$

104.1 $\text{ModifyValueId}(id, val) \triangleq$

.2 { $\lambda \text{env}.$ let $\text{te_id} = \text{MkTag}(\text{'TypeEnvId'}, ()$),
 .3 $\text{st_id} = \text{GetStateTypeId}(\text{env})$ in
 .4 if $\text{te_id} \in \text{dom}(\text{env}) \wedge \text{env}(\text{te_id}) \in \text{LOC_ENV}$
 .5 then if $\text{id} \in \text{dom}(\text{env}(\text{te_id}))$
 .6 then if $\text{val} \in \|\text{env}(\text{te_id})(\text{id})\|$
 .7 then overwrite(env , { $\text{id} \mapsto \text{val}$ })
 .8 else err
 .9 else $\text{ModifyState}(\text{id}, \text{val})(\text{env})$
 .10 else $\text{ModifyState}(\text{id}, \text{val})(\text{env})$ }

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. $\text{ModifyState} : \text{ValueId} \times \text{VAL} \rightarrow \text{Def}$

105.1 $\text{ModifyState}(\text{id}, \text{val})(\text{env}) \triangleq$

.2 let $\text{st_id} = \text{GetStateTypeId}(\text{env})$,
 .3 $\text{mod} = \{ \text{id} \mapsto \text{val} \}$ in
 .4 if $\text{st_id} = \text{nostate}$
 .5 then err
 .6 else let $\text{st_v_id} = \text{GetStateValId}(\text{CUR}, \text{st_id})$ in
 .7 if $\text{id} = \text{st_v_id}$
 .8 then if $\text{val} \in \|\text{env}(\text{st_id})\|$
 .9 then overwrite(env , mod)
 .10 else err
 .11 else let $\text{MkTag}(\text{'ValueId'}, \text{id}') = \text{id}$,
 .12 $\text{MkTag}(\text{'StateTypeId'}, \text{st_id}') = \text{st_id}$ in
 .13 if $\text{st_v_id} \in \text{dom}(\text{env}) \wedge$
 .14 $\text{MkTag}(\text{'RecModId'}, \text{st_id}') \in \text{dom}(\text{env}) \wedge$
 .15 $\text{env}(\text{MkTag}(\text{'RecModId'}, \text{st_id}')) \in (\text{IE}(\text{VAL}) \rightarrow \text{RECORD_VAL} \rightarrow \text{VAL})$
 .16 then let $\text{mod_id} = \text{MkTag}(\text{'RecModId'}, \text{st_id}')$,
 .17 $\text{old_st_v} = \text{env}(\text{st_v_id})$ in
 .18 let $\text{new_st} = \text{env}(\text{mod_id})(\text{mod})(\text{old_st_v})$ in
 .19 if $\text{new_st} \in \|\text{env}(\text{st_id})\|$
 .20 then let $\text{st_m} = \{ \text{st_v_id} \mapsto \text{new_st} \}$ 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

106. $\text{ModifyFieldRef} : \text{FieldRef} \times \text{VAL} \rightarrow \text{LDef}$

106.1 $\text{ModifyFieldRef}(\text{MkTag}(\text{'FieldRef'}, (sd, sel)), val) \triangleq$
 .2 $\text{PropErr}(\{\lambda \text{env}. \text{if } \text{rev}(\text{env}) \notin \text{RECORD_VAL}$
 .3 $\text{then } \{\underline{\text{err}}\}$
 .4 $\text{else let } tag = \text{ShowRecTagVal}(\text{rev}(\text{env})) \text{ in}$
 .5 $\text{let } mfn = \text{GetRecModId}(tag) \text{ in}$
 .6 $\text{if } mfn \in \underline{\text{dom}}(\text{env}) \wedge$
 .7 $\text{env}(mfn) \in (\text{IE}(\text{VAL}) \rightarrow \text{RECORD_VAL} \rightarrow \text{VAL})$
 .8 $\text{then } \text{Modify}(sd, \text{env}(mfn)(\{sel \mapsto val\})(\text{rev}(\text{env})))$
 .9 $\text{else } \{\underline{\text{err}}\}$
 .10 $| \text{rev} \in \text{LookUp}(sd)\})$

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. $\text{ModifyMapOrSeqRef} : \text{MapOrSeqRef} \times \text{VAL} \rightarrow \text{LDef}$

107.1 $\text{ModifyMapOrSeqRef}(\text{MkTag}(\text{'MapOrSeqRef'}, (sd, sel)), val) \triangleq$
 .2 $\text{PropErr}(\{\lambda \text{env}. \text{if } sdev(\text{env}) \notin \text{MAP_VAL}$
 .3 $\text{then if } sdev(\text{env}) \notin \text{LIST_VAL}$
 .4 $\text{then } \{\underline{\text{err}}\}$
 .5 $\text{else let } \text{MkTag}(\text{'seq'}, l) = sdev(\text{env}) \text{ in}$
 .6 $\text{if } sev(\text{env}) \in \underline{\text{inds}}(l)$
 .7 $\text{then let } l' = [\text{if } i = sev(\text{env})$
 .8 $\text{then } val$
 .9 $\text{else } l(i)$
 .10 $| \text{i} \in \underline{\text{inds}}(l)] \text{ in}$
 .11 $\text{Modify}(sd, \text{MkTag}(\text{'seq'}, l'))$
 .12 $\text{else } \{\underline{\text{err}}\}$
 .13 $\text{else let } \text{MkTag}(\text{'map'}, m) = sdev(\text{env}) \text{ in}$
 .14 $\text{if } sev(\text{env}) \in \underline{\text{dom}}(m)$
 .15 $\text{then let } v = \underline{\text{overwrite}}(m, \{sev(\text{env}) \mapsto val\}) \text{ in}$
 .16 $\text{Modify}(sd, \text{MkTag}(\text{'map'}, v))$
 .17 $\text{else } \{\underline{\text{err}}\}$
 .18 $| \text{sdev} \in \text{LookUp}(sd), sev \in \text{EvalExpr}(sel)\})$

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.

End of annotations

108. *LookUp* : *StateDesignator* → *LEEeval*

108.1 *LookUp(st)* \triangleq

- .2 cases *ShowTag(st)* :
- .3 'ValueId' → { λenv . if *st* ∈ dom(*env*)
 then *env(st)*
 else \perp },
- .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* → *LEEeval*

109.1 *LookUpFieldRef(MkTag('FieldRef', (sd, sel)))* \triangleq

- .2 { λenv .if *rev(env)* \notin RECORD_VAL
 then \perp
 else let *tag* = *ShowRecTagVal(rev(env))* in
 let *sfn* = *GetRecSelId(tag, sel)* in
 if *sfn* ∈ dom(*env*) \wedge *env(sfn)* ∈ RECORD_VAL → VAL
 then *env(sfn)(rev(env))*
 else \perp
 | *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 type⁶ (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

⁶The domain containing record selector functions is also called *SEL_FUN* in the definition of the extended semantic domains.

```

110.   LookUpMapOrSeqRef: MapOrSeqRef → LEEval
110.1  LookUpMapOrSeqRef(MkTag('MapOrSeqRef', (sd, sel))) ≡
.2    {  $\lambda \text{env}.$ if sdev(env)  $\notin \text{MAP\_VAL}$ 
.3      then if sdev(env)  $\notin \text{LIST\_VAL}$ 
.4        then  $\perp$ 
.5        else let MkTag('seq', l) = sdev(env) in
.6          if sev(env)  $\in \text{inds}(l)$ 
.7            then l(sev(env))
.8            else  $\perp$ 
.9        else let MkTag('map', m) = sdev(env) in
.10       if sev(env)  $\in \text{dom}(m)$ 
.11       then m(sev(env))
.12       else  $\perp$ 
.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

5.5 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.

5.5.1 Underlying Theory For Statements

This subsubsection 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 $F(f) = \text{comp}(b, f)$, 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 $f_0 = \perp$ and $f_{n+1} = \text{comp}(b, f_n)$, $n \in \mathbb{N}$.

Let *X* be a cpo ordered by \sqsubseteq_X and with bottom \perp . Let *B* be any subset of $|X|$, possibly without \perp , it is intended to be the denotation of the body of a nondeterministic loop. Let *comp* : *B* × *X* → *X* be a function continuous with respect to the second argument. Define a set function *NonDetComp*⁷ by the formula: $\text{NonDetComp}(A) = \{ \text{comp}(b, a) \mid b \in B \wedge a \in A \}$. 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.

⁷*NonDetComp* will be a functional similar to *F* for dealing with loops which contains non-determinism.

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 $(H : \mathbb{L}(B) \rightarrow X)$:

$$\begin{cases} H([]) &= \perp \\ H(s) &= \text{comp}(\underline{\text{hd}}(s), H(\underline{\text{tl}}(s))), \quad s \neq [] \end{cases} \quad (5.1)$$

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:

$$H(s) = \bigsqcup_X (H(s_n) \mid n \in \mathbb{N}), \quad (5.2)$$

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 (5.1) is still valid for infinite sequences. For,

$$\begin{aligned} H(s) &= \bigsqcup_X \{H(s_n) \mid n \geq N\} \\ &= \bigsqcup_X \{\text{comp}(\underline{\text{hd}}(s_n), H(\underline{\text{tl}}(s_n))) \mid n \in \mathbb{N}\} \\ &= \bigsqcup_X \{\text{comp}(\underline{\text{hd}}(s), H(\underline{\text{tl}}(s_n))) \mid n \in \mathbb{N}\} \\ &= \text{comp}(\underline{\text{hd}}(s), \bigsqcup_X \{H(\underline{\text{tl}}(s_n)) \mid n \in \mathbb{N}\}) \\ &= \text{comp}(\underline{\text{hd}}(s), H(\underline{\text{tl}}(s))). \end{aligned}$$

These five steps can be explained:

1. By the definition in (5.2).
2. By (5.1) applied to finite sequences.
3. Because $\underline{\text{hd}}(s) = \underline{\text{hd}}(s_n)$.
4. By the continuity of comp .
5. By (5.2) applied to the tail of s , and the continuity of $\underline{\text{tl}}$.

The set of all infinite iterations $\{H(s) \mid s \in \mathbb{L}_\omega(B)\}$ will be denoted by an expression $\text{NonDetIter}(B, \text{comp})$, where NonDetIter is formally defined as one of the semantic functions. This set is a fixed point of NonDetComp , formally:

$$\text{NonDetIter}(B, \text{comp}) = \{\text{comp}(b, a) \mid b \in B \wedge a \in \text{NonDetIter}(B, \text{comp})\},$$

as can be seen by:

$$\begin{aligned} \{\text{comp}(b, a) \mid b \in B \wedge a \in \text{NonDetIter}(B, \text{comp})\} &= \{\text{comp}(b, H(s)) \mid b \in B \wedge s \in \mathbb{L}_\omega(B)\} \\ &= \{H(\underline{\text{join}}_\omega([b], s)) \mid b \in B \wedge s \in \mathbb{L}_\omega(B)\} \\ &= \{H(s) \mid s \in \mathbb{L}_\omega(B)\} \\ &= \text{NonDetIter}(B, \text{comp}) \end{aligned}$$

This construction will be used to define the semantics of non-deterministic iterations (*While* and *RecTrapStmt*). 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 a simple extension of the $\underline{\text{join}}$ operator and it is defined as:

$$\begin{aligned} \underline{\text{join}}_\omega : \mathbb{L}(A) \times \mathbb{L}_\omega(A) &\rightarrow \mathbb{L}_\omega(A) \\ \underline{\text{join}}_\omega([a_1, \dots, a_n], [b_1, \dots]) &= [a_1, \dots, a_n, b_1, \dots] \end{aligned}$$

5.5.2 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

5.5.3 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) } ∪
              dom(explfns) ∪ dom(implfns) in
          .4  let env'' = overwrite(env, fnenv) in
              .5  if def(env'') = err ∨ BotEnv(def(env''))
              .6  then (env, ret, ⊥)
              .7  else Restore(env, ids, ev(def(env'')))
          .8  | def ∈ EvalLocalDef(vals, explfns, implfns), ev ∈ EvalStmt(in), env' ∈ ENV ·
          .9  VerifyExplFns(explfns, env') ∧ VerifyImplFns(implfns, env') } in
          .10
          .11  if lsev = { }
          .12  then { λenv . (env, ret, ⊥) }
          .13  else lsev

```

Annotations to *EvalLetStmt*:

112. This function gives meaning to *Let* statements⁸.
- .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. $\text{EvalLetBeSTStmt} : \text{LetBeSTStmt} \rightarrow \text{LSEval}$
- 113.1 $\text{EvalLetBeSTStmt}(\text{MkTag}(\text{'LetBeSTStmt'}, (\text{bind}, \text{st}, \text{in}))) \triangleq$
- .2 $\text{let } \text{ids} = \text{ColIdSet}(\text{bind}) \text{ in}$
- .3 $\text{PropS}(\{\lambda \text{env}. \text{let } \text{venv_s} = \bigcup \{ \text{bev}(\text{env}) \mid \text{bev} \in \text{EvalBind}(\text{bind})\} \text{ in}$
- .4 $\text{let } \text{venv_s}' = \{ \text{venv} \mid \text{venv} \in \text{venv_s} \cdot \text{venv} \neq \text{err} \wedge \neg \text{BotEnv}(\text{venv}) \wedge$
- .5 $\text{stev}(\underline{\text{overwrite}}(\text{env}, \text{venv})) = \text{True}() \} \text{ in}$
- .6 $\{ \text{Restore}(\text{env}, \text{ids}, \text{inev}(\underline{\text{overwrite}}(\text{env}, \text{venv}))) \mid \text{venv} \in \text{venv_s}'\}$
- .7 $\mid \text{stev} \in \text{EvalExpr}(\text{st}), \text{inev} \in \text{EvalStmt}(\text{in}) \})$

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.
- .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

⁸Notice 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.

```

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 statements⁹.
- .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

⁹Notice 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).

5.5.4 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.
 - .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 { $\lambda \text{env} . (\text{env}, \text{cont}, \text{nil})$ }
 - .4 else { $\lambda \text{env}.\text{let } (\text{env}', \text{mode}, \text{val}) = \text{sev}_1(\text{env}) \text{ in}$
 - .5 if *mode* ≠ cont
 - .6 then (*env'*, *mode*, *val*)
 - .7 else *sev*₂(*env'*)
 - .8 | *sev*₁ ∈ *EvalStmt(hd(stmt_l))*,
 - .9 | *sev*₂ ∈ *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)})

```

Annotations to *EvalAssign*:

- 117. This function gives meaning to assignment statements¹⁰.
- .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

¹⁰As in the *Def* statement an *Assignment* statement may have side effects by means of calls to value returning operations.

5.5.5 Conditional Statements

118. $\text{EvalIfStmt} : \text{IfStmt} \rightarrow \text{LSEval}$

118.1 $\text{EvalIfStmt}(\text{MkTag}(\text{'IfStmt'}, (\text{test}, \text{cons}, \text{altn}))) \triangleq$
 .2 $\{ \lambda \text{env}. \text{if } \text{testev}(\text{env}) \notin \text{BOOL_VAL}$
 .3 $\text{then } (\text{env}, \underline{\text{ret}}, \perp)$
 .4 $\text{else if } \text{testev}(\text{env}) = \text{True}()$
 .5 $\text{then } \text{consev}(\text{env})$
 .6 $\text{else } \text{altnev}(\text{env})$
 .7 $| \text{testev} \in \text{EvalExpr}(\text{test}), \text{consev} \in \text{EvalStmt}(\text{cons}), \text{altnev} \in \text{EvalStmt}(\text{altn}) \}$

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. $\text{EvalCasesStmt} : \text{CasesStmt} \rightarrow \text{LSEval}$

119.1 $\text{EvalCasesStmt}(\text{MkTag}(\text{'CasesStmt'}, (\text{sel}, \text{altns}))) \triangleq$
 .2 $\text{PropS}(\{ \lambda \text{env}. \text{if } \text{selev}(\text{env}) = \perp$
 .3 $\text{then } \{ (\text{env}, \underline{\text{ret}}, \perp) \}$
 .4 $\text{else } \{ \text{sev}(\text{env}) | \text{sev} \in \text{ApplyCasesStmt}(\text{selev}(\text{env}), \text{altns}) \}$
 .5 $| \text{selev} \in \text{EvalExpr}(\text{sel}) \})$

Annotations to *EvalCasesStmt*:

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. $\text{ApplyCasesStmt} : \text{VAL} \times \text{IL}(\text{CaseStmtAltn}) \rightarrow \text{LSEval}$

120.1 $\text{ApplyCasesStmt}(\text{val}, \text{csa_l}) \triangleq$
 .2 $\text{if } \underline{\text{len}}(\text{csa_l}) = 0$
 .3 $\text{then } \{ \lambda \text{env}. (\text{env}, \underline{\text{cont}}, \underline{\text{nil}}) \}$
 .4 $\text{else } \{ \lambda \text{env}. \text{let } \text{res} = \text{caev}(\text{env}) \text{ in }$
 .5 $\text{if } \text{res} = \underline{\text{unmatch}}$
 .6 $\text{then } c(\text{env})$
 .7 $\text{else } \text{res}$
 .8 $| \text{caev} \in \text{ApplyCaseStmtAltn}(\text{val}, \underline{\text{hd}}(\text{csa_l})),$
 .9 $c \in \text{ApplyCasesStmt}(\text{val}, \underline{\text{tl}}(\text{csa_l})) \}$

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. $ApplyCaseStmtAltn : VAL \times CaseStmtAltn \rightarrow LSEval \cup \{\underline{unmatch}\}$
- 121.1 $ApplyCaseStmtAltn(val, MkTag('CaseStmtAltn', (pat, body))) \triangleq$
 - .2 $PropS(\{\lambda env.let s = \{ Restore(env, ColIdSet(pat), boev(env')) | boev \in EvalStmt(body), env' \in NonErrPat(pat, val, env) \} in$
 - .3 $\quad \quad \quad \text{if } s = \{ \}$
 - .4 $\quad \quad \quad \text{then } \{\underline{unmatch}\}$
 - .5 $\quad \quad \quad \text{else } s$
 - .6 $\quad \quad \quad \})$

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

5.5.6 Loop Statements

122. $EvalSeqForLoop : SeqForLoop \rightarrow LSEval$
- 122.1 $EvalSeqForLoop(MkTag('SeqForLoop', (cv, dirn, seq, body))) \triangleq$
 - .2 $PropS(\{\lambda env.\text{if } L_ev(env) \notin LIST_VAL$
 - .3 $\quad \quad \quad \text{then } \{(env, \underline{ret}, \perp)\}$
 - .4 $\quad \quad \quad \text{else let } L_v = \text{if } dirn = \text{FORWARDS}$
 - .5 $\quad \quad \quad \quad \quad \text{then } StripSeqTagVal(L_ev(env))$
 - .6 $\quad \quad \quad \quad \quad \text{else } Reverse(StripSeqTagVal(L_ev(env))) \text{ in}$
 - .7 $\quad \quad \quad \quad \quad \{ sev(env) \mid sev \in DoSeqLoop(cv, L_v, body) \}$
 - .8 $\quad \quad \quad \quad \quad \mid L_ev \in EvalExpr(seq) \})$

Annotations to *EvalSeqForLoop*:

122. This function gives meaning to *For Loops* 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 FORWARD 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*({ $\lambda \text{env}.$ if *s-ev*(*env*) \notin SET_VAL
 .3 then {(*env*, ret, \perp) }
 .4 else let *s-v* = *StripSetTagVal*(*s-ev*(*env*)) in
 .5 $\bigcup\{\{ \text{sev}(\text{env}) \mid \text{sev} \in \text{DoSeqLoop}(\text{cv}, \text{l-v}, \text{body})\}$
 .6 $\mid \text{l-v} \in \text{SetToSeq}(\text{s-v})\}$
 .7 | *s-ev* ∈ *EvalExpr*(*set*) })

Annotations to *EvalSetForLoop*:

123. This function gives meaning to *For Loops* 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

124. *EvalIndexForLoop* : *IndexForLoop* → *LSEval*
 124.1 *EvalIndexForLoop*(*MkTag*('IndexForLoop', (*cv*, *lb*, *ub*, *by*, *body*))) ≡
 .2 *PropS*({ $\lambda \text{env}.$ if *lbev*(*env*) \notin NUM_VAL ∨
 .3 *ubev*(*env*) \notin NUM_VAL ∨
 .4 *stev*(*env*) \notin NUM_VAL
 .5 then {(*env*, ret, \perp) }
 .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* $\notin \mathbb{Z}$ ∨ *ub-v* $\notin \mathbb{Z}$ ∨ *st-v* $\notin \mathbb{Z}$
 .10 then {(*env*, ret, \perp) }
 .11 else let *seq* = *StepsToSeq*(*lb-v*, *ub-v*, *st-v*) in
 .12 if *seq* = err
 .13 then {(*env*, ret, \perp) }
 .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* × $\text{IL}(\text{VAL})$ × *Stmt* → *LSEval*

125.1 *DoSeqLoop*(*pat*, *lv*, *body*) \triangleq

- .2 if *len(lv)* = 0
- .3 then { λenv . (*env*, *cont*, *nil*) }
- .4 else let *ids* = *CollIdSet*(*pat*) in
 - .5 *PropS*($\{\lambda \text{env}.\{\text{let } (\text{env}'', \text{mode}, \text{val}) = \text{sev}(\text{env}') \text{ in}$
 - .6 if *mode* ≠ *cont*
 - .7 then *Restore*(*env*, *ids*, *sev*(*env*'))
 - .8 else *Restore*(*env*, *ids*, *sev'*(*env*''))
 - .9 | *env*' ∈ *NonErrPat*(*pat*, *hd(lv)*, *env*),
 - .10 | *sev'* ∈ *DoSeqLoop*(*pat*, *tl(lv)*, *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 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. $\text{EvalWhileLoop} : \text{WhileLoop} \rightarrow \text{LSEval}$

126.1 $\text{EvalWhileLoop}(\text{MkTag}(\text{'WhileLoop}', (\text{test}, \text{body}))) \triangleq$

.2 let $\text{LoopDen} = \text{ENV} \simeq ((\text{ENV} \times \text{MODE} \times (\text{VAL} \cup \{\underline{\text{nil}}\})) \times \text{VAL})$,

.3 $\text{BodyDen} = \{ \lambda \text{env}. \text{let } (\text{env}', \text{mode}, \text{val}) = \text{boev}(\text{env}) \text{ in}$

.4 $((\text{env}', \text{mode}, \text{val}), \text{tev}(\text{env}'))$
|.5 $| \text{boev} \in \text{EvalStmt}(\text{body}), \text{tev} \in \text{EvalExpr}(\text{test}) \} \text{ in}$

.6 let $\text{comp} = \lambda \text{comp_one} \in \text{BodyDen}, \text{comp_rest} \in \text{LoopDen}. \lambda \text{env}.$

.7 let $((\text{env}', \text{mode}, \text{val}'), \text{val}) = \text{comp_one}(\text{env}) \text{ in}$

.8 if $\text{mode} \neq \underline{\text{cont}}$
|.9 then $((\text{env}', \text{mode}, \text{val}'), \text{val})$
|.10 else if $\text{val} \notin \text{BOOL_VAL}$
|.11 then $((\text{env}, \underline{\text{ret}}, \perp), \perp)$
|.12 else if $\text{val} = \text{False}()$
|.13 then $((\text{env}', \underline{\text{cont}}, \underline{\text{nil}}), \text{False}())$
|.14 else if $\text{env}' \in \delta_0(\text{comp_rest})$
|.15 then $\text{comp_rest}(\text{env}') \text{ in}$

.16 $\{ \lambda \text{env}. \text{if } \text{tev}(\text{env}) \notin \text{BOOL_VAL}$
|.17 then $(\text{env}, \underline{\text{ret}}, \perp)$
|.18 else if $\text{tev}(\text{env}) = \text{False}()$
|.19 then $(\text{env}, \underline{\text{cont}}, \underline{\text{nil}})$
|.20 else if $\text{env} \notin \delta_0(\text{comp_rest})$
|.21 then $(\text{env}, \underline{\text{ret}}, \perp)$
|.22 else $\pi_1(\text{comp_rest}(\text{env}))$

.23 | $\text{tev} \in \text{EvalExpr}(\text{test}), \text{comp_rest} \in \text{NonDetIter}(\text{BodyDen}, \text{comp}) \}$

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.
- .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

5.5.7 Non-Deterministic Sequences

127. $\text{EvalNonDetStmt} : \text{NonDetStmt} \rightarrow \text{LSEval}$
 127.1 $\text{EvalNonDetStmt}(\text{MkTag}(\text{'NonDetStmt'}, \text{stmt_l})) \triangleq$
 .2 $\bigcup \{ \text{EvalSequence}(\text{MkTag}(\text{'Sequence'}, \text{s_l})) \mid \text{s_l} \in \text{Permutations}(\text{stmt_l}) \}$

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

5.5.8 Call and Return Statements

128. $\text{EvalCall} : \text{Call} \rightarrow \text{LSEval}$
 128.1 $\text{EvalCall}(\text{MkTag}(\text{'Call'}, (\text{oprt}, \text{arg}, \text{st}))) \triangleq$
 .2 $\text{PropS}(\{ \lambda \text{env}. \text{if } \text{oprt} \notin \underline{\text{dom}}(\text{env}) \vee \text{env}(\text{oprt}) \notin \text{OPVAL}$
 .3 $\text{then} \{ (\text{env}, \underline{\text{ret}}, \perp) \}$
 .4 $\text{else} \bigcup \{ \text{if } \text{st} = \underline{\text{nil}}$
 .5 $\text{then} \{ (\text{env}, \text{mode}, \text{out_v}) \}$
 .6 $\text{else let } \text{m_s} = \{ \text{m}(\text{env}) \mid \text{m} \in \text{Modify}(\text{st}, \text{out_st}) \} \text{ in}$
 .7 $\{ (\text{e}, \text{mode}, \text{out_v}) \mid \text{e} \in \text{m_s} \cdot \text{e} \neq \underline{\text{err}} \} \cup$
 .8 $\{ (\text{env}, \underline{\text{ret}}, \perp) \mid \underline{\text{err}} \in \text{m_s} \}$
 .9 $\mid (\text{in_v}, \text{in_st}, \text{out_st}, \text{out_v}, \text{mode}) \in \text{env}(\text{oprt}),$
 .10 $\text{in_st} \in \text{if } \text{st} = \underline{\text{nil}}$
 .11 $\text{then let } \text{st_id} = \text{GetStateTypeId}(\text{env}) \text{ in}$
 .12 $\text{if } \text{st_id} = \underline{\text{nostate}}$
 .13 $\text{then} \{ \underline{\text{nil}} \}$
 .14 $\text{else} \{ \text{env}(\text{GetStateValId}(\text{CUR}, \text{st_id})) \}$
 .15 $\text{else} \{ \text{ev}(\text{env}) \mid \text{ev} \in \text{LookUp}(\text{st}) \},$
 .16 $\text{in_v} \in \{ \text{ev}(\text{env}) \mid \text{ev} \in \text{EvalExpr}(\text{arg}) \} \} \cup$
 .17 $\{ (\text{env}, \underline{\text{ret}}, \perp)$
 .18 $\mid \text{in_st} \in \text{if } \text{st} = \underline{\text{nil}}$
 .19 $\text{then let } \text{st_id} = \text{GetStateTypeId}(\text{env}) \text{ in}$
 .20 $\text{if } \text{st_id} = \underline{\text{nostate}}$
 .21 $\text{then} \{ \underline{\text{nil}} \}$
 .22 $\text{else} \{ \text{env}(\text{GetStateValId}(\text{CUR}, \text{st_id})) \}$
 .23 $\text{else} \{ \text{ev}(\text{env}) \mid \text{ev} \in \text{LookUp}(\text{st}) \},$
 .24 $\text{in_v} \in \{ \text{ev}(\text{env}) \mid \text{ev} \in \text{EvalExpr}(\text{arg}) \} \cdot$
 .25 $\forall \text{out_st}, \text{out_v} \in \text{VAL}, \text{m} \in \text{MODE} \cdot$
 .26 $(\text{in_v}, \text{in_st}, \text{out_st}, \text{out_v}, \text{m}) \notin \text{env}(\text{oprt}) \}$
 .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

129. *EvalReturnStmt* : *ReturnStmt* → *LSEval*
- 129.1 *EvalReturnStmt*(*MkTag*('ReturnStmt', *expr*)) ≡
- .2 if *expr* = nil
 .3 then { $\lambda \text{env}.$ (*env*, ret, nil) }
 .4 else { $\lambda \text{env}.$ (*env*, ret, *ev*(*env*)) | *ev* ∈ *EvalExpr*(*expr*) }

Annotations to *EvalReturnStmt*:

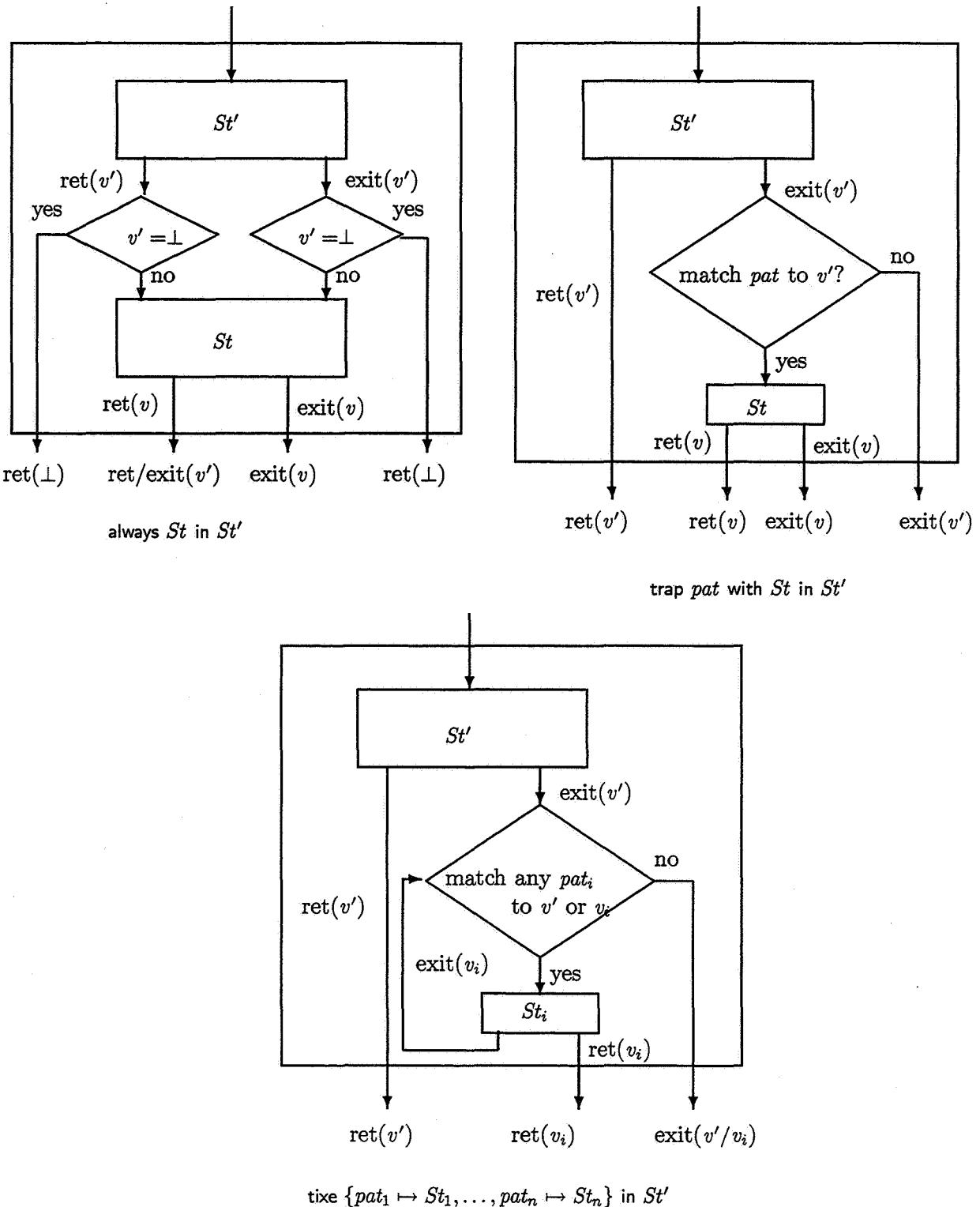
129. This function gives meaning to return statements.
- .2-.3 If no value is returned the return statement denotes one statement evaluator with nil as a value component.
- .4 Otherwise there will be one statement evaluator per expression evaluator.

End of annotations

5.5.9 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.



□

130. $\text{EvalAlways} : \text{Always} \rightarrow \text{LSEval}$
 130.1 $\text{EvalAlways}(\text{MkTag}(\text{'Always'}, (\text{post}, \text{body}))) \triangleq$
 .2 $\{\lambda \text{env}. \text{let } (\text{env}', \text{mode}, \text{val}) = \text{boev}(\text{env}) \text{ in}$
 .3 $\text{if } \text{val} = \perp$
 .4 $\text{then } \text{boev}(\text{env})$
 .5 $\text{else let } (\text{env}'', \text{mode}', \text{val}') = \text{poev}(\text{env}') \text{ in}$
 .6 $\text{if } \text{mode}' \neq \underline{\text{exit}}$
 .7 $\text{then } (\text{env}'', \text{mode}, \text{val})$
 .8 $\text{else } \text{poev}(\text{env}')$
 .9 $| \text{boev} \in \text{EvalStmt}(\text{body}), \text{poev} \in \text{EvalStmt}(\text{post}) \}$

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. $\text{EvalTrapStmt} : \text{TrapStmt} \rightarrow \text{LSEval}$
 131.1 $\text{EvalTrapStmt}(\text{MkTag}(\text{'TrapStmt'}, (\text{pat}, \text{post}, \text{body}))) \triangleq$
 .2 $\text{let } \text{ids} = \text{ColIdSet}(\text{pat}) \text{ in}$
 .3 $\text{PropS}(\{\lambda \text{env}. \text{let } (\text{env}', \text{mode}, \text{val}) = \text{boev}(\text{env}) \text{ in}$
 .4 $\text{if } \text{mode} \neq \underline{\text{exit}} \vee \text{val} = \underline{\text{nil}}$
 .5 $\text{then } \{(\text{env}', \text{mode}, \text{val})\}$
 .6 $\text{else let } s = \{ \text{Restore}(\text{env}', \text{ids}, \text{poev}(\text{nenv}))$
 .7 $| \text{poev} \in \text{EvalStmt}(\text{post}),$
 .8 $\text{nenv} \in \text{NonErrPat}(\text{pat}, \text{val}, \text{env}') \} \text{ in}$
 .9 $\text{if } s = \{ \}$
 .10 $\text{then } \{(\text{env}', \underline{\text{exit}}, \text{val})\}$
 .11 $\text{else } s$
 .12 $| \text{boev} \in \text{EvalStmt}(\text{body}) \}$

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

```

132.   EvalRecTrapStmt : RecTrapStmt → LSEval
132.1  EvalRecTrapStmt(MkTag('RecTrapStmt', (traps, body))) ≅
.2    let ids =  $\bigcup\{\text{ColIdSet}(p) \mid p \in \underline{\text{dom}}(\text{traps})\}$  in
.3    let RecDen =  $\text{ENV} \times \text{VAL} \simeq ((\text{ENV} \times \text{MODE} \times (\text{VAL} \cup \{\underline{\text{nil}}\}))$ 
.4     $\times \{\underline{\text{match}}, \underline{\text{unmatch}}\}),$ 
.5    TrapDen = PropS( $\{\lambda \text{env} \in \text{ENV}, \text{val} \in \text{VAL} .$ 
.6      let s =  $\bigcup\{\{\text{Restore}(\text{env}, \text{ids}, \text{poev}(\text{nenv}))$ 
.7        | poev ∈ EvalStmt(traps(p)),
.8        nenv ∈ NonErrPat(p, val, env)\}
.9        | p ∈  $\underline{\text{dom}}(\text{traps})\}$  in
.10       if s = { }
.11       then  $\{( (\text{env}, \underline{\text{exit}}, \text{val}), \underline{\text{unmatch}} ) \}$ 
.12       else  $\{ (e, \underline{\text{match}}) \mid e \in s \}\}$  in
.13     let comp =  $\lambda \text{comp\_one} \in \text{TrapDen}, \text{comp\_rest} \in \text{RecDen} .$ 
.14        $\lambda \text{env} \in \text{ENV}, \text{val} \in \text{VAL} .$ 
.15       let  $((\text{env}', \text{mode}', \text{val}'), m) = \text{comp\_one}(\text{env}, \text{val})$  in
.16         if m =  $\underline{\text{unmatch}}$  ∨ mode' ≠  $\underline{\text{exit}}$  ∨ val =  $\underline{\text{nil}}$ 
.17         then  $((\text{env}', \text{mode}', \text{val}'), m)$ 
.18         else if  $(\text{env}', \text{val}') \in \delta_0(\text{comp\_rest})$ 
.19           then comp\_rest(env', val') in
.20            $\{\lambda \text{env}. \text{let } (\text{env}', \text{mode}, \text{val}) = \text{boev}(\text{env})$  in
.21             if mode ≠  $\underline{\text{exit}}$  ∨ val =  $\underline{\text{nil}}$ 
.22             then  $(\text{env}', \text{mode}, \text{val})$ 
.23             else if  $(\text{env}', \text{val}) \notin \delta_0(\text{comp\_rest})$ 
.24               then  $(\text{env}, \underline{\text{ret}}, \perp)$ 
.25               else  $\pi_1(\text{comp\_rest}(\text{env}', \text{val}))$ 
.26           | boev ∈ EvalStmt(body), comp\_rest ∈ NonDetIter(TrapDen, comp)\}

```

Annotations to *EvalRecTrapStmt*:

132. This function gives meaning to *Recursive 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 $\underline{\text{exit}}$, 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.
- .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 { $\lambda \text{env}.$ (*env*, exit, nil) }
 - .4 else let *lev* = *EvalExpr*(*expr*) in
 - .5 { $\lambda \text{env}.$ if *ev*(*env*) = \perp
 - .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

5.6 Patterns and Bindings

In this section 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 value¹¹. 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 considered¹². 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:

¹¹When 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.

¹²A ‘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.

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	(<i>env</i> , <i>ret</i> , ⊥) 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.

5.6.1 Patterns

134. $\text{EvalPattern} : \text{Pattern} \rightarrow \text{LPatEval}$

134.1 $\text{EvalPattern}(\text{pat}) \triangleq$

- .2 **cases** *ShowTag*(*pat*) :
- .3 ‘*PatternId*’ $\rightarrow \text{EvalPatternId}(\text{pat}),$
- .4 ‘*MatchVal*’ $\rightarrow \text{EvalMatchVal}(\text{pat}),$
- .5 ‘*SeqEnumPattern*’ $\rightarrow \text{EvalSeqEnumPattern}(\text{pat}),$
- .6 ‘*SeqConcPattern*’ $\rightarrow \text{EvalSeqConcPattern}(\text{pat}),$
- .7 ‘*TuplePattern*’ $\rightarrow \text{EvalTuplePattern}(\text{pat}),$
- .8 ‘*RecordPattern*’ $\rightarrow \text{EvalRecordPattern}(\text{pat}),$
- .9 ‘*SetEnumPattern*’ $\rightarrow \text{EvalSetEnumPattern}(\text{pat}),$
- .10 ‘*SetUnionPattern*’ $\rightarrow \text{EvalSetUnionPattern}(\text{pat}),$
- .11 ‘*SetConstrPattern*’ $\rightarrow \text{EvalSetConstrPattern}(\text{pat}),$
- .12 ‘*TypeConstrPattern*’ $\rightarrow \text{EvalTypeConstrPattern}(\text{pat})$

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. $\text{EvalPatternId} : \text{PatternId} \rightarrow \text{LPatEval}$

135.1 $\text{EvalPatternId}(\text{MkTag}(\text{'PatternId'}, \text{id})) \triangleq$

- .2 { $\lambda \text{val}.\lambda \text{env}.$ if *id* = 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

136. $\text{EvalMatchVal} : \text{MatchVal} \rightarrow \text{LPatEval}$

136.1 $\text{EvalMatchVal}(\text{MkTag}(\text{'MatchVal'}, \text{expr})) \triangleq$
 .2 { $\lambda \text{val}. \lambda \text{env}. \text{if } \text{val} = \perp \vee \text{eval}(\text{env}) = \perp$
 .3 then err
 .4 else if $\text{eval}(\text{env}) = \text{val}$
 .5 then { \mapsto }
 .6 else unmatch
 .7 | $\text{eval} \in \text{EvalExpr}(\text{expr})$ }

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. $\text{EvalSeqEnumPattern} : \text{SeqEnumPattern} \rightarrow \text{LPatEval}$

137.1 $\text{EvalSeqEnumPattern}(\text{MkTag}(\text{'SeqEnumPattern'}, \text{pat_l})) \triangleq$
 .2 $\text{EvalPatternList}(\text{pat_l})$

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. $\text{EvalSeqConcPattern} : \text{SeqConcPattern} \rightarrow \text{LPatEval}$

138.1 $\text{EvalSeqConcPattern}(\text{MkTag}(\text{'SeqConcPattern'}, (\text{p}_1, \text{p}_2))) \triangleq$
 .2 let *mul_pev_s* = { $\lambda \text{val}. \lambda \text{env}. \text{if } \text{val} \in \text{LIST_VAL}$
 .3 then let *val_p_s* = *ChopSeqVal*(*val*) in
 .4 { let *venv_s* = { *pev_1(v_1)*(*env*),
 .5 *pev_2(v_2)*(*env*) } in
 .6 if { err, unmatch } $\cap \text{venv_s} \neq \{ \}$
 .7 then if err $\in \text{venv_s}$
 .8 then err
 .9 else unmatch
 .10 else if *Compatible*(*venv_s*)
 .11 then *Merge*(*venv_s*)
 .12 else unmatch
 .13 | $(v_1, v_2) \in \text{val_p_s} \setminus \{ \text{unmatch} \}$
 .14 else if *val* = \perp
 .15 then { *EvalBotPat*($\{ p_1, p_2 \}$) }
 .16 else { unmatch }
 .17 | $\text{pev}_1 \in \text{EvalPattern}(p_1), \text{pev}_2 \in \text{EvalPattern}(p_2)$ } 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:
 $\text{IP}(\text{VAL} \rightarrow \text{ENV} \rightarrow \text{IP}(\text{VEnv} \cup \{ \text{err} \})))$).
 .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.

- .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 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

139. $EvalTuplePattern : TuplePattern \rightarrow LPatEval$

139.1 $EvalTuplePattern(MkTag('TuplePattern', fields)) \triangleq$

```

.2 {  $\lambda val. \lambda env. \text{if } val \in TUPLE\_VAL$ 
.3      $\text{then let } MkTag('tuple', l) = val \text{ in}$ 
.4          $pev(MkTag('seq', l))(env)$ 
.5      $\text{else if } val = \perp$ 
.6          $\text{then } EvalBotPat(\underline{\text{elems}}(fields))$ 
.7          $\text{else } \underline{\text{unmatch}}$ 
.8      $| pev \in EvalPatternList(fields) \}$ 

```

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. $EvalRecordPattern : RecordPattern \rightarrow LPatEval$

140.1 $EvalRecordPattern(MkTag('RecordPattern', (id, fields))) \triangleq$

```

.2 {  $\lambda val. \lambda env. \text{if } val \in RECORD\_VAL$ 
.3      $\text{then let } MkTag('record', MkTag(id, l)) = val \text{ in}$ 
.4          $\text{if } id \neq id'$ 
.5          $\text{then } \underline{\text{unmatch}}$ 
.6          $\text{else } pev(MkTag('seq', l))(env)$ 
.7      $\text{else if } val = \perp$ 
.8          $\text{then } EvalBotPat(\underline{\text{elems}}(fields))$ 
.9          $\text{else } \underline{\text{unmatch}}$ 
.10     $| pev \in EvalPatternList(fields) \}$ 

```

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.

.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 ∈ LL(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 ∈ IM(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: $\text{IP}(\text{VAL} \rightarrow \text{ENV} \rightarrow \text{IP}(\text{VEnv} \cup \{\underline{\text{err}}\}))$).
- .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

142. *EvalSetUnionPattern* : *SetUnionPattern* \rightarrow *LPatEval*
142.1 *EvalSetUnionPattern*(*MkTag*('SetUnionPattern', (*p*₁, *p*₂))) \triangleq
.2 let *mul-pev-s* = { $\lambda val.\lambda env.$ if *val* \in SET_VAL
.3 then let *pos-val-p-s* = *ChopSetVal*(*val*) in
.4 { let *venv-s* = { *pev*₁(*s*₁)(*env*), *pev*₂(*s*₂)(*env*) } in
.5 if { err, unmatch } \cap *venv-s* \neq { }
.6 then if err \in *venv-s*
.7 then err
.8 else unmatch
.9 else if *Compatible*(*venv-s*)
.10 then *Merge*(*venv-s*)
.11 else unmatch
.12 | (*s*₁, *s*₂) \in *pos-val-p-s* } \ { unmatch }
.13 else if *val* = \perp
.14 then { *EvalBotPat*({ *p*₁, *p*₂ }) }
.15 else { unmatch }
.16 | *pev*₁ \in *EvalPattern*(*p*₁), *pev*₂ \in *EvalPattern*(*p*₂) } 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:
 $IP(VAL \rightarrow ENV \rightarrow IP(VEnv \cup \{\underline{err}\}))$).
.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* \rightarrow *LPatEval*
143.1 *EvalSetConstrPattern*(*MkTag*('SetConstrPattern', (*pat*, *set*))) \triangleq
.2 { $\lambda val.\lambda env.$ let *v* = *ev*(*env*) in
.3 if *v* \in SET_VAL
.4 then let *s* = *StripSetTag Val*(*v*) in
.5 if *val* \in (*s* \cup { \perp })
.6 then *pev*(*val*)(*env*)
.7 else unmatch
.8 else err
.9 | *ev* \in *EvalExpr*(*set*), *pev* \in *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 .

End of annotations

144. $EvalTypeConstrPattern : TypeConstrPattern \rightarrow LPatEval$
- 144.1 $EvalTypeConstrPattern(MkTag('TypeConstrPattern', (pat, type))) \triangleq$
.2 $\{ \lambda val. \lambda env. \text{let } d = EvalType(type)(env) \text{ in}$
.3 $\quad \text{if } d = \underline{\text{err}} \vee \neg CheckTagEnv(env, type)$
.4 $\quad \text{then } \underline{\text{err}}$
.5 $\quad \text{else if } val \in (\|d\| \cup \{\perp\})$
.6 $\quad \text{then } pev(val)(env)$
.7 $\quad \text{else } \underline{\text{unmatch}}$
.8 $\quad | pev \in EvalPattern(pat) \}$

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

Auxiliary Functions

145. $ChopSetVal : SET_VAL \rightarrow \mathbb{F}(SET_VAL \times SET_VAL)$
- 145.1 $ChopSetVal(MkTag('set', elem_s)) \triangleq$
.2 $\{ (MkTag('set', el_s_1), MkTag('set', el_s_2)) \mid el_s_1, el_s_2 \in SET_VAL \cdot$
.3 $\quad el_s_1 \cup el_s_2 = elem_s \}$

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 $ChopSeqVal(MkTag('seq', l)) \triangleq$
.2 $\{ (MkTag('seq', l_1), MkTag('seq', l_2)) \mid l_1, l_2 \in LIST_VAL \cdot \underline{\text{join}}(l_1, l_2) = l \}$

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

147. $\text{EvalPatternList} : \text{IL}(\text{Pattern}) \rightarrow \text{LPatEval}$

147.1 $\text{EvalPatternList}(\text{pat.l}) \triangleq$

```

.2   let  $\text{lpev.l} = [\text{EvalPattern}(\text{pat.l}(i)) \mid i \in \text{inds}(\text{pat.l})]$  in
.3     let  $\text{pev.l.s} = \{l \mid l \in \text{IL}(\text{PatEval}) \cdot \underline{\text{len}}(l) = \underline{\text{len}}(\text{pat.l}) \wedge$ 
.4        $\forall i \in \text{inds}(\text{pat.l}) \cdot$ 
.5        $l(i) \in \text{lpev.l}(i)\}$  in
.6      $\{\lambda \text{val}. \lambda \text{env}. \text{if } \text{val} \in \text{LIST\_VAL}$ 
.7       then let  $\text{l} = \text{StripSeqTagVal}(\text{val})$  in
.8         if  $\underline{\text{len}}(\text{l}) \neq \underline{\text{len}}(\text{pat.l})$ 
.9           then unmatch
.10          else if  $\underline{\text{len}}(\text{pat.l}) = 0$ 
.11            then  $\{\mapsto\}$ 
.12            else let  $\text{venv.s} = \{\text{pev.l}(i)(l(i))(\text{env})$ 
.13               $\mid i \in \text{inds}(\text{pat.l})\}$  in
.14                if  $\{\text{err}, \text{unmatch}\} \cap \text{venv.s} \neq \{\}$ 
.15                  then if  $\text{err} \in \text{venv.s}$ 
.16                    then err
.17                    else unmatch
.18                  else if  $\text{Compatible}(\text{venv.s})$ 
.19                    then Merge( $\text{venv.s}$ )
.20                    else unmatch
.21                  else if  $\text{val} = \perp$ 
.22                    then EvalBotPat( $\underline{\text{elems}}(\text{pat.l})$ )
.23                    else unmatch
.24                 |  $\text{pev.l} \in \text{pev.l.s}\}$ 

```

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

148. $\text{EvalBotPat} : \text{IF}(\text{Pattern}) \rightarrow \text{ENV}$

148.1 $\text{EvalBotPat}(p_s) \triangleq$

.2 $\text{Merge}(\{ \text{cases } p :$

.3 $\text{MkTag}(\text{'PatternId}', id) \rightarrow \{ id \mapsto \perp \},$

.4 $\text{MkTag}(\text{'MatchVal}', _) \rightarrow \text{err},$

.5 $\text{MkTag}(\text{'SeqEnumPattern}', l) \rightarrow \text{EvalBotPat}(\underline{\text{elems}}(l)),$

.6 $\text{MkTag}(\text{'SeqConcPattern}', (p_1, p_2)) \rightarrow \text{EvalBotPat}(\{ p_1, p_2 \}),$

.7 $\text{MkTag}(\text{'RecordPattern}', (_, l)) \rightarrow \text{EvalBotPat}(\underline{\text{elems}}(l)),$

.8 $\text{MkTag}(\text{'TuplePattern}', l) \rightarrow \text{EvalBotPat}(\underline{\text{elems}}(l)),$

.9 $\text{MkTag}(\text{'SetEnumPattern}', s) \rightarrow \text{EvalBotPat}(s),$

.10 $\text{MkTag}(\text{'SeqUnionPattern}', (p_1, p_2)) \rightarrow \text{EvalBotPat}(\{ p_1, p_2 \})$

.11 $| p \in p_s \})$

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. $\text{PropToLPatEval} : \text{IP}(\text{VAL} \rightarrow \text{ENV} \rightarrow \text{IP}(\text{ENV})) \rightarrow \text{LPatEval}$

149.1 $\text{PropToLPatEval}(\text{mul_pev_s}) \triangleq$

.2 $\{ \text{pev} \in \text{PatEval} \mid \exists mpev \in \text{mul_pev_s} .$

.3 $\forall \text{val} \in \text{VAL} .$

.4 $\forall \text{env} \in \text{ENV} . \text{if } mpev(\text{val})(\text{env}) = \{ \}$

.5 $\text{then pev}(\text{val})(\text{env}) = \underline{\text{unmatch}}$

.6 $\text{else pev}(\text{val})(\text{env}) \in mpev(\text{val})(\text{env}) \}$

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. $\text{NonErrPat} : \text{Pattern} \times \text{VAL} \times \text{ENV} \rightarrow \text{IP}(\text{ENV})$

150.1 $\text{NonErrPat}(\text{pat}, \text{val}, \text{env}) \triangleq$

.2 $\{ \underline{\text{overwrite}}(\text{env}, \text{venv})$

.3 $| \text{venv} \in \{ \text{pev}(\text{val})(\text{env}) \mid \text{pev} \in \text{EvalPattern}(\text{pat}) \} .$

.4 $\text{venv} \neq \text{err} \wedge \text{venv} \neq \underline{\text{unmatch}} \wedge \neg \text{BotEnv}(\text{venv}) \}$

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

5.6.2 Bindings

```

151.   EvalBind : Bind → LBindEval
151.1  EvalBind(bind) ≡
      .2  cases ShowTag(bind) :
          .3  ‘SetBind’ → let MkTag(‘SetBind’, (pat, set)) = bind in
          .4    {  $\lambda \text{env}.$ 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    {  $\lambda \text{env}.$ let d = EvalType(type)(env) in
          .12      if d = err ∨  $\neg \text{CheckTagEnv}(env, type)$ 
          .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

5.7 Auxiliary Functions and Predicates

This section presents all the auxiliary functions and predicates used in the definition of the dynamic semantics. The functions and predicates are grouped into subsubsections according to their kind.

5.7.1 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 added¹³: This applies to selector, constructor, and modifier-functions. The model must also be enlarged with a function to check the constancy¹⁴ of the state inside implicit operations. This is done in the function *ExpandImplStateDef*.

¹³Here 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.

¹⁴This function must ensure that all read state variables are unchanged.

152. $\text{ExpandImplDests} : \text{IE}(\text{TypeDef}) \times (\text{StateDef} \cup \{\text{nil}\}) \rightarrow \text{ENV} \rightarrow \text{ENV} \cup \{\text{err}\}$

152.1 $\text{ExpandImplDests}(td_m, sd)(env) \triangleq$
 .2 $\text{let } impl_ty_env = \text{ExpandImplRecTypeDests}(td_m)(env) \text{ in}$
 .3 $\text{if } impl_ty_env = \text{err}$
 .4 $\text{then } \text{err}$
 .5 $\text{else } \text{ExpandImplStateDef}(sd)(impl_ty_env)$

Annotations to *ExpandImplDests*:

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 *ExpandImplRecTypeDests*.
- .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. $\text{ExpandImplRecTypeDests} : \text{IE}(\text{TypeDef}) \rightarrow \text{ENV} \rightarrow \text{ENV} \cup \{\text{err}\}$

153.1 $\text{ExpandImplRecTypeDests}(td_m)(env) \triangleq$
 .2 $\text{let } rec_m_s = \{ \text{ColRecs}(\text{SelShape}(td_m(id))) \mid id \in \text{dom}(td_m) \} \text{ in}$
 .3 $\text{if } \text{err} \in rec_m_s \vee \neg \text{Compatible}(rec_m_s)$
 .4 $\text{then } \text{err}$
 .5 $\text{else let } rec_def_m = \text{Merge}(rec_m_s) \text{ in}$
 .6 $\text{let } rec_d_m = \{ id \mapsto \text{EvalCompositeType}(rec_def_m(id))(env)$
 .7 $\mid id \in \text{dom}(rec_def_m) \} \text{ in}$
 .8 $\text{if } \text{err} \in \text{rng}(rec_d_m)$
 .9 $\text{then } \text{err}$
 .10 $\text{else let } rec_m = \{ id \mapsto (rec_def_m(id), rec_d_m(id))$
 .11 $\mid id \in \text{dom}(rec_def_m) \} \text{ in}$
 .12 $\text{let } ty_sel_m = \text{CompFieldSelectors}(rec_m),$
 .13 $mk_cons_m = \text{CompRecordConstructors}(rec_m),$
 .14 $rec_mod_m = \text{CompRecordModifiers}(rec_m),$
 .15 $tagenv = \{ \text{MkTag}('TagEnvId', ()) \mapsto rec_def_m \} \text{ in}$
 .16 $\text{if } ty_sel_m = \text{err}$
 .17 $\text{then } \text{err}$
 .18 $\text{else } \text{overwrite}(env, \text{Merge}(\{ ty_sel_m, mk_cons_m, rec_mod_m, tagenv \}))$

Annotations to *ExpandImplRecTypeDests*:

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*¹⁵ or if the maps are not compatible¹⁶, 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.
- .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* : $(StateDef \cup \{\underline{nil}\}) \rightarrow ENV \rightarrow ENV \cup \{\underline{err}\}$
- 154.1 *ExpandImplStateDef*(*sd*)(*env*) \triangleq
 - .2 if *sd* = *nil*
 - .3 then *env*
 - .4 else overwrite(*env*, { *MkTag*('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

¹⁵An 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*.

¹⁶If the same tag identifier is mapped to different composite types in different maps.

5.7.2 Functions for Extending the Environment

155. $\text{ExtImplOpEnv} : \text{Id} \times \text{VAL} \times (\text{Id} \cup \{\underline{\text{nil}}\}) \times (\text{VAL} \cup \{\underline{\text{nil}}\}) \times (\text{VAL} \cup \{\underline{\text{nil}}\}) \times (\text{VAL} \cup \{\underline{\text{nil}}\}) \times (\text{VAL} \cup \{\underline{\text{nil}}\}) \times \text{IF}(\text{ExtVarInf}) \rightarrow \text{ENV} \rightarrow (\text{ENV} \cup \{\underline{\text{err}}\} \times \text{ENV} \cup \{\underline{\text{err}}\})$

155.1 $\text{ExtImplOpEnv}(\text{inp_id}, \text{inp_v}, \text{out_id}, \text{out_v}, \text{inp_s}, \text{out_s}, \text{exts})(\text{env}) \triangleq$

.2 let $\text{st_id} = \text{GetStateTypeId}(\text{env})$,

.3 $\text{inp_m} = \{ \text{inp_id} \mapsto \text{inp_v} \}$,

.4 $\text{out_m} = \text{if } \text{out_id} = \underline{\text{nil}}$
then { \mapsto }
else { $\text{out_id} \mapsto \text{out_v}$ } in

.5 if ($(\text{inp_id} = \text{out_id}) \wedge (\text{inp_v} \neq \text{out_v})$) \vee
 $((\text{st_id} = \underline{\text{nostate}}) \wedge (\text{inp_s} \neq \underline{\text{nil}} \vee \text{out_s} \neq \underline{\text{nil}}))$

.6 then $\underline{\text{err}}$

.7 else if $\text{st_id} = \underline{\text{nostate}}$
then ($\underline{\text{overwrite}}(\text{env}, \text{inp_m}), \underline{\text{overwrite}}(\text{env}, \underline{\text{merge}}(\text{inp_m}, \text{out_m}))$)

.8 else let $\text{sel_m} = \{ \text{MkTag}(\text{'ValueId'}, \text{id}) \mapsto \text{MkTag}(\text{'RecSelId'}, (\text{st_id}, \text{id}))$
| $\text{MkTag}(\text{'ValueId'}, \text{id}) \in \underline{\text{elems}}(\text{CollIdList}(\text{exts})) \}$,

.9 $\text{st_v_id} = \text{GetStateValid}(\text{CUR}, \text{st_id})$,

.10 $\text{old_st_v_id} = \text{GetStateValid}(\text{OLD}, \text{st_id})$ in
if $\exists \text{sel_id} \in \underline{\text{rng}}(\text{sel_m}) \cdot \text{sel_id} \notin \underline{\text{dom}}(\text{env})$

.11 then $\underline{\text{err}}$

.12 else let $\text{pre_st_m} = \{ \text{id} \mapsto \text{env}(\text{sel_m}(\text{id}))(\text{inp_s}) \mid \text{id} \in \underline{\text{dom}}(\text{sel_m}) \} \cup$
{ $\text{st_v_id} \mapsto \text{inp_s}$ },

.13 $\text{post_old_st_m} = \{ \text{MkTag}(\text{'OldId'}, \text{id}) \mapsto \text{env}(\text{sel_m}(\text{id}))(\text{inp_s})$
| $\text{MkTag}(\text{'ValueId'}, \text{id}) \in \underline{\text{dom}}(\text{sel_m}) \} \cup$
{ $\text{old_st_v_id} \mapsto \text{inp_s}$ },

.14 $\text{post_new_st_m} = \{ \text{id} \mapsto \text{env}(\text{sel_m}(\text{id}))(\text{out_s}) \mid \text{id} \in \underline{\text{dom}}(\text{sel_m}) \} \text{ in}$

.15 let $\text{pre_env} = \underline{\text{merge}}(\text{inp_m}, \text{pre_st_m})$,

.16 $\text{post_env} = \underline{\text{Merge}}(\{ \text{inp_m}, \text{post_old_m}, \text{post_new_m}, \text{out_m} \})$ in
($\underline{\text{overwrite}}(\text{env}, \text{pre_env}), \underline{\text{overwrite}}(\text{env}, \text{post_env})$)

.17

.18

.19

.20

.21

.22

.23

.24

.25

.26

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

156. $\text{ExtExplOpEnv} : Id \times VAL \times (VAL \cup \{\underline{\text{nil}}\}) \times (VAL \cup \{\underline{\text{nil}}\}) \times \text{IF}(\text{ExtVarInf}) \rightarrow ENV \rightarrow (ENV \cup \{\underline{\text{err}}\} \times ENV \cup \{\underline{\text{err}}\})$

156.1 $\text{ExtExplOpEnv}(inp_id, inp_v, inp_s, out_s, exts)(env) \triangleq$
 .2 let $st_id = \text{GetStateTypeId}(env)$,
 .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 $sel_m = \{MkTag('ValueId', id) \mapsto MkTag('RecSelId', (st_id, id))$
 .9 $| MkTag('ValueId', id) \in \underline{\text{elems}}(\text{CollIdList}(exts))\}$,
 .10 $st_v_id = \text{GetStateValId}(\text{CUR}, st_id)$ 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. $\text{ExtImplFnsEnv} : \text{IE}(\text{ImplFnDef}) \rightarrow ENV \rightarrow (\text{IP}(\text{IE}(VEnv)) \cup \{\underline{\text{err}}\})$

157.1 $\text{ExtImplFnsEnv}(ifd_m)(env) \triangleq$
 .2 let $id_s = \underline{\text{dom}}(ifd_m)$ in
 .3 if $\exists id \in id_s \cdot \text{ExtImplFnEnv}(ifd_m(id))(env) = \underline{\text{err}}$
 .4 then $\underline{\text{err}}$
 .5 else let $mul_venv_m = \{id \mapsto \text{ExtImplFnEnv}(ifd_m(id))(env)$
 .6 $| id \in id_s\}$ in
 .7 $\{m \mid m \in \text{IE}(VEnv) \cdot \underline{\text{dom}}(m) = id_s \wedge$
 .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

```

158.   ExtImplFnEnv : ImplFnDef → ENV → (IP(VEnv) ∪ {err})
158.1  ExtImplFnEnv(MkTag('ImplFnDef', (h, _, _))(env)) ≡
.2    let MkTag('ImplFnHeading', (MkTag('Par', (id, t)), _, _)) = h in
.3    let d = EvalType(t)(env) in
.4    if d = err ∨ CheckTagEnv(env, t)
.5    then err
.6    else { {id ↦ v} | v ∈ ||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 → ENV → IL(DOM) → (IP(VEnv) ∪ {err})
159.1  ExtImplPolyFnEnv(MkTag('ImplPolyFnDef', (tp_l, h, _, _))(env)(d_l)) ≡
.2    let MkTag('ImplFnHeading', (MkTag('par', (id, t)), _, _)) = h,
.3    env' = ExtDomEnv(tp_l, d_l) in
.4    if env' = err
.5    then err
.6    else let d = EvalType(t)(overwrite(env, env')) in
.7      if d = err ∨ CheckTagEnv(env, t)
.8      then err
.9      else { {id ↦ v} | v ∈ ||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 : IL(Id) × IL(DOM) → (ENV ∪ {err})
160.1  ExtDomEnv(id_l, d_l) ≡
.2    if (len(id_l) ≠ len(d_l)) ∨ (len(id_l) ≠ card(elems(id_l)))
.3    then err
.4    else if id_l = []
.5    then { ↦ }
.6    else { id_l(i) ↦ d_l(i) | i ∈ 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

161. $\text{ExtValEnv} : \text{IL}(Id) \times \text{IL}(VAL) \rightarrow (ENV \cup \{\underline{\text{err}}\})$
- 161.1 $\text{ExtValEnv}(id_l, v_l) \triangleq$
- .2 $\text{if } (\underline{\text{len}}(id_l) \neq \underline{\text{len}}(v_l)) \vee (\underline{\text{len}}(id_l) \neq \underline{\text{card}}(\underline{\text{elems}}(id_l)))$
- .3 $\text{then } \underline{\text{err}}$
- .4 $\text{else if } id_l = []$
- .5 $\text{then } \{\mapsto\}$
- .6 $\text{else } \{ id_l(i) \mapsto v_l(i) \mid i \in \underline{\text{inds}}(id_l) \}$

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. $\text{ExtLocDomEnv} : Id \times DOM \rightarrow ENV \rightarrow ENV$
- 162.1 $\text{ExtLocDomEnv}(id, dom)(env) \triangleq$
- .2 $\text{let } teid = \text{MkTag}('TypeEnvId', ()),$
- .3 $map = \{ id \mapsto dom \} \text{ in }$
- .4 $\text{if } teid \in \underline{\text{dom}}(env)$
- .5 $\text{then let } te = \underline{\text{overwrite}}(env(teid), map) \text{ in }$
- .6 $\underline{\text{overwrite}}(env, \{ teid \mapsto te \})$
- .7 $\text{else } \underline{\text{overwrite}}(env, map)$

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

5.7.3 Functions and Predicates to deal with the State

```

163.   AssertStateConstancy : ENV → (IF(ExtVarInf) → (VAL ∪ {nil} × VAL ∪ {nil})) → IB
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 } ⊆  $\delta_0(\text{env}(id))$ 
.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 $\text{IF}(\text{ExtVarInf}) \rightarrow (\text{VAL} \cup \{\underline{\text{nil}}\} \times \text{VAL} \cup \{\underline{\text{nil}}\}) \rightarrow \text{IB}$ is also called *STATE-FUN* in the definition of the extended semantic domains.

End of annotations

```

164.   CheckStateConstancy : ENV → IF(ExtVarInf) → (VAL ∪ {nil} × VAL ∪ {nil}) → IB
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 can be seen in line 9 of the definition above.

This Curried has the type $\text{IF}(\text{ExtVarInf}) \rightarrow (\text{VAL} \cup \{\underline{\text{nil}}\} \times \text{VAL} \cup \{\underline{\text{nil}}\}) \rightarrow \text{IB}$ which also is called *STATE_FUN* in the definition of the extended semantic domains.

End of annotations

165. $\text{CheckTagEnv} : \text{Pred}(\text{ENV} \times (\text{Type} \cup \{\underline{\text{nil}}\}))$

165.1 $\text{CheckTagEnv}(env, type) \triangleq$

.2 $\text{let } tenvid = \text{MkTag}('TagEnvId', ()) \text{ in}$

.3 $\text{if } tenvid \in \underline{\text{dom}}(\text{ENV}) \wedge \text{env}(tenvid) \in \text{IE}(\text{CompositeType})$

.4 $\text{then if } type = \underline{\text{nil}}$

.5 then T

.6 $\text{else let } def_m = \text{env}(tenvid),$

.7 $\text{com_tp_m} = \text{ColRecs}(type) \text{ in}$

.8 $\underline{\text{dom}}(\text{com_tp_m}) \subseteq \underline{\text{dom}}(def_m) \wedge \text{Compatible}(\{\text{com_tp_m}, def_m\})$

.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. $\text{Restore} : \text{ENV} \times \text{IF}(\text{Id}) \times (\text{ENV} \times \text{MODE} \times (\text{VAL} \cup \{\underline{\text{nil}}\})) \rightarrow (\text{ENV} \times \text{MODE} \times (\text{VAL} \cup \{\underline{\text{nil}}\}))$

166.1 $\text{Restore}(oldenv, ids, (env, mode, val)) \triangleq$

.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

167. $\text{ResLocId} : \text{ENV} \times \text{Id} \times (\text{ENV} \times \text{MODE} \times (\text{VAL} \cup \{\underline{\text{nil}}\}))$
 $\rightarrow (\text{ENV} \times \text{MODE} \times (\text{VAL} \cup \{\underline{\text{nil}}\}))$

167.1 $\text{ResLocId}(\text{oldenv}, \text{id}, (\text{env}, \text{mode}, \text{val})) \triangleq$
.2 $\text{let } \text{teid} = \text{MkTag}(\text{'TypeEnvId'}, ()) \text{ in}$
.3 $\text{if } \text{teid} \notin \underline{\text{dom}}(\text{env}) \vee \text{env}(\text{teid}) \notin \text{LOC_ENV} \vee \text{id} \notin \underline{\text{dom}}(\text{env}(\text{teid}))$
.4 $\text{then } (\text{env}, \text{ret}, \perp)$
.5 $\text{else if } \text{teid} \notin \underline{\text{dom}}(\text{oldenv}) \vee \text{oldenv}(\text{teid}) \notin \text{LOC_ENV}$
.6 $\text{then if } \text{id} \notin \underline{\text{dom}}(\text{oldenv})$
.7 $\text{then } (\underline{\text{restrict}}(\text{env}, \{\text{id}, \text{teid}\}), \text{mode}, \text{val})$
.8 $\text{else let } \text{oldmap} = \{\text{id} \mapsto \text{oldenv}(\text{id})\} \text{ in}$
.9 $(\underline{\text{overwrite}}(\underline{\text{restrict}}(\text{env}, \{\text{teid}\}), \text{oldmap}), \text{mode}, \text{val})$
.10 $\text{else if } \text{id} \notin \underline{\text{dom}}(\text{oldenv}(\text{teid}))$
.11 $\text{then let } \text{te} = \underline{\text{subtract}}(\text{env}(\text{teid}), \{\text{id}\}) \text{ in}$
.12 $\text{if } \text{id} \notin \underline{\text{dom}}(\text{oldenv})$
.13 $\text{then } (\underline{\text{overwrite}}(\underline{\text{restrict}}(\text{env}, \{\text{id}\}), \{\text{teid} \mapsto \text{te}\}), \text{mode}, \text{val})$
.14 $\text{else let } \text{oldmap} = \underline{\text{merge}}(\{\text{id} \mapsto \text{oldenv}(\text{id})\}, \{\text{teid} \mapsto \text{te}\}) \text{ in}$
.15 $(\underline{\text{overwrite}}(\text{env}, \text{oldmap}), \text{mode}, \text{val})$
.16 $\text{else let } \text{te} = \underline{\text{overwrite}}(\text{env}(\text{teid}), \{\text{id} \mapsto \text{oldenv}(\text{teid})(\text{id})\}) \text{ in}$
.17 $\text{if } \text{id} \notin \underline{\text{dom}}(\text{oldenv})$
.18 $\text{then } (\underline{\text{overwrite}}(\underline{\text{restrict}}(\text{env}, \{\text{id}\}), \{\text{teid} \mapsto \text{te}\}), \text{mode}, \text{val})$
.19 $\text{else let } \text{oldmap} = \underline{\text{merge}}(\{\text{teid} \mapsto \text{te}\}, \{\text{id} \mapsto \text{oldenv}(\text{id})\}) \text{ in}$
.20 $(\underline{\text{overwrite}}(\text{env}, \text{oldmap}), \text{mode}, \text{val})$

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

5.7.4 Functions and Predicates to deal with Curried Functions

168. $\text{EqCurDoms} : \text{Pred}(\mathbb{N} \times \text{VAL} \times \text{VAL})$

168.1 $\text{EqCurDoms}(n, v_1, v_2) \triangleq$
.2 $\text{if } n = 0$
.3 $\text{then } \top$
.4 $\text{else if } \text{IsCurFunVal}(n, v_1) \wedge \text{IsCurFunVal}(n, v_2)$
.5 $\text{then let } \text{MkTag}(\text{'fun'}, f_1) = v_1,$
.6 $\text{MkTag}(\text{'fun'}, f_2) = v_2 \text{ in}$
.7 $\text{if } \delta_0(f_1) = \delta_0(f_2)$
.8 $\text{then } \forall e \in \delta_0(f_1) \cdot \text{EqCurDoms}(n - 1, f_1(e), f_2(e))$
.9 $\text{else } \mathbf{F}$
.10 $\text{else } \mathbf{F}$

Annotations to *EqCurDoms*:

168. This predicate checks whether the two values, v_1 and v_2 , have equal domains in n Curried levels.
- .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. $IsCurFunVal : \text{Pred}(\mathbb{N} \times \text{VAL})$

169.1 $IsCurFunVal(n, val) \triangleq$

- .2 if $n = 0$
- .3 then T
- .4 else if $val \in \text{FUN_VAL}$
- .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. $ApplyCurFn : \text{VAL} \times \mathbb{L}(\text{VAL}) \rightarrow \text{VAL}$

170.1 $ApplyCurFn(v, v_l) \triangleq$

- .2 if $v_l = []$
- .3 then v
- .4 else if $v \in \text{FUN_VAL}$
- .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

5.7.5 Compute Functions

171. $CompFieldSelectors : \text{IE}(\text{CompositeType} \times \text{RECORD_DOM}) \rightarrow \text{IE}(\text{RECORD_VAL} \rightarrow \text{VAL}) \cup \{\underline{\text{err}}\}$

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 $\underline{\text{Merge}}(\{\text{CompRecordSel}(rec_m(id)) \mid id \in \underline{\text{dom}}(rec_m)\})$

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 $RECORD_VAL \rightarrow VAL$ is also called SEL_FUN in the definition of the extended semantic domains.

End of annotations

172. $CompRecordSel : CompositeType \times RECORD_DOM \rightarrow \mathbb{E}(RECORD_VAL \rightarrow VAL) \cup \{\underline{err}\}$
- 172.1 $CompRecordSel(MkTag('CompositeType', (id, f_l)), den) \triangleq$
 .2 let $sel_m = ColSelMap(f_l)$ in
 .3 if $sel_m = \underline{err}$
 .4 then \underline{err}
 .5 else { $GetRecSelId(id, sel) \mapsto GenSelectorFn(den, sel_m, sel)$ | $sel \in \underline{dom}(sel_m)$ }

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 $RECORD_VAL \rightarrow VAL$ is also called SEL_FUN in the definition of the extended semantic domains.

End of annotations

173. $CompRecordConstructors : \mathbb{E}(CompositeType \times RECORD_DOM) \rightarrow \mathbb{E}(\mathbb{I}(VAL) \rightarrow RECORD_VAL)$
- 173.1 $CompRecordConstructors(rec_m) \triangleq$
 .2 Merge({ $GetRecConsId(id) \mapsto GenConstructorFn(rec_m(id))$ | $id \in \underline{dom}(rec_m)$ })

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 $\mathbb{I}(VAL) \rightarrow RECORD_VAL$ is also called $CONS_FUN$ in the definition of the extended semantic domains.

End of annotations

174. $CompRecordModifiers : \mathbb{E}(CompositeType \times RECORD_DOM) \rightarrow \mathbb{E}(\mathbb{E}(VAL) \rightarrow RECORD_VAL \rightarrow VAL)$
- 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 | $id \in \underline{dom}(rec_m)$ }

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 $\mathbb{E}(VAL) \rightarrow RECORD_VAL \rightarrow VAL$ is also called MOD_FUN in the definition of the extended semantic domains.

End of annotations

5.7.6 Generate Functions

175. $\text{GenRecordModFn} : \text{Id} \times \text{IE}(\text{RECORD_VAL} \rightarrow \text{VAL}) \times \text{RECORD_DOM} \rightarrow (\text{IE}(\text{VAL}) \rightarrow \text{RECORD_VAL} \rightarrow \text{VAL})$

175.1 $\text{GenRecordModFn}(\text{tag}, \text{sel_m}, \text{den}) \triangleq$
 .2 $\lambda \text{mod_m} \in \text{IE}(\text{VAL}) .$
 .3 $\text{let } \text{mod_m}' = \{ \text{GetRecSelId}(\text{tag}, \text{mod_id}) \mapsto \text{mod_m}(\text{mod_id})$
 .4 $|\text{mod_id} \in \underline{\text{dom}}(\text{mod_m}) \} \text{ in}$
 .5 $\text{if } \underline{\text{dom}}(\text{mod_m}') \subseteq \underline{\text{dom}}(\text{sel_m})$
 .6 $\text{then let } \text{mod_ids} = \underline{\text{dom}}(\text{mod_m}'),$
 .7 $\text{const_ids} = \underline{\text{dom}}(\text{sel_m}) \setminus \underline{\text{dom}}(\text{mod_m}') \text{ in}$
 .8 $\lambda \text{val} \in \text{RECORD_VAL} .$
 .9 $\text{if } \text{val} \in \|\text{den}\|$
 .10 $\text{then } \text{Iota}(\{ \text{val}' | \text{val}' \in \|\text{den}\| \cdot \forall \text{id} \in \text{mod_ids} .$
 .11 $\text{sel_m}(\text{id})(\text{val}') = \text{mod_m}'(\text{id}) \wedge$
 .12 $\forall \text{id} \in \text{const_ids} .$
 .13 $\text{sel_m}(\text{id})(\text{val}) = \text{sel_m}(\text{id})(\text{val}') \})$
 .14 $\text{else } \perp$
 .15 $\text{else } \lambda \text{val} \in \text{RECORD_VAL} . \perp$

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 value¹⁷.
- .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. $\text{GenSelectorFn} : \text{RECORD_DOM} \times \text{IE}(\text{IN}) \times \text{Id} \rightarrow (\text{RECORD_VAL} \rightarrow \text{VAL})$

176.1 $\text{GenSelectorFn}(\text{den}, \text{sel_m}, \text{sel}) \triangleq$
 .2 $\lambda \text{val} \in \text{RECORD_VAL} . \text{if } \text{val} \in \|\text{den}\| \wedge \text{sel} \in \underline{\text{dom}}(\text{sel_m})$
 .3 $\text{then let } \text{val_l} = \text{StripRecTagVal}(\text{val}) \text{ in}$
 .4 $\text{if } \text{sel_m}(\text{sel}) \in \underline{\text{inds}}(\text{val_l})$
 .5 $\text{then } \text{val_l}(\text{sel_m}(\text{sel}))$
 .6 $\text{else } \perp$
 .7 $\text{else } \perp$

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

¹⁷This domain is also called *MOD_FUN* in the definition of the extended semantic domains.

177. $\text{GenConstructorFn} : \text{CompositeType} \times \text{RECORD_DOM} \rightarrow (\mathbb{L}(\text{VAL}) \rightarrow \text{RECORD_VAL}_\perp)$

177.1 $\text{GenConstructorFn}(\text{type}, \text{dom}) \triangleq$
 .2 $\text{let } v_l.s = \{ \text{StripRecTagVal}(\text{rec}) \mid \text{rec} \in \|\text{dom}\| \},$
 .3 $\text{MkTag}(\text{'CompositeType}', (\text{tag}, _)) = \text{type} \text{ in}$
 .4 $\lambda v.l \in \mathbb{L}(\text{VAL}). \text{if } v.l \in v.l.s$
 .5 $\text{then } \text{MkTag}(\text{'record'}, \text{MkTag}(\text{tag}, v.l))$
 .6 $\text{else } \perp$

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. $\text{GenPolyApply} : \text{POLYVAL} \times \mathbb{L}(\text{DOM}) \times \text{VAL} \rightarrow \text{VAL}$

178.1 $\text{GenPolyApply}(\text{poly}, d.l, \text{val}) \triangleq$
 .2 $\text{let } f = \text{poly}(d.l) \text{ in}$
 .3 $\text{if } f \in \text{FUN_VAL}$
 .4 $\text{then let } \text{MkTag}(\text{'fun'}, fn) = f \text{ in}$
 .5 $\text{if } \text{val} \in \delta_0(fn)$
 .6 $\text{then } fn(\text{val})$
 .7 $\text{else } \perp$
 .8 $\text{else } \perp$

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

5.7.7 Make Functions

179. $\text{MakePolyFnDef} : \mathbb{L}_1(\text{TypeVar}) \times \text{Lambda} \rightarrow \text{LPEval}$

179.1 $\text{MakePolyFnDef}(t.l, \text{body}) \triangleq$
 .2 $\{ \lambda \text{env} . \lambda d.l . \text{if } (\underline{\text{len}}(d.l) \neq \underline{\text{len}}(t.l)) \vee (\underline{\text{len}}(t.l) \neq \underline{\text{card}}(\underline{\text{elems}}(t.l)))$
 .3 $\text{then } \perp$
 .4 $\text{else } ev(\underline{\text{overwrite}}(\text{env}, \text{ExtDomEnv}(t.l, d.l)))$
 .5 $| ev \in \text{EvalExpr}(\text{body}) \}$

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

180. $\text{MakeFuncAbs} : Id \times Type \times Type \times Expr \rightarrow ENV \rightarrow \text{IP}(VAL)$

180.1 $\text{MakeFuncAbs}(id, domtp, rngtp, expr)(env) \triangleq$

.2 let $d_1 = \text{EvalType}(domtp)(env)$,
 .3 $d_2 = \text{EvalType}(rngtp)(env)$,
 .4 $lev = \text{EvalExpr}(expr)$ in
 .5 if $d_1 = \underline{\text{err}} \vee d_2 = \underline{\text{err}}$
 .6 then $\{\perp\}$
 .7 else let $fn_s = \{\lambda par \in \|d_1\|. ev(\underline{\text{overwrite}}(env, \{id \mapsto par\}))$
 .8 | $ev \in lev\}$ in
 .9 { if $\forall a \in \|d_1\| \cdot fn(a) \in \|d_2\|$
 .10 then $MkTag('fun', fn)$
 .11 else \perp
 .12 | $fn \in fn_s\}$

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. $\text{MakeEvalMapSet} : \text{IE}(ImplFnDef) \cup \text{IE}(ImplPolyFnDef) \rightarrow$
 $(Id \times (ImplFnDef \cup ImplPolyDef) \rightarrow Expr) \rightarrow$
 $\text{IP}(\text{IE}(EEval))$

181.1 $\text{MakeEvalMapSet}(ifd_m)(f) \triangleq$

.2 let $id_s = \underline{\text{dom}}(ifd_m)$ in
 .3 let $lev_m = \{id \mapsto \text{EvalExpr}(f(id, ifd_m(id))) \mid id \in id_s\}$ in
 .4 $\{m \mid m \in \text{IE}(EEval) \cdot \underline{\text{dom}}(m) = id_s \wedge \forall id \in id_s \cdot m(id) \in lev_m(id)\}$

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. $\text{MakeFns} : \text{IE}(ImplFnDef) \rightarrow ENV \rightarrow (\text{IP}(\text{IE}(VAL)) \cup \{\underline{\text{err}}\})$

182.1 $\text{MakeFns}(implfndef_m)(env) \triangleq$

.2 let $id_s = \underline{\text{dom}}(implfndef_m)$ in
 .3 let $tp_m = \{id \mapsto \text{GetType}(implfndef_m(id)) \mid id \in id_s\}$ in
 .4 let $d_m = \text{EvalTypeMap}(tp_m)(env)$ in
 .5 if $d_m = \underline{\text{err}} \vee \exists tp \in \text{rng}(tp_m) \cdot \neg \text{CheckTagEnv}(env, tp)$
 .6 then $\underline{\text{err}}$
 .7 else $\{m \mid m \in \text{IE}(VAL) \cdot \underline{\text{dom}}(m) = id_s \wedge$
 .8 $\forall id \in id_s \cdot m(id) \in \|d_m(id)\| \wedge$
 .9 $IsFunCont(m(id), d_m(id))\}$

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. $\text{MakePolyFns} : \text{IE}(\text{polyImplFnDef}) \rightarrow \text{ENV} \rightarrow (\text{IP}(\text{IE}(\text{POLYVAL})) \cup \{\text{err}\})$
- 183.1 $\text{MakePolyFns}(\text{implfndef}_m)(\text{env}) \triangleq$
- .2 $\text{let } id_s = \text{dom}(\text{implfndef}_m) \text{ in}$
- .3 $\text{let } tp_m = \{id \mapsto \text{GetType}(\text{implfndef}_m(id)) \mid id \in id_s\},$
- .4 $\text{var_m} = \{id \mapsto \text{ColTypeVars}(\text{implfndef}_m(id)) \mid id \in id_s\} \text{ in}$
- .5 $\text{let } d_m = \{id \mapsto \text{EvalPolyType}(\text{var_m}(id), tp_m(id))(\text{env}) \mid id \in id_s\} \text{ in}$
- .6 $\text{if } \exists tp \in \text{rng}(tp_m) \cdot \neg \text{CheckTagEnv}(\text{env}, tp)$
- .7 then err
- .8 $\text{else } \{m \mid m \in \text{IE}(\text{POLYVAL}) \cdot \text{dom}(m) = id_s \wedge$
- .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 $\text{IsFunCont}(m(id)(d_l), d_m(id)(d_l))\}$

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

5.7.8 Get Functions

184. $\text{GetFnApp} : Id \times \text{ImplFnDef} \rightarrow \text{Expr}$
- 184.1 $\text{GetFnApp}(\text{fnid}, \text{MkTag}(\text{'ImplFnDef}', (\text{heading}, _, _))) \triangleq$
- .2 $\text{let } \text{MkTag}(\text{'ImplFnHeading'}, (\text{MkTag}(\text{'Par'}, (\text{argid}, _)), _, _)) = \text{heading} \text{ in}$
- .3 $\text{MkTag}(\text{'Apply'}, (\text{fnid}, \text{argid}))$

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

185. $\text{GetType} : \text{ImplFnDef} \cup \text{ImplPolyFnDef} \cup \text{ExplFnDef} \cup \text{ExplPolyFnDef} \rightarrow \text{Type}$

185.1 $\text{GetType}(\text{def}) \triangleq$
 .2 cases $\text{ShowTag}(\text{def})$:
 .3 ‘ ImplFnDef ’ $\rightarrow \text{let } \text{MkTag}(\text{'ImplFnDef}', (\text{h}, _, _)) = \text{def} \text{ in}$
 .4 $\text{let } \text{MkTag}(\text{'ImplFnHeading}', (\text{par}, \text{r_t}, _)) = \text{h} \text{ in}$
 .5 $\text{let } \text{MkTag}(\text{'Par}', (_, \text{d_t})) \text{ in}$
 .6 $\text{MkTag}(\text{'FnType}', (\text{d_t}, \text{r_t})),$
 .7 ‘ ImplPolyFnDef ’ $\rightarrow \text{let } \text{MkTag}(\text{'ImplPolyFnDef}', (_, \text{h}, _, _)) = \text{def} \text{ in}$
 .8 $\text{let } \text{MkTag}(\text{'ImplFnHeading}', (\text{par}, \text{r_t}, _)) = \text{h} \text{ in}$
 .9 $\text{let } \text{MkTag}(\text{'Par}', (_, \text{d_t})) \text{ in}$
 .10 $\text{MkTag}(\text{'FnType}', (\text{d_t}, \text{r_t})),$
 .11 ‘ ExplFnDef ’ $\rightarrow \text{let } \text{MkTag}(\text{'ExplFnDef}', (\text{d_t}, \text{r_t}, _, _, _)) = \text{def} \text{ in}$
 .12 $\text{MkTag}(\text{'FnType}', (\text{d_t}, \text{r_t})),$
 .13 ‘ ExplPolyFnDef ’ $\rightarrow \text{let } \text{MkTag}(\text{'ExplPolyFnDef}', (_, \text{tp}, _, _, _)) = \text{def} \text{ in}$
 .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. $\text{GetExpr} : \text{Pattern} \rightarrow \text{Expr}$

186.1 $\text{GetExpr}(\text{pat}) \triangleq$
 .2 cases pat :
 .3 $\text{MkTag}(\text{'PatternId}', \text{id}) \rightarrow \text{id},$
 .4 $\text{MkTag}(\text{'MatchVal}', \text{expr}) \rightarrow \text{expr},$
 .5 $\text{MkTag}(\text{'SeqEnumPattern}', \text{p_l}) \rightarrow \text{let } \text{e_l} = \text{ApplySeq}(\text{GetExpr})(\text{p_l}) \text{ in}$
 .6 $\text{MkTag}(\text{'SeqEnumeration}', \text{e_l}),$
 .7 $\text{MkTag}(\text{'SeqConcPattern}', (\text{p}_1, \text{p}_2)) \rightarrow \text{let } \text{e}_1 = \text{GetExpr}(\text{p}_1),$
 .8 $\text{e}_2 = \text{GetExpr}(\text{p}_2) \text{ in}$
 .9 $\text{let } \text{e} = (\text{e}_1, \text{SEQCONC}, \text{e}_2) \text{ in}$
 .10 $\text{MkTag}(\text{'BinaryExpr}', \text{e}),$
 .11 $\text{MkTag}(\text{'RecordPattern}', (\text{id}, \text{p_l})) \rightarrow \text{let } \text{e_l} = \text{ApplySeq}(\text{GetExpr})(\text{p_l}) \text{ in}$
 .12 $\text{let } \text{e} = (\text{id}, \text{e_l}) \text{ in}$
 .13 $\text{MkTag}(\text{'RecordConstructor}', \text{e}),$
 .14 $\text{MkTag}(\text{'TuplePattern}', \text{p_l}) \rightarrow \text{let } \text{e_l} = \text{ApplySeq}(\text{GetExpr})(\text{p_l}) \text{ in}$
 .15 $\text{MkTag}(\text{'TupleConstructor}', \text{e_l}),$
 .16 $\text{MkTag}(\text{'SetUnionPattern}', (\text{p}_1, \text{p}_2)) \rightarrow \text{let } \text{e}_1 = \text{GetExpr}(\text{p}_1),$
 .17 $\text{e}_2 = \text{GetExpr}(\text{p}_2) \text{ in}$
 .18 $\text{let } \text{e} = (\text{e}_1, \text{SETUNION}, \text{e}_2) \text{ in}$
 .19 $\text{MkTag}(\text{'BinaryExpr}', \text{e}),$
 .20 $\text{MkTag}(\text{'SetEnumPattern}', \text{p_s}) \rightarrow \text{let } \text{e_s} = \{ \text{GetExpr}(\text{p}) \mid \text{p} \in \text{p_s} \} \text{ in}$
 .21 $\text{MkTag}(\text{'SetEnumeration}', \text{e_s}),$
 .22 $\text{MkTag}(\text{'SetConstrPattern}', (\text{p}, _)) \rightarrow \text{GetExpr}(\text{p}),$
 .23 $\text{MkTag}(\text{'TypeConstrPattern}', (\text{p}, _)) \rightarrow \text{GetExpr}(\text{p})$

Annotations to GetExpr :

186. This auxiliary function takes a syntactic pattern and yields the corresponding syntactic expression.

End of annotations

187. $\text{GetStateDom} : \text{ENV} \rightarrow \text{DOM}$

187.1 $\text{GetStateDom}(\text{env}) \triangleq$
 .2 let $\text{st_id} = \text{GetStateTypeId}(\text{env})$ in
 .3 if $\text{st_id} = \text{nostate}$
 .4 then $(\emptyset_{\perp}, \{ \})$
 .5 else if $\text{env}(\text{st_id}) \in \text{DOM}$
 .6 then $\text{env}(\text{st_id})$
 .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. $\text{GetStateTypeId} : \text{ENV} \rightarrow \text{StateTypeId} \cup \{ \text{nostate} \}$

188.1 $\text{GetStateTypeId}(\text{env}) \triangleq$
 .2 if $\exists ! \text{id} \in \underline{\text{dom}}(\text{env}) \cdot \text{ShowTag}(\text{id}) = \text{'StateTypeId'} \wedge \text{env}(\text{id}) \in \text{DOM}$
 .3 then $\text{Iota}(\{ \text{id} \mid \text{id} \in \underline{\text{dom}}(\text{env}) \cdot \text{ShowTag}(\text{id}) = \text{'StateTypeId'} \})$
 .4 else 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. $\text{GetStateValId} : \{ \text{CUR, OLD} \} \times \text{StateTypeId} \rightarrow \text{ValueId} \cup \text{OldId}$

189.1 $\text{GetStateValId}(\text{mode}, \text{MkTag}(\text{'StateTypeId'}, \text{st_id})) \triangleq$
 .2 if $\text{mode} = \text{CUR}$
 .3 then $\text{MkTag}(\text{'ValueId'}, \text{st_id})$
 .4 else $\text{MkTag}(\text{'OldId'}, \text{st_id})$

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. $\text{GetRecSelId} : \text{Id} \times \text{Id} \rightarrow \text{RecSelId}$

190.1 $\text{GetRecSelId}(\text{tag}, \text{field}) \triangleq$
 .2 $\text{MkTag}(\text{'RecSelId'}, (\text{StripTag}(\text{tag}), \text{StripTag}(\text{field})))$

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. $\text{GetRecModId} : \text{ValueId} \rightarrow \text{RecModId}$

191.1 $\text{GetRecModId}(\text{MkTag}(\text{'ValueId'}, \text{tag})) \triangleq$
 .2 $\text{MkTag}(\text{'RecModId'}, \text{tag})$

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. $\text{GetRecConsId} : \text{ValueId} \rightarrow \text{RecConsId}$

- 192.1 $\text{GetRecConsId}(\text{MkTag}(\text{'ValueId'}, \text{tag})) \triangleq$
.2 $\text{MkTag}(\text{'RecConsId'}, \text{tag})$

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

5.7.9 Collector Functions

```

193.   ColRecs : Type → (IE(CompositeType) ∪ {err})
193.1  ColRecs(type) ≡
       .2 cases type :
       .3   MkTag('CompositeType', (id, f-l)) → let ty-s = elems(ColTypeList(f-l)) in
       .4     let r-s = { ColRecs(t) | t ∈ ty-s } in
       .5       if err ∉ r-s ∧
       .6         Compatible(r-s ∪ { { id ↦ type } })
       .7         then Merge(r-s ∪ { { id ↦ type } })
       .8         else err,
       .9   MkTag('ProductType', t-l) → let r-s = { ColRecs(t) | t ∈ elems(t-l) } in
      .10     if err ∉ r-s ∧ Compatible(r-s)
      .11       then Merge(r-s)
      .12       else err
      .13   MkTag('UnionType', t-s) → let r-s = { ColRecs(t) | t ∈ t-s } in
      .14     if err ∉ r-s ∧ Compatible(r-s)
      .15       then Merge(r-s)
      .16       else err
      .17   MkTag('OptionalType', t) → ColRecs(t),
      .18   MkTag('SetType', t) → ColRecs(t),
      .19   MkTag('Seq0Type', t) → ColRecs(t),
      .20   MkTag('Seq1Type', t) → ColRecs(t),
      .21   MkTag('GeneralMapType', (t1, t2)) → let r-s = { ColRecs(t1), ColRecs(t2) } in
      .22     if err ∉ r-s ∧ Compatible(r-s)
      .23       then Merge(r-s)
      .24       else err,
      .25   MkTag('InjectiveMapType', (t1, t2)) → let r-s = { ColRecs(t1), ColRecs(t2) } in
      .26     if err ∉ r-s ∧ Compatible(r-s)
      .27       then Merge(r-s)
      .28       else err,
      .29   MkTag('FnType', (t1, t2)) → let r-s = { ColRecs(t1), ColRecs(t2) } in
      .30     if err ∉ r-s ∧ Compatible(r-s)
      .31       then Merge(r-s)
      .32       else err,
      .33   MkTag('BasicType', _) → { ↦ },
      .34   MkTag('TypeId', _) → { ↦ },
      .35   MkTag('TypeVar', _) → { ↦ },
      .36   MkTag('QuoteType', _) → { ↦ }

```

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 : IL(Field) ∪ IL(Par) → IL(Type)
194.1  ColTypeList(f-l) ≡
       .2 if f-l = []
       .3 then []
       .4 else cases hd(f-l) :
       .5   MkTag('Field', ( _, type )) → join([ type ], ColTypeList(tl(f-l))),
       .6   MkTag('Par', ( _, type )) → join([ type ], ColTypeList(tl(f-l)))

```

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. $\text{ColIdSet} : \text{Pattern} \cup \text{Bind} \rightarrow \text{IF}(Id)$

$$195.1 \quad \text{ColIdSet}(pb) \triangleq$$

.2	cases <i>pb</i> :	
.3	<i>MkTag</i> ('PatternId', <u>nil</u>)	$\rightarrow \{ \}$,
.4	<i>MkTag</i> ('PatternId', <i>id</i>)	$\rightarrow \{ id \}$,
.5	<i>MkTag</i> ('MatchVal', -)	$\rightarrow \{ \}$,
.6	<i>MkTag</i> ('SeqEnumPattern', <i>p_l</i>)	$\rightarrow \bigcup \{ \text{ColIdSet}(p) \mid p \in \underline{\text{elems}}(p_l) \}$,
.7	<i>MkTag</i> ('SeqConcPattern', (<i>p</i> ₁ , <i>p</i> ₂))	$\rightarrow \text{ColIdSet}(p_1) \cup \text{ColIdSet}(p_2)$,
.8	<i>MkTag</i> ('RecordPattern', (-, <i>p_l</i>))	$\rightarrow \bigcup \{ \text{ColIdSet}(p) \mid p \in \underline{\text{elems}}(p_l) \}$,
.9	<i>MkTag</i> ('TuplePattern', <i>p_l</i>)	$\rightarrow \bigcup \{ \text{ColIdSet}(p) \mid p \in \underline{\text{elems}}(p_l) \}$,
.10	<i>MkTag</i> ('SetUnionPattern', (<i>p</i> ₁ , <i>p</i> ₂))	$\rightarrow \text{ColIdSet}(p_1) \cup \text{ColIdSet}(p_2)$,
.11	<i>MkTag</i> ('SetEnumPattern', <i>p_s</i>)	$\rightarrow \bigcup \{ \text{ColIdSet}(p) \mid p \in p_s \}$,
.12	<i>MkTag</i> ('SetConstrPattern', (<i>p</i> , -))	$\rightarrow \text{ColIdSet}(p)$,
.13	<i>MkTag</i> ('TypeConstrPattern', (<i>p</i> , -))	$\rightarrow \text{ColIdSet}(p)$,
.14	<i>MkTag</i> ('SetBind', (<i>p</i> , -))	$\rightarrow \text{ColIdSet}(p)$,
.15	<i>MkTag</i> ('TypeBind', (<i>p</i> , -))	$\rightarrow \text{ColIdSet}(p)$

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. $\text{ColIdList} : \text{IL}(\text{Par}) \cup \text{IL}(\text{TypeVar}) \rightarrow \text{IL}(Id)$

$$196.1 \quad \text{ColIdList}(\text{params}) \triangleq$$

.2	if <i>params</i> = []	
.3	then []	
.4	else cases <i>hd</i> (<i>params</i>) :	
.5	<i>MkTag</i> ('Par', (<i>id</i> , -))	$\rightarrow \underline{\text{join}}([id], \text{ColIdList}(\underline{\text{tl}}(\text{params})))$,
.6	<i>MkTag</i> ('TypeVar', <i>id</i>)	$\rightarrow \underline{\text{join}}([id], \text{ColIdList}(\underline{\text{tl}}(\text{params})))$

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. $\text{ColParList} : \text{ExplPolyFnDef} \cup \text{ExplFnDef} \rightarrow \text{IL}(\text{Par})$

$$197.1 \quad \text{ColParList}(\text{def}) \triangleq$$

.2	cases <i>def</i> :	
.3	<i>MkTag</i> ('ExplFnDef', (-, -, -, <i>body</i> , -))	$\rightarrow \text{ColPars}(\text{body})$,
.4	<i>MkTag</i> ('ExplPolyFnDef', (-, -, -, <i>body</i> , -))	$\rightarrow \text{ColPars}(\text{body})$

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

198. $\text{ColPars} : \text{Expr} \rightarrow \text{IL}(\text{Par})$

198.1 $\text{ColPars}(\text{expr}) \triangleq$
 .2 if $\text{ShowTag}(\text{expr}) \neq \text{'Lambda'}$
 .3 then []
 .4 else let $\text{MkTag}(\text{'Lambda}', (\text{par}, \text{body})) = \text{expr}$ in
 .5 $\underline{\text{join}}([\text{par}], \text{ColPars}(\text{body}))$

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. $\text{ColTypeVars} : \text{ExplPolyFnDef} \rightarrow \text{IL}_1(\text{Id})$

199.1 $\text{ColTypeVars}(\text{MkTag}(\text{'ExplPolyFnDef'}, (\text{tv_l}, _, _, _))) \triangleq$
 .2 $[\text{StripTag}(\text{tv_l}(i)) \mid i \in \underline{\text{inds}}(\text{tv_l})]$

Annotations to ColTypeVars :

199. This auxiliary function collects a list of identifiers from type variables from a polymorphic function definition.

End of annotations

200. $\text{ColInvFns} : \text{IE}(\text{TypeDef}) \rightarrow \text{IM}(\text{Pattern}, \text{ExplFnDef})$

200.1 $\text{ColInvFns}(\text{td_m}) \triangleq$
 .2 let $btp = \text{MkTag}(\text{'BasicType'}, \text{BOOLEAN})$,
 .3 $pre = \text{MkTag}(\text{'BoolLit'}, \text{TRUE})$ in
 .4 { let $invId = \text{MkTag}(\text{'PatternId'}, \text{SelInvId}(\text{td_m}(\text{id})))$,
 .5 $body = \text{SelInvExpr}(\text{td_m}(\text{id}))$ in
 .6 $invId \mapsto \text{MkTag}(\text{'ExplFnDef'}, (\text{SelShape}(\text{td_m}(\text{id})), btp, pre, body, \text{TOTAL}))$
 .7 | $\text{id} \in \underline{\text{dom}}(\text{td_m})$ }

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. $\text{ColSelMap} : \text{IL}(\text{Field}) \rightarrow \text{IE}(\text{IN}) \cup \{\underline{\text{err}}\}$

201.1 $\text{ColSelMap}(\text{f.l}) \triangleq$
 .2 if $\underline{\text{card}}(\underline{\text{elems}}(\text{f.l})) \neq \underline{\text{len}}(\text{f.l})$
 .3 then $\underline{\text{err}}$
 .4 else { $\text{SelSel}(\text{f.l}(i)) \mapsto i \mid i \in \underline{\text{inds}}(\text{f.l}) \cdot \text{SelSel}(\text{f.l}(i)) \neq \underline{\text{nil}}$ }

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

5.7.10 Selector Functions

202. $SelPre : ImplFnDef \cup ImplPolyFnDef \cup ExplFnDef \cup ExplPolyFnDef \rightarrow Expr$

202.1 $SelPre(def) \triangleq$
 .2 cases $ShowTag(def)$:
 .3 ' $ImplFnDef$ ' \rightarrow let $MkTag('ImplFnDef', (_, pre, _)) = def$ in
 .4 pre ,
 .5 ' $ImplPolyFnDef$ ' \rightarrow let $MkTag('ImplPolyFnDef', (_, _, pre, _)) = def$ in
 .6 pre ,
 .7 ' $ExplFnDef$ ' \rightarrow let $MkTag('ExplFnDef', (_, _, pre, _, _)) = def$ in
 .8 pre ,
 .9 ' $ExplPolyFnDef$ ' \rightarrow 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. $SelPost : ImplFnDef \cup ImplPolyFnDef \rightarrow Expr$

203.1 $SelPost(def) \triangleq$
 .2 cases $ShowTag(def)$:
 .3 ' $ImplFnDef$ ' \rightarrow let $MkTag('ImplFnDef', (_, _, post)) = def$ in
 .4 $post$,
 .5 ' $ImplPolyFnDef$ ' \rightarrow 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. $SelRes : ImplFnDef \cup ImplPolyFnDef \rightarrow Id$

204.1 $SelRes(def) \triangleq$
 .2 cases $ShowTag(def)$:
 .3 ' $ImplFnDef$ ' \rightarrow let $MkTag('ImplFnDef', (h, _, _)) = def$ in
 .4 let $MkTag('ImplFnHeading', (_, _, resum)) = h$ in
 .5 $resum$,
 .6 ' $ImplPolyFnDef$ ' \rightarrow 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. $SelBody : ExplFnDef \cup ValDef \rightarrow Expr$

205.1 $SelBody(expldef) \triangleq$
 .2 cases $expldef$:
 .3 $MkTag('ExplFnDef', (_, _, _, body, _)) \rightarrow body$,
 .4 $MkTag('ValDef', (_, expr)) \rightarrow expr$

Annotations to *SelBody*:

205. This auxiliary function selects the body of either an explicit function definition or a value definition.

End of annotations

206. $\text{SelKind} : \text{ExplFnDef} \cup \text{ExplPolyFnDef} \rightarrow \{\text{TOTAL}, \text{PARTIAL}\}$

$$206.1 \quad \text{SelKind}(\text{explfndef}) \triangleq$$

.2 cases *explfndef*:

.3 $\text{MkTag}(\text{'ExplFnDef}', (_, _, _, _, \text{kind})) \rightarrow \text{kind}$,

.4 $\text{MkTag}(\text{'ExplPolyFnDef}', (_, _, _, _, \text{kind})) \rightarrow \text{kind}$

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. $\text{SelTypeVars} : \text{ExplPolyFnDef} \rightarrow \mathbb{L}(\text{TypeVar})$

$$207.1 \quad \text{SelTypeVars}(\text{MkTag}(\text{'ExplPolyFnDef}', (\text{tpvars}, _, _, _, _))) \triangleq$$

.2 [let $\text{MkTag}(\text{'Type Var}', \text{id}) = \text{tpvars}(i)$ in $\text{id} \mid i \in \text{inds}(\text{tpvars})$]

Annotations to *SelTypeVars*:

207. This auxiliary function selects the list of type variables from a polymorphic explicit function definition.

End of annotations

208. $\text{SelPolyFn} : \text{ExplPolyFnDef} \rightarrow \mathbb{L}_1(\text{TypeVar}) \times \text{Lambda}$

$$208.1 \quad \text{SelPolyFn}(\text{MkTag}(\text{'ExplPolyFnDef}', (\text{tpvars}, _, _, \text{lambda}, _))) \triangleq$$

.2 $(\text{tpvars}, \text{lambda})$

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. $\text{SelShape} : \text{TypeDef} \rightarrow \text{Type}$

$$209.1 \quad \text{SelShape}(\text{MkTag}(\text{'TypeDef}', (\text{shape}, _))) \triangleq$$

.2 shape

Annotations to *SelShape*:

209. This auxiliary function selects the shape (i.e. a type) of a type definition.

End of annotations

210. $\text{SelInvId} : \text{TypeDef} \rightarrow \text{Id}$

$$210.1 \quad \text{SelInvId}(\text{MkTag}(\text{'TypeDef}', (_, (\text{id}, _)))) \triangleq$$

.2 id

Annotations to *SelInvId*:

210. This auxiliary function selects the invariant identifier from a type definition.

End of annotations

211. $\text{SelInvExpr} : \text{TypeDef} \rightarrow \text{Lambda}$
 211.1 $\text{SelInvExpr}(\text{MkTag}(\text{'TypeDef}', (_, (_, \lambda)))) \triangleq$
 .2 λ

Annotations to SelInvExpr :

211. This auxiliary function selects the invariant expression from a type definition.
End of annotations

212. $\text{SelSel} : \text{Field} \rightarrow \text{Id} \cup \{\underline{\text{nil}}\}$
 212.1 $\text{SelSel}(\text{MkTag}(\text{'Field}', (\text{sel}, _))) \triangleq$
 .2 if $\text{sel} = \underline{\text{nil}}$
 .3 then $\underline{\text{nil}}$
 .4 else sel

Annotations to SelSel :

212. This auxiliary function selects the selector identifier from Field , if present.
End of annotations

213. $\text{SelExtVarInfMode} : \text{ExtVarInf} \rightarrow \{\text{READ}, \text{READWRITE}\}$
 213.1 $\text{SelExtVarInfMode}(\text{MkTag}(\text{'ExtVarInf}', (_, _, \text{mode}))) \triangleq$
 .2 mode

Annotations to SelExtVarInfMode :

213. This auxiliary function selects the mode from a ExtVarInf .
End of annotations

214. $\text{SelExtVarInfId} : \text{ExtVarInf} \rightarrow \text{token}$
 214.1 $\text{SelExtVarInfId}(\text{MkTag}(\text{'ExtVarInf}', (_, \text{MkTag}(_, \text{token}), _))) \triangleq$
 .2 token

Annotations to SelExtVarInfId :

214. This auxiliary function selects the token of the identifier from ExtVarInf .
End of annotations

215. $\text{SelPattern} : \text{Bind} \rightarrow \text{Pattern}$
 215.1 $\text{SelPattern}(\text{bind}) \triangleq$
 .2 cases bind :
 .3 $\text{MkTag}(\text{'TypeBind}', (\text{p}, _)) \rightarrow \text{p}$,
 .4 $\text{MkTag}(\text{'SetBind}', (\text{p}, _)) \rightarrow \text{p}$

Annotations to SelPattern :

215. This auxiliary function selects the pattern of a binding.
End of annotations

5.7.11 Tag-processing Functions

216. $\text{MkTag} : \text{Id} \times \text{VAL} \rightarrow \text{VAL}$
 216. MkTag must satisfy :
 (1) $\forall id \in \text{Id}, val \in \text{VAL} \setminus \{\perp\} \cdot \|\mathcal{T}^{id}(\{val\}_{\perp}, \{val\})\| = \{\text{MkTag}(id, val)\}$

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* : $VAL \rightarrow Id$

- 217.1 $ShowTag(MkTag(id, _)) \triangleq$
.2 id

Annotations to *ShowTag*:

217. Get the tag.

End of annotations

218. *StripTag* : $VAL \rightarrow VAL$

- 218.1 $StripTag(MkTag(., v)) \triangleq$
.2 v

Annotations to *StripTag*:

218. Strip the tag from the value.

End of annotations

219. *StripRecTagVal* : $VAL \rightarrow \mathbb{L}(VAL) \cup \{\underline{err}\}$

- 219.1 $StripRecTagVal(v) \triangleq$
.2 if $ShowTag(v) = 'record'$
.3 then let $MkTag('record', l) = StripTag(v)$ in
.4 l
.5 else 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

220. *ShowRecTagVal* : $VAL \rightarrow Id \cup \{\underline{err}\}$

- 220.1 $ShowRecTagVal(v) \triangleq$
.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. $\text{StripSeqTagVal} : \text{VAL} \rightarrow \text{IL}(\text{VAL}) \cup \{\underline{\text{err}}\}$
221.1 $\text{StripSeqTagVal}(v) \triangleq$
.2 if $\text{ShowTag}(v) = \text{'seq'}$
.3 then let $\text{MkTag}(\text{'seq'}, l) = \text{StripTag}(v)$ in
.4 l
.5 else err

Annotations to *StripSeqTagVal*:

221. Strip the ‘seq’ tag.
End of annotations

222. $\text{StripSetTagVal} : \text{VAL} \rightarrow \text{IF}(\text{VAL}) \cup \{\underline{\text{err}}\}$
222.1 $\text{StripSetTagVal}(v) \triangleq$
.2 if $\text{ShowTag}(v) = \text{'set'}$
.3 then let $\text{MkTag}(\text{'set'}, s) = \text{StripTag}(v)$ in
.4 s
.5 else err

Annotations to *StripSetTagVal*:

222. Strip the ‘set’ tag.
End of annotations

223. $\text{StripMapTagVal} : \text{VAL} \rightarrow \text{IM}(\text{VAL}, \text{VAL}) \cup \{\underline{\text{err}}\}$
223.1 $\text{StripMapTagVal}(v) \triangleq$
.2 if $\text{ShowTag}(v) = \text{'map'}$
.3 then let $\text{MkTag}(\text{'map'}, m) = \text{StripTag}(v)$ in
.4 m
.5 else err

Annotations to *StripMapTagVal*:

223. Strip the ‘map’ tag.
End of annotations

224. $\text{StripNumTagVal} : \text{VAL} \rightarrow \text{IR} \cup \{\underline{\text{err}}\}$
224.1 $\text{StripNumTagVal}(v) \triangleq$
.2 if $\text{ShowTag}(v) = \text{'num'}$
.3 then let $\text{MkTag}(\text{'num'}, n) = \text{StripTag}(v)$ in
.4 n
.5 else err

Annotations to *StripNumTagVal*:

224. Strip the ‘num’ tag.
End of annotations

5.7.12 General Functions and Predicates

Most of the functions and predicates in this subsubsection 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 chapter about the domain universe.

225. $\text{Injective} : \text{Pred}(\text{IM}(X, Y))$

225.1 $\text{Injective}(m) \triangleq$

$$.2 \quad \forall x_1, x_2 \in \underline{\text{dom}}(m) \cdot x_1 \neq x_2 \Rightarrow m(x_1) \neq m(x_2)$$

Annotations to Injective:

225. This predicate checks whether a map is injective, i.e. that all domain elements are mapped to different range elements.

End of annotations

226. $\text{IsFunCont} : \text{Pred}(\text{FUN_VAL} \times \text{FUN_DOM})$

226.1 $\text{IsFunCont}(\text{MkTag}('fun', f), d) \triangleq$

$$.2 \quad (\forall x_1, x_2 \in \delta_0(f) \cdot x_1 \sqsubseteq_{\delta_0(d)} x_2 \Rightarrow f(x_1) \sqsubseteq_{\delta_1(d)} f(x_2)) \wedge$$

$$.3 \quad (\forall \{x_i \mid i \in \omega\} \subseteq \delta_0(f) \cdot \forall i \in \omega \cdot x_i \sqsubseteq_{\delta_0(d)} x_{i+1} \Rightarrow$$

$$.4 \quad f(\bigsqcup_{\delta_0(d)} \{x_i \mid i \in \omega\}) = \bigsqcup_{\delta_1(d)} \{f(x_i) \mid i \in \omega\})$$

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. $\text{IsCont}_C : \text{Pred}(C \rightarrow C)$

227.1 $\text{IsCont}_C(f) \triangleq$

$$.2 \quad (\forall x_1, x_2 \in |C| \cdot x_1 \sqsubseteq_C x_2 \Rightarrow f(x_1) \sqsubseteq_C f(x_2)) \wedge$$

$$.3 \quad (\forall \{x_i \mid i \in \omega\} \subseteq |C| \cdot \forall i \in \omega \cdot x_i \sqsubseteq_C x_{i+1} \Rightarrow$$

$$.4 \quad f(\bigsqcup_C \{x_i \mid i \in \omega\}) = \bigsqcup_C \{f(x_i) \mid i \in \omega\})$$

Annotations to IsCont_C:

227. The predicate IsCont_C verifies a given cpo C if a function $f : C \rightarrow C$ is continuous with respect to the ordering \sqsubseteq_C .

End of annotations

228. $\text{IsCpo} : \text{Pred}(\text{QUASIDOM})$

228.1 $\text{IsCpo}((|A|, \sqsubseteq_A), \|A\|) \triangleq$

$$.2 \quad \forall x \in |A| \cdot x \sqsubseteq_A x \wedge$$

$$.3 \quad \forall x, y \in |A| \cdot x \sqsubseteq_A y \wedge y \sqsubseteq_A x \Rightarrow x = y \wedge$$

$$.4 \quad \forall x, y, z \in |A| \cdot x \sqsubseteq_A y \wedge y \sqsubseteq_A z \Rightarrow x \sqsubseteq_A z \wedge$$

$$.5 \quad \exists \text{bot} \in |A| \cdot \forall x \in |A| \cdot \text{bot} \sqsubseteq_A x \wedge$$

$$.6 \quad \forall f \in \text{IN} \rightarrow A \cdot (\forall i, j \in \text{IN} \cdot i \leq j \Rightarrow f(i) \sqsubseteq_A f(j)) \Rightarrow$$

$$.7 \quad \exists \text{lub} \in |A| \cdot (\forall i \in \text{IN} \cdot f(i) \sqsubseteq_A \text{lub}) \wedge$$

$$.8 \quad (\forall u \in |A| \cdot (\forall i \in \text{IN} \cdot f(i) \sqsubseteq_A u) \Rightarrow \text{lub} \sqsubseteq_A u)$$

Annotations to IsCpo:

228. This predicate checks whether a domain from QUASIDOM forms a CPO (see definition 4.1.7 in chapter 4). 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 4.1.1 in chapter 4.

- .5 Here it is checked that there exists a least element in the CPO (see also definition 4.1.5 in chapter 4).
- .6–8 Here it is checked that there exists a least upper bound for each countable chain in the domain (see definition 4.1.6 in chapter 4).

End of annotations

229. *Reverse* : $\mathbb{L}(X) \rightarrow \mathbb{L}(X)$
- 229.1 $\text{Reverse}(l) \triangleq$
- .2 if $l = []$
 - .3 then $[]$
 - .4 else join($\text{Reverse}(\underline{\text{tl}}(l))$, [hd(l)])

Annotations to Reverse:

229. This is an auxiliary function that simply reverses a sequence.

End of annotations

230. *ApplySeq* : $(X \rightarrow Y) \rightarrow \mathbb{L}(X) \rightarrow \mathbb{L}(Y)$
- 230.1 $\text{ApplySeq}(\text{func})(\text{seq}) \triangleq$
- .2 if $\text{seq} = []$
 - .3 then $[]$
 - .4 else join([$\text{func}(\underline{\text{hd}}(\text{seq}))$], $\text{ApplySeq}(\text{func})(\underline{\text{tl}}(\text{seq}))$)

Annotations to ApplySeq:

230. The result of applying $\text{ApplySeq}(\text{func})$ 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* : $\mathbb{F}(X) \rightarrow X$
- 231.1 $\text{Iota}(x) \triangleq$
- .2 if card(x) = 1
 - .3 then let $\{v\} = x$ in
 - .4 v
 - .5 else \perp

Annotations to Iota:

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* : $\mathbb{P}(X) \rightarrow X$
232. *Choose* must satisfy :
- (1) $\forall s \in \mathbb{P}(X) \cdot s \neq \{\} \Rightarrow \text{Choose}(s) \in s$

Annotations to Choose:

232. This auxiliary function selects an arbitrary element from a non-empty arbitrary set.

End of annotations

233. $\text{BotEnv} : \text{Pred}(ENV)$

233.1 $\text{BotEnv}(env) \triangleq$
.2 $\perp \in \text{rng}(env)$

Annotations to BotEnv :

233. This function is an auxiliary predicate which checks whether any bindings occur to the bottom value.

End of annotations

234. $\text{SetToSeq} : \text{IF}(X) \rightarrow \text{IF}(\text{L}(X))$

234.1 $\text{SetToSeq}(set) \triangleq$
.2 $\text{if } set = \{ \}$
.3 $\text{then } \{ [] \}$
.4 $\text{else } \{ \text{join}([e], rest) \mid e \in set, rest \in \text{SetToSeq}(set \setminus \{ e \}) \}$

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. $\text{StepsToSeq} : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \text{L}(NUM_VAL) \cup \{ \text{err} \}$

235.1 $\text{StepsToSeq}(lb, ub, step) \triangleq$
.2 $\text{if } step = 0$
.3 then err
.4 $\text{else if } ((lb < ub) \wedge (step < 0)) \vee ((lb > ub) \wedge (step > 0))$
.5 $\text{then } []$
.6 $\text{else join}([MkTag('num', lb)], \text{StepsToSeq}(lb + step, ub, step))$

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

236. $\text{NonDetIter} : \text{IP}(X) \times (X \times X \rightarrow X) \rightarrow \text{IP}(X)$

236.1 $\text{NonDetIter}(x_s, comp) \triangleq$
.2 $\text{if } \forall x \in x_s, y, z \in X \cdot y \sqsubseteq_X z \Rightarrow comp(x, y) \sqsubseteq_X comp(x, z)$
.3 $\text{then let } h = \lambda l \in \text{L}_\omega(x_s). \text{if } l \in \text{L}(x_s)$
.4 $\text{then if } \underline{\text{len}}(l) = 0$
.5 $\text{then } \perp$
.6 $\text{else } comp(\underline{\text{hd}}(l), h(\underline{\text{tl}}(l)))$
.7 $\text{else } \bigsqcup_X \{ h(\text{ApprList}(l)(n)) \mid n \in \mathbb{N} \} \text{ in}$
.8 $\{ h(l) \mid l \in \text{L}_\omega(x_s) \}$
.9 $\text{else } \{ \}$

Annotations to NonDetIter :

236. This function, given a subset x_s of X and a function $comp : X \times X \rightarrow X$ 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. $ApprList : \mathbb{L}_\omega(X) \rightarrow \mathbb{N} \rightarrow \mathbb{L}(X)$
- 237.1 $ApprList(l) \triangleq$
- .2 $\lambda n. \text{if } n = 0$
 - .3 $\quad \text{then } []$
 - .4 $\quad \text{else } \underline{\text{join}}([\underline{\text{hd}}(l)], ApprList(\underline{\text{tl}}(l))(n - 1))$

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. $PropErr : \mathbb{P}(X \rightarrow \mathbb{P}(Y)) \rightarrow \mathbb{P}(X \rightarrow (Y \cup \{\underline{\text{err}}\}))$
- 238.1 $PropErr(mul_ldef_s) \triangleq$
- .2 $\{ \text{def} \mid \text{def} \in (X \rightarrow Y \cup \{\underline{\text{err}}\}) \cdot$
 - .3 $\quad \exists m_ldef \in mul_ldef_s \cdot \forall x \in X \cdot \text{if } m_ldef(x) = \{ \}$
 - .4 $\quad \quad \quad \text{then } \text{def}(x) = \underline{\text{err}}$
 - .5 $\quad \quad \quad \text{else } \text{def}(x) \in m_ldef(x) \}$

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, $\underline{\text{err}}$ is returned. Note that all places where $PropErr$ is called and where $\underline{\text{err}}$ cannot be used, it is ensured that the sets from the codomain are non-empty.

End of annotations

239. $PropS : \mathbb{P}(X \rightarrow \mathbb{P}(Y)) \rightarrow \mathbb{P}(X \rightarrow (Y \cup (X \times \{\underline{\text{ret}}\} \times \{\perp\})))$
- 239.1 $PropS(mul_ldef_s) \triangleq$
- .2 $\{ \text{def} \mid \text{def} \in (X \rightarrow Y \cup \{\underline{\text{err}}\}) \cdot$
 - .3 $\quad \exists m_ldef \in mul_ldef_s \cdot \forall x \in X \cdot \text{if } m_ldef(x) = \{ \}$
 - .4 $\quad \quad \quad \text{then } \text{def}(x) = (x, \underline{\text{ret}}, \perp)$
 - .5 $\quad \quad \quad \text{else } \text{def}(x) \in m_ldef(x) \}$

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 \perp , 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. $PropE : \text{IP}(X \rightarrow \text{IP}(Y)) \rightarrow \text{IP}(X \rightarrow (Y \cup \{\perp\}))$
- 240.1 $PropE(mul_ldef_s) \triangleq$
 - .2 $\{ def \mid def \in (X \rightarrow Y \cup \{\text{err}\}) \cdot$
 - .3 $\exists m_ldef \in mul_ldef_s \cdot \forall x \in X \cdot \text{if } m_ldef(x) = \{ \} \text{ then } def(x) = \perp$
 - .4 $\text{else } def(x) \in m_ldef(x) \}$
 - .5

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. $Compatible : \text{Pred}(\text{IP}(\text{IM}(X, Y)))$
- 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

- 242. $Partition : \text{IP}(\text{IP}(X)) \rightarrow (\text{IP}(\text{IP}(X)) \cup \{\text{err}\})$
- 242.1 $Partition(set_s) \triangleq$
 - .2 $\text{if } \{ \} \in set_s$
 - .3 then err
 - .4 $\text{else } \{ s \mid s \in \text{IP}(X) \cdot \underline{\text{card}}(s) \leq \underline{\text{card}}(set_s) \wedge$
 - .5 $\forall set \in set_s \cdot \exists elem \in s \cdot elem \in set \wedge$
 - .6 $\underline{\text{card}}(\{ e \mid e \in s \cdot \forall s_1 \in set_s \setminus \{ set \} \cdot e \notin s_1 \}) \leq 1 \}$

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 exist 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. $\text{Permutations} : \mathbb{L}(X) \rightarrow \mathbb{F}(\mathbb{L}(X))$

243.1 $\text{Permutations}(\text{list}) \triangleq$

$$\begin{aligned} .2 \quad & \{ l \mid l \in \mathbb{L}(X) \cdot \underline{\text{len}}(\text{list}) = \underline{\text{len}}(l) \wedge \\ .3 \quad & \forall e \in \underline{\text{elems}}(l) \cdot \underline{\text{card}}(\{ j \mid j \in \underline{\text{inds}}(l) \wedge l(j) = e \}) = \\ .4 \quad & \underline{\text{card}}(\{ j \mid j \in \underline{\text{inds}}(\text{list}) \wedge \text{list}(j) = e \}) \} \end{aligned}$$

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. $\text{AllPairComb} : \mathbb{L}(\mathbb{P}(X) \times \mathbb{P}(X)) \rightarrow \mathbb{P}(\mathbb{L}(X \times X))$

244.1 $\text{AllPairComb}(\text{set_p_l}) \triangleq$

$$\begin{aligned} .2 \quad & \{ l \mid l \in \mathbb{L}(X \times X) \cdot \underline{\text{len}}(l) = \underline{\text{len}}(\text{set_p_l}) \wedge \\ .3 \quad & \forall i, j \in \underline{\text{inds}}(l) \cdot \\ .4 \quad & \text{let } (el_{11}, el_{12}) = l(i), \\ .5 \quad & (el_{21}, el_{22}) = l(j), \\ .6 \quad & (set_{11}, set_{12}) = \text{set_p_l}(i), \\ .7 \quad & (set_{21}, set_{22}) = \text{set_p_l}(j) \text{ in} \\ .8 \quad & el_{11} \in set_{11} \wedge el_{12} \in set_{12} \wedge \\ .9 \quad & el_{21} \in set_{21} \wedge el_{22} \in set_{22} \} \end{aligned}$$

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. $\text{True} : () \rightarrow \text{BOOL_VAL}$

245.1 $\text{True}() \triangleq$

$$.2 \quad \text{MkTag}(\text{'bool'}, \top)$$

Annotations to True:

245. This auxiliary function is just a short hand for the ‘truth’ element in BOOL_VAL .

End of annotations

246. $\text{False} : () \rightarrow \text{BOOL_VAL}$

246.1 $\text{False}() \triangleq$

$$.2 \quad \text{MkTag}(\text{'bool'}, \mathbf{F})$$

Annotations to False:

246. This auxiliary function is just a short hand for the ‘falsehood’ element in BOOL_VAL .

End of annotations

Chapter 6

The Mathematical Concrete Syntax and Outer Abstract Syntax

Productions of the concrete syntax are preceded by a \star . Productions of the abstract syntax are *emphasized* and preceded by a \bullet or \circ .

6.1 Document

- \star document = definition block, { definition block } ;
 - \circ $Document = DefinitionBlock^+$
-

6.2 Definitions

- \star definition block = type definitions
 - | state definition
 - | value definitions
 - | function definitions
 - | operation definitions ;
 - \circ $DefinitionBlock = TypeDefinitions$
 - | $StateDef$
 - | $ValueDefinitions$
 - | $FunctionDefinitions$
 - | $OperationDefinitions$
-

6.2.1 Type Definitions

- \star type definitions = ‘types’, type definition, { ‘;’, type definition } ;
- \bullet $TypeDefinitions :: typedefs : TypeDef^+$

- \star type definition = untagged type definition | tagged type definition ;

- $TypeDef = UnTaggedTypeDef \mid TaggedTypeDef$
-

* untagged type definition = identifier, '=', type, [invariant] ;

- $UnTaggedTypeDef ::= id : Id$
 $shape : Type$
 $typeinv : [Invariant]$
-

* tagged type definition = identifier, '::', field list, [invariant] ;

- $TaggedTypeDef ::= id : Id$
 $fields : FieldList$
 $typeinv : [Invariant]$
-

* 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 ;

- $Type = BracketedType$
| $BasicType$
| $QuoteType$
| $CompositeType$
| $UnionType$
| $ProductType$
| $OptionalType$
| $SetType$
| $SeqType$
| $MapType$
| $FnType$
| $TypeName$
| $TypeVar$
-

* bracketed type = '(', type, ')' ;

- $BracketedType ::= type : Type$
-

- * basic type = ‘B’ | ‘N’ | ‘N₁’ | ‘Z’ | ‘Q’ | ‘R’ | ‘char’ | ‘token’ ;
- *BasicType* = BOOLEAN | NAT | NATONE | INTEGER | RAT | REAL | CHAR | TOKEN

- * quote type = quote literal ;
- *QuoteType* :: *lit* : *QuoteLit*

- * composite type = ‘compose’, identifier, ‘of’, field list, ‘end’ ;
- *CompositeType* :: *id* : *Id*
 fields : *FieldList*

- * field list = { field } ;
- *FieldList* = *Field*^{*}

- * field = [identifier, ‘:’], type ;
- *Field* :: *sel* : [*Id*]
 type : *Type*

- * union type = type, ‘|’, type, { ‘|’, type } ;

UnionType :: *summands* : *Type*⁺

inv *mk-UnionType*(*ts*) \triangleq len *ts* \geq 2

- * product type = type, ‘×’, type, { ‘×’, type } ;

ProductType :: *factors* : *Type*⁺

inv *mk-ProductType*(*ts*) \triangleq len *ts* \geq 2

- * optional type = ‘[’, type, ‘]’ ;
- *OptionalType* :: *type* : *Type*

- * set type = type, ‘-set’ ;

- $\text{SetType} :: \text{elemtp} : \text{Type}$
-

* seq type = seq0 type | seq1 type ;

- $\text{SeqType} = \text{Seq0 Type} \mid \text{Seq1 Type}$
-

* seq0 type = type, '*' ;

- $\text{Seq0 Type} :: \text{elemtp} : \text{Type}$
-

* seq1 type = type, '+' ;

- $\text{Seq1 Type} :: \text{elemtp} : \text{Type}$
-

* map type = general map type | injective map type ;

- $\text{MapType} = \text{GeneralMapType} \mid \text{InjectiveMapType}$
-

* general map type = type, ' \xrightarrow{m} ', type ;

- $\text{GeneralMapType} :: \text{mapdom} : \text{Type}$
 $\text{maprng} : \text{Type}$
-

* injective map type = type, ' $\xleftarrow{x}\xrightarrow{m}$ ', type ;

- $\text{InjectiveMapType} :: \text{mapdom} : \text{Type}$
 $\text{maprng} : \text{Type}$
-

* function type = partial function type | total function type ;

- $\text{FnType} = \text{PartialFnType} \mid \text{TotalFnType}$
-

* partial function type = discretionary type, ' \rightarrow ', type ;

- $\text{PartialFnType} :: \text{fn dom} : \text{DiscretionaryType}$
 $\text{fn rng} : \text{Type}$
-

* total function type = discretionary type, ' \xrightarrow{t} ', type ;

- $TotalFnType ::= fn dom : DiscretionaryType$
 $fn rng : Type$
-

- * discretionary type = type | '(', ')' ;
 - o $DiscretionaryType = Type \mid \text{UNITTYPE}$
-
- * type name = name ;
 - o $TypeName = Name$
-

- * type variable = type variable identifier ;
 - o $TypeVar = TypeVarId$
-
- * type variable identifier = '@', identifier ;
- $TypeVarId ::= Id$
-

6.2.2 State Definition

- * state definition = 'state', identifier, 'of',
field list,
[invariant],
[initialization],
'end' ;
 - $StateDef ::= stid : Id$
 $fields : FieldList$
 $stinv : [Invariant]$
 $stinit : [Initialization]$
-

- * invariant = 'inv', invariant initial function ;

- o $Invariant = InvInitFn$
-

- * initialization = 'init', invariant initial function ;

- o $Initialization = InvInitFn$
-

- * invariant initial function = pattern, ' \triangle ', expression ;

- $InvInitFn ::= pat : Pattern$
 $expr : Expr$
-

6.2.3 Value Definitions

- * value definitions = 'values', value definition, { ';' , value definition } ;
 - *ValueDefinitions :: valdefs : ValueDef⁺*
-

- * value definition = pattern, [':', type], '=' , expression ;
 - *ValueDef :: pat : Pattern
type : [Type]
val : Expr*
-

6.2.4 Function Definitions

- * function definitions = 'functions', function definition, { ';' , function definition } ;
 - *FunctionDefinitions :: funcdefs : FunctionDef⁺*
-

- * function definition = explicit function definition | implicit function definition ;
 - *FunctionDef = ExplFnDef | ImplFnDef*
-

- * explicit function definition = identifier, [type variable list], ':', function type, identifier, parameters list,
‘△’, expression,
[‘pre’, expression] ;

- *ExplFnDef :: id : Id
tpparms : OTypeVarList
type : FnType
idrepeated : Id
parms : ParametersList
body : Expr
fnpre : [Expr]*
-

- * implicit function definition = identifier, [type variable list], parameter types, identifier type pair,
[‘pre’, expression],
‘post’, expression ;

- *ImplFnDef :: id : Id
tpparms : OTypeVarList
partps : ParameterTypes
residtype : IdType
fnpre : [Expr]
fnpost : Expr*
-

* type variable list = '[', type variable identifier, { ',', type variable identifier }, ']' ;

o *OTypeVarList* = *TypeVar**

* identifier type pair = identifier, ':', type ;

• *IdType*::*id* : *Id*
 type : *Type*

* parameter types = '(', [pattern type pair list], ')' ;

o *ParameterTypes* = *OPatTypePairList*

* pattern type pair list = pattern type pair, { ',', pattern type pair } ;

o *OPatTypePairList* = *PatTypePair**

* pattern type pair = pattern list, ':', type ;

• *PatTypePair*::*pat* : *Patlist*
 type : *Type*

* parameters list = parameters, { parameters } ;

o *ParametersList* = *Parameters*⁺

* parameters = '(', [pattern list], ')' ;

o *Parameters* = *OPatList*

6.2.5 Operation Definitions

* operation definitions = 'operations', operation definition, { ',', operation definition } ;

• *OperationDefinitions*::*opdefs* : *OperationDef*⁺

* operation definition = explicit operation definition | implicit operation definition ;

o *OperationDef* = *ExplOprtDef* | *ImplOprtDef*

* explicit operation definition = identifier, `:', operation type,
identifier, parameters,
' Δ ', statement,
['pre', expression] ;

- *ExplOprtDef* :: *id* : *Id*
optype : *OpType*
idrepeated : *Id*
params : *Parameters*
body : *Stmt*
oppre : [*Expr*]
-

* implicit operation definition = identifier, parameter types,
[identifier type pair],
[externals],
['pre', expression],
['post', expression,
[exceptions] ;

- *ImplOprtDef* :: *id* : *Id*
params : *ParameterTypes*
residtype : [*IdType*]
opext : *OExternals*
oppre : [*Expr*]
oppost : *Expr*
excps : *OExceptions*
-

* operation type = discretionary type, ' \xrightarrow{o} ', discretionary type ;

- *OpType* :: *opdom* : *DiscretionaryType*
oprng : *DiscretionaryType*
-

* externals = 'ext', var information, { var information } ;

- *OExternals* = *VarInf**
-

* var information = mode, name list, [`:', type] ;

- *VarInf* :: *mode* : *Mode*
vars : *NameList*
type : [*Type*]
-

* mode = 'rd' | 'wr' ;

- *Mode* = READ | READWRITE
-

* exceptions = 'errs', error list ;
* error list = error, { error } ;
o *OExceptions* = *Error**

* error = identifier, ':', expression, '→', expression ;
• *Error*::*id* : *Id*
 cond : *Expr*
 action : *Expr*

6.3 Expressions

* expression list = expression, { ',', expression } ;
o *ExprList* = *Expr*⁺
o *OExprList* = *Expr**

* 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 ;

- *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*
-

6.3.1 Bracketed Expressions

- * bracketed expression = ‘(’, expression, ‘)’ ;
 - *BracketedExpr* ::= *expr* : *Expr*
-

6.3.2 Local Binding Expressions

- * let expression = ‘let’, local definition { ‘;’, local definition }, ‘in’, expression ;
 - *LetExpr* ::= *localdefs* : *LocalDef*⁺
 body : *Expr*

 - * let be expression = ‘let’, bind, [‘be’, ‘st’, expression], ‘in’, expression ;
 - *LetBeSTExpr* ::= *bind* : *Bind*
 cond : [*Expr*]
 body : *Expr*
-

- * def expression = 'def', def bind, { ';' , def bind }, 'in', expression ;
 - *DefExpr*::
 defs : *DefBind*⁺
 body : *Expr*
-

- * def bind = pattern bind, '=', expression ;
 - *DefBind*::
 lhs : *PatternBind*
 rhs : *Expr*
-

6.3.3 Conditional Expressions

- * if expression = 'if', expression, 'then', expression, { elseif expression }, 'else', expression ;
 - *IfExpr*::
 test : *Expr*
 cons : *Expr*
 elsifaltn : *ElsifExpr*^{*}
 altn : *Expr*
-

- * elseif expression = 'elseif', expression, 'then', expression ;
 - *ElsifExpr*::
 test : *Expr*
 cons : *Expr*
-

- * cases expression = 'cases', expression, ':',
 cases expression alternatives,
 [';' , others expression],
 'end' ;
 - *CasesExpr*::
 sel : *Expr*
 altns : *CaseExprAlternatives*
 other : [*OthersExpr*]
-

- * cases expression alternatives = cases expression alternative, { ';' , cases expression alternative } ;
 - *CaseExprAlternatives* = *CaseAltn*⁺
-

- * cases expression alternative = pattern list, '→', expression ;
 - *CaseAltn*::
 match : *PatList*
 body : *Expr*
-

- * others expression = 'others', '→', expression ;
 - *OthersExpr* = *Expr*
-

6.3.4 Unary Expressions

* unary expression = prefix expression | map inverse expression ;

o $\text{UnaryExpr} = \text{PrefixExpr} \mid \text{MapInverseExpr}$

* prefix expression = unary operator, expression ;

• $\text{PrefixExpr} :: op : \text{UnaryOp}$
 $expr : Expr$

* 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 ;

o $\text{UnaryOp} = \text{NUMPLUS}$
| NUMMINUS
| NUMABS
| FLOOR
| NOT
| SETCARD
| SETPOWER
| SETDISTR UNION
| $\text{SETDISTR INTERSECT}$
| SEQHEAD
| SEQTAIL
| SEQLEN
| SEQELEMS
| SEQINDICES
| SEQDISTRCONC
| MAPDOM
| MAPRNG
| MAPDISTRMERGE

* unary plus = '+' ;

- * unary minus = ‘-’;
 - * arithmetic abs = ‘abs’;
 - * floor = ‘floor’;
 - * not = ‘¬’;
 - * set cardinality = ‘card’;
 - * finite power set = ‘ \mathcal{F} ’;
 - * distributed set union = ‘ \cup ’;
 - * distributed set intersection = ‘ \cap ’;
 - * 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, ‘ $^{-1}$ ’;

- *MapInverseExpr::operand : Expr*
-

6.3.5 Binary Expressions

- * binary expression = expression, binary operator, expression ;
 - *BinaryExpr::left : Expr*
 op : BinaryOp
 right : Expr
-

- * 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
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 ;

- $BinaryOp = \text{NUMPLUS}$
 - | 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
-

- * arithmetic plus = '+';
- * arithmetic minus = '-';
- * arithmetic multiplication = '×';
- * arithmetic divide = '/';
- * arithmetic integer division = 'div';
- * arithmetic rem = 'rem';
- * arithmetic mod = 'mod';
- * less than = '<';
- * less than or equal = ' \leq ';
- * greater than = '>';
- * greater than or equal = ' \geq ';

- * 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 .

6.3.6 Quantified Expressions

- * quantified expression = all expression
 - | exists expression
 - | exists unique expression ;
 - $QuantExpr = AllExpr$
 - | $ExistsExpr$
 - | $ExistsUniqueExpr$
-

* all expression = ‘ \forall ’, bind list, ‘ $,$ ’, expression ;

- *AllExpr :: bind : BindList
pred : Expr*
-

* exists expression = ‘ \exists ’, bind list, ‘ $,$ ’, expression ;

- *ExistsExpr :: bind : BindList
pred : Expr*
-

* exists unique expression = ‘ $\exists!$ ’, bind, ‘ $,$ ’, expression ;

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

6.3.7 Iota Expression

* iota expression = ‘ ι ’, bind, ‘ $,$ ’, expression ;

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

6.3.8 Set Expressions

* set enumeration = ‘{’, [expression list], ‘}’ ;

- *SetEnumeration :: els : OExprList*
-

* set comprehension = ‘{’, expression, ‘|’, bind list, [‘ $,$ ’, expression], ‘}’ ;

- *SetComprehension :: elem : Expr
bind : BindList
pred : [Expr]*
-

* set range expression = ‘{’, expression, ‘;’, ‘...’, ‘;’, expression, ‘}’ ;

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

6.3.9 Sequence Expressions

- * sequence enumeration = '[', [expression list], ']' ;
- *SeqEnumeration::els : OExprList*

- * sequence comprehension = '[', expression, '|', set bind, ['.', expression], ']' ;
- *SeqComprehension::elem : Expr*
bind : SetBind
pred : [Expr]

- * subsequence = expression, '(', expression, ',', '...', ',', expression, ')' ;
- *SubSequence::sequence : Expr*
frompos : Expr
topos : Expr

6.3.10 Map Expressions

- * map enumeration = '{', maplet, { ',', maplet }, '}' | '{', '↦', '}' ;
- *MapEnumeration::els : Maplet**

- * maplet = expression, '↦', expression ;
- *Maplet::mapdom : Expr*
maprng : Expr

- * map comprehension = '{', maplet, '|', bind list, ['.', expression], '}' ;
- *MapComprehension::elem : Maplet*
bind : BindList
pred : [Expr]

6.3.11 Tuple Constructor Expression

- * tuple constructor = 'mk', '(', expression, ',', expression list, ')' ;
- TupleConstructor::fields : ExprList*
- inv $mk\text{-}TupleConstructor(fs) \triangleq \text{len } fs \geq 2$

6.3.12 Record Expressions

* record constructor = name, '(', [expression list], ')' ;

- *RecordConstructor::tag : Id*
fields : OExprList
-

* record modifier = ' μ ', '(', expression, ',', record modification, { ',', record modification }, ')' ;

- *RecordModifier::rec : Expr*
modifiers : RecordModification⁺
-

* record modification = identifier, ' \mapsto ', expression ;

- *RecordModification::field : Id*
new : Expr
-

6.3.13 Apply Expressions

* apply = expression, '(', [expression list], ')' ;

- *Apply::fct : Expr*
arg: OExprList
-

* field select = expression, ':', identifier ;

- *FieldSelect::record : Expr*
field : Id
-

* function type instantiation = name, '[', type, { ',', type }, ']' ;

- *FctTypeInst::polyfct : Name*
inst : Type⁺
-

6.3.14 Lambda Expression

* lambda expression = ' λ ', type bind list, ',', expression ;

- *Lambda::parms : TypeBindList*
body : Expr
-

6.3.15 Is Expressions

* is expression = is defined type expression | is basic type expression ;

o $IsExpr = IsDefTypeExpr \mid IsBasicTypeExpr$

* is defined type expression = identifier, '(', expression, ')' ;

• $IsDefTypeExpr ::= deftype : Id$
 arg : Expr

* is basic type expression = is basic type, '(', expression, ')' ;

• $IsBasicTypeExpr ::= type : IsBasicType$
 arg : Expr

* is basic type = 'is-', ('B' | 'N' | 'N₁' | 'Z' | 'Q' | 'R' | 'char' | 'token') ;

o $IsBasicType = \text{BOOLEAN} \mid \text{NAT} \mid \text{NATONE} \mid \text{INTEGER} \mid \text{RAT} \mid \text{REAL} \mid \text{CHAR} \mid \text{TOKEN}$

6.3.16 Names

* name = identifier ;

• $Name ::= name : Id$

* name list = name, { ',', name } ;

o $NameList = Name^+$

* old name = identifier, ' $\overleftarrow{ }$ ' ;

• $OldName ::= name : Id$

note: An old name such as $identifier^{\leftarrow}$ can also be written as $\overleftarrow{identifier}$.

6.4 State Designators

* state designator = name
| field reference
| map or sequence reference ;

o *StateDesignator* = *Name*
| *FieldRef*
| *MapOrSeqRef*

* field reference = state designator, '.', identifier ;

• *FieldRef* :: *var* : *StateDesignator*
sel : *Id*

* map or sequence reference = state designator, '(', expression, ')' ;

• *MapOrSeqRef* :: *var* : *StateDesignator*
arg : *Expr*

6.5 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 ;

- o $\text{Stmt} = \text{LetStmt}$
 - | LetBeSTStmt
 - | DefStmt
 - | BlockStmt
 - | AssignStmt
 - | IfStmt
 - | CasesStmt
 - | SeqForLoop
 - | SetForLoop
 - | IndexForLoop
 - | WhileLoop
 - | NonDetStmt
 - | Call
 - | ReturnStmt
 - | AlwaysStmt
 - | TrapStmt
 - | RecTrapStmt
 - | ExitStmt
 - | IdentStmt
-

6.5.1 Local Binding Statements

* let statement = 'let', local definition, { ',', local definition }, 'in', statement ;

- $\text{LetStmt} :: \text{letdefs} : \text{LocalDef}^+$
 $\text{body} : \text{Stmt}$
-

* local definition = value definition | function definition ;

- o $\text{LocalDef} = \text{FunctionDef} \mid \text{ValueDef}$
-

* let be statement = 'let', bind, ['be', 'st', expression], 'in', statement ;

- $\text{LetBeSTStmt} :: \text{bind} : \text{Bind}$
 $\text{cond} : [\text{Expr}]$
 $\text{body} : \text{Stmt}$
-

* def statement = 'def', equals definition, { ',', equals definition }, 'in', statement ;

- $\text{DefStmt} :: \text{eqdefs} : \text{EqDef}^+$
 $\text{body} : \text{Stmt}$
-

* equals definition = pattern bind, '=', (expression | call statement) ;

- $\text{EqDef} :: \text{lhs} : \text{PatternBind}$
 $\text{rhs} : \text{Expr} \mid \text{Call}$
-

6.5.2 Block and Assignment Statements

* block statement = '(', { dcl statement }, statement, { ';' , statement }, ')' ;

- *BlockStmt::dcls : DclStmt**
stmts : Stmt⁺
-

* dcl statement = 'dcl', assignment definition, ';' ;

- *DclStmt = AssignDef*
-

* assignment definition = identifier, ':', type, [':=' , expression | ':=' , call statement] ;

- *AssignDef::var : Id*
tp : Type
dclinit : [Expr | Call]
-

* assign statement = state designator, ':=' , (expression | call statement) ;

- *AssignStmt::lhs : StateDesignator*
rhs : Expr | Call
-

6.5.3 Conditional Statements

* if statement = 'if', expression, 'then', statement, { elseif statement }, ['else', statement] ;

- *IfStmt::test : Expr*
cons : Stmt
*elsifaltn : ElsifStmt**
altn : Stmt
-

* elseif statement = 'elseif', expression, 'then', statement ;

- *ElsifStmt::test : Expr*
cons : Stmt
-

* cases statement = 'cases', expression, ':',
cases statement alternatives,
[',', others statement],
'end' ;

- *CasesStmt::sel : Expr*
altns : CaseStmtAlternatives
other : [OthersStmt]
-

- * cases statement alternatives = cases statement alternative, { ',', cases statement alternative } ;
- o *CasesStmtAlternatives* = *CaseStmtAltn*⁺

- * cases statement alternative = pattern list, '→', statement ;
- *CaseStmtAltn* :: *match* : *PatList*
 body : *Stmt*

- * others statement = 'others', '→', statement ;
- o *OthersStmt* = *Stmt*

6.5.4 Loop Statements

- * sequence for loop = 'for', pattern bind, 'in', ['reverse'], expression,
 'do', statement ;
- *SqForLoop* :: *cv* : *PatternBind*
 dirn : [REVERSE]
 forseq : *Expr*
 body : *Stmt*

- * set for loop = 'for', 'all', pattern, '∈', expression,
 'do', statement ;
- *SetForLoop* :: *cv* : *Pattern*
 forset : *Expr*
 body : *Stmt*

- * index for loop = 'for', identifier, '=', expression, 'to', expression, ['by', expression],
 'do', statement ;
- *IndexForLoop* :: *cv* : *Id*
 lb : *Expr*
 ub : *Expr*
 step : [*Expr*]
 body : *Stmt*

- * while loop = 'while', expression, 'do', statement ;
- *WhileLoop* :: *test* : *Expr*
 body : *Stmt*

6.5.5 Nondeterministic Statement

- * nondeterministic statement = ‘||’, ‘(’, statement, { ‘,’, statement }, ‘)’ ;
 - *NonDetStmt::stmts : Stmt⁺*
-

6.5.6 Call and Return Statements

- * call statement = name, ‘(’, [expression list], ‘)’, [‘using’, state designator] ;
 - *Call::oprt : Name*
args : OExprList
callst : [StateDesignator]
 - * return statement = ‘return’, [expression] ;
 - *ReturnStmt::expr : [Expr]*
-

6.5.7 Exception Handling Statements

- * always statement = ‘always’, statement, ‘in’, statement ;
- *AlwaysStmt::alupost : Stmt*
body : BlockStmt
- * trap statement = ‘trap’, pattern bind, ‘with’, statement, ‘in’, statement ;
- *TrapStmt::pat : PatternBind*
trappost : Stmt
body : BlockStmt
- * recursive trap statement = ‘tixe’, traps, ‘in’, statement ;
- *RecTrapStmt::traps : Traps*
body : BlockStmt
- * traps = ‘{’, trap, { ‘,’, trap }, ‘}’ ;
- *Traps = Trap⁺*
- * trap = pattern bind, ‘→’, statement ;

- *Trap*::*match* : *PatternBind*
trapost : *Stmt*
-

* exit statement = 'exit', [expression] ;

- *ExitStmt*::*expr* : [*Expr*]
-

6.5.8 Identity Statement

* identity statement = 'skip' ;

- *IdentStmt* = SKIP
-

6.6 Patterns and Bindings

6.6.1 Patterns

* pattern = pattern identifier
| match value
| set enum pattern
| set union pattern
| seq enum pattern
| seq conc pattern
| tuple pattern
| record pattern ;

- *Pattern* = *PatternId*
| *MatchVal*
| *SetEnumPattern*
| *SetUnionPattern*
| *SeqEnumPattern*
| *SeqConcPattern*
| *TuplePattern*
| *RecordPattern*
-

* pattern identifier = identifier | ‘_’ ;

- *PatternId*::*name* : [*Id*]
-

* match value = ‘(’, expression, ‘)’ | symbolic literal ;

- *MatchVal*::*val* : *Expr*
-

* set enum pattern = ‘{’, pattern list, ‘}’ ;

- *SetEnumPattern*::*els* : *PatList*
-

* set union pattern = pattern, ‘U’, pattern ;

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

* seq enum pattern = ‘[’, pattern list, ‘]’ ;

- *SeqEnumPattern*::*els* : *PatList*
-

* seq conc pattern = pattern, ‘~’, pattern ;

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

* tuple pattern = ‘mk-’, ‘(’, pattern, ‘,’, pattern list, ‘)’ ;

TuplePattern::*fields* : *PatList*

inv *mk-TuplePattern*(*pl*) \triangleq len *pl* \geq 2

* record pattern = name, ‘(’, [pattern list], ‘)’ ;

- *RecordPattern*::*id* : *Id*
 fields : *OPatList*
-

* pattern list = pattern, { ‘,’, pattern } ;

o *PatList* = *Pattern*⁺

o *OPatList* = *Pattern*^{*}

6.6.2 Bindings

* pattern bind = pattern | bind ;

o *PatternBind* = *Pattern* | *Bind*

* bind = set bind | type bind ;

- o $Bind = SetBind \mid TypeBind$
-

* set bind = pattern, ‘ \in ’, expression ;

- $SetBind ::= pat : Pattern$
 $bindset : Expr$
-

* type bind = pattern, ‘ $:$ ’, type ;

- $TypeBind ::= pat : Pattern$
 $type : Type$
-

* bind list = multiple bind, { ‘ $,$ ’, multiple bind } ;

- o $BindList = MultBind^+$
-

* multiple bind = multiple set bind | multiple type bind ;

- o $MultBind = MultSetBind \mid MultTypeBind$
-

* multiple set bind = pattern list, ‘ \in ’, expression ;

- $MultSetBind ::= pats : PatList$
 $expr : Expr$
-

* multiple type bind = pattern list, ‘ $:$ ’, type ;

- $MultTypeBind ::= pats : PatList$
 $tp : Type$
-

* type bind list = type bind, { ‘ $,$ ’, type bind } ;

- o $TypeBindList = TypeBind^+$
-

6.7 Lexical Specification

6.7.1 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.

6.7.2 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 6.1, with the forms of characters used in this report; 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 6.1 *) ;
- * keyword letter = (* see table 6.1 *) ;
- * distinguished letter = (* see table 6.1 *) ;
- * Greek letter = (* see table 6.1 *) ;
- * digit = (* see table 6.1 *) ;
- * delimiter character = (* see table 6.1 *) ;
- * other character = (* see table 6.1 *) ;

Table 6.1: Character set

plain letter:														
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>	<i>k</i>	<i>l</i>	<i>m</i>		
<i>n</i>	<i>o</i>	<i>p</i>	<i>q</i>	<i>r</i>	<i>s</i>	<i>t</i>	<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>		
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>	<i>G</i>	<i>H</i>	<i>I</i>	<i>J</i>	<i>K</i>	<i>L</i>	<i>M</i>		
<i>N</i>	<i>O</i>	<i>P</i>	<i>Q</i>	<i>R</i>	<i>S</i>	<i>T</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>Y</i>	<i>Z</i>		
keyword letter:														
<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>h</i>	<i>i</i>	<i>j</i>	<i>k</i>	<i>l</i>	<i>m</i>		
<i>n</i>	<i>o</i>	<i>p</i>	<i>q</i>	<i>r</i>	<i>s</i>	<i>t</i>	<i>u</i>	<i>v</i>	<i>w</i>	<i>x</i>	<i>y</i>	<i>z</i>		
distinguished letter:														
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>	<i>G</i>	<i>H</i>	<i>I</i>	<i>J</i>	<i>K</i>	<i>L</i>	<i>M</i>		
<i>N</i>	<i>O</i>	<i>P</i>	<i>Q</i>	<i>R</i>	<i>S</i>	<i>T</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>Y</i>	<i>Z</i>		
<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>	<i>G</i>	<i>H</i>	<i>I</i>	<i>J</i>	<i>K</i>	<i>L</i>	<i>M</i>		
<i>N</i>	<i>O</i>	<i>P</i>	<i>Q</i>	<i>R</i>	<i>S</i>	<i>T</i>	<i>U</i>	<i>V</i>	<i>W</i>	<i>X</i>	<i>Y</i>	<i>Z</i>		
Greek letter:														
α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ			
ν	ξ	\circ	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω			
<i>A</i>	<i>B</i>	Γ	Δ	<i>E</i>	<i>Z</i>	<i>H</i>	Θ	<i>I</i>	<i>K</i>	<i>A</i>	<i>M</i>			
<i>N</i>	Ξ	<i>O</i>	Π	<i>P</i>	Σ	<i>T</i>	Υ	Φ	<i>X</i>	Ψ	Ω			
digit:														
0	1	2	3	4	5	6	7	8	9					
delimiter character:														
,	:	;	=	()		-	[]	{	}	*		
+	/	\nwarrow	\nearrow	\searrow	\swarrow	∇	Δ	\neq	\forall	\exists	.	\rightarrow		
+	\xrightarrow{m}	\xleftarrow{m}	\rightarrow	\xrightarrow{t}	\xrightarrow{o}	∇	Δ	\parallel	\Rightarrow	\Leftrightarrow	.	\hookrightarrow	\trianglelefteq	
\cup	\cap	\triangleleft	\triangleright	\triangleleft	\triangleright	∇	Δ	ι	λ	μ	ϵ	\times	\dagger	\circ
\wedge	\vee	\boxplus	\boxminus	\boxtimes	\boxdot	\boxuparrow	\boxdownarrow	\boxsubset	\boxsupset	\boxtimes				
other character:														
-	'	,	"	"	@	!	'							

6.7.3 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’
 | ‘ti’ | ‘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.

- o $Id = ValueId \mid PreId \mid PostId \mid InvId \mid InitId \mid MkId \mid IsId$

The identifiers in MCS and OAS are related as described in the following table:

MCS identifier has prefix which is	OAS identifier belongs to
<i>non-reserved</i>	<i>ValueId</i>
<i>pre-</i>	<i>PreId</i>
<i>post-</i>	<i>PostId</i>
<i>inv-</i>	<i>InvId</i>
<i>init-</i>	<i>InitId</i>
<i>mk-</i>	<i>MkId</i>
<i>is-</i>	<i>IsId</i>

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 $mk\text{-}PreId(f_1)$ is using the same token as the name of the corresponding function which will be called $mk\text{-}ValueId(f_1)$).

• *ValueId*:: token

• *PreId*:: token

• *PostId*:: token

• *InvId*:: token

• *InitId*:: token

• *MkId*:: token

• *IsId*:: token

* symbolic literal = numeric literal
| boolean literal
| nil literal
| character literal
| text literal
| quote literal ;

○ *Literal* = *NumLit*

| *BoolLit*
| *NilLit*
| *CharLit*
| *TextLit*
| *QuoteLit*

* numeral = digit, { digit } ;

* numeric literal = numeral, ['.', digit, { digit }], [exponent] ;

* exponent = '×10ⁿ', ['+' | '-'], numeral ;

note: $\times 10^n$ can be written $\times 10^n$.

• *NumLit*:: val : ℝ

* boolean literal = 'true' | 'false' ;

• *BoolLit*:: val : B

* nil literal = 'nil' ;

- *NilLit*::
-

* character literal = “”, character – newline, “” ;

- *CharLit*:: *val* : char
-

* text literal = “”, { “”’ | character – (“” | newline) }, “” ;

- *TextLit*:: *val* : char*
-

* quote literal = distinguished letter, { distinguished letter | digit | ‘-’ | “” } ;

- *QuoteLit*:: *val* : char*
-

* comment = ‘–’, { character – newline }, newline ;

6.8 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.

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 ’, ‘ \dots ’, etc.

The precedence ordering on the families is:

combinators > applicators > evaluators > relations > connectives > constructors

6.8.1 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

6.8.2 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 ;

6.8.3 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} ' | ' \cup ' | ' \cap ' ;

sequence prefix operator = 'hd' | 'tl' | 'len' | 'inds' | 'elems' | 'conc' ;

map inverse = expression, ' $^{-1}$ ' ;

map prefix operator = 'dom' | 'rng' | 'merge' ;

arithmetic infix operator = '+' | '-' | 'x' | '/' | 'rem' | 'mod' | 'div' ;

```

set infix operator = 'U' | 'n' | 'V';
sequence infix operator = '~~';
map infix operator = '⊸' | '†' | '◁' | '⊸' | '▷' | '▷';

```

The precedence ordering follows a pattern of analogous operators. The family is defined in the following table.

precedence level	arithmetic	set	map	sequence
1	+ -	$\cup \backslash$	$\bowtie \dagger$	\curvearrowright
2	$\times /$ 'rem' 'mod' 'div'	\cap		
3				-1
4			$\triangleleft \triangleleft$ $\triangleright \triangleright$	
5	(unary) + (unary) - 'abs' 'floor'	'card' \mathcal{F} \cap \cup	'dom' 'rng' 'merge'	'len' 'elems' 'hd' 'tl' 'conc' 'inds'

6.8.4 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 = '=' | '≠' | '<' | '≤' | '>' | '≥' ;
set relational operator = '⊆' | 'C' | '∈' | '∉' ;

```

precedence level	relation
1	\leq <
	\geq >
	= ≠
	\subseteq \subset
	\in \notin

All operators in the Relations family have equal precedence.

6.8.5 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 = '¬' ;
logical infix operator = '∧' | '∨' | '⇒' | '↔' ;

```

precedence level	connective
1	\Leftrightarrow
2	\Rightarrow
3	\vee
4	\wedge
5	\neg

6.8.6 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.

6.8.7 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.

6.8.8 The Type Operators

Type operators have their own separate precedence ordering, as follows:

1. Function and map types: $\rightarrow \xrightarrow{t} \xleftarrow{m}$, \xrightarrow{m} (right grouping).
2. Union type: $|$ (left grouping).
3. Other binary type operators: \times (no grouping).
4. Unary type operators: $*$, $+$, '-set'.

Chapter 7

The Interchange Concrete Syntax

7.1 Introduction

The interchange concrete syntax is an alternative to the mathematical concrete syntax of chapter ??, based on the coded character set of ISO 646 – 1983. 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.

7.2 Lexis

The character set consists of the following subset of the coded character set defined in ISO 646:

- 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
 1. a space character
 2. a horizontal tabulation character
 3. 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]

7.3 Symbols

Table 7.1 below shows the equivalent interchange syntax representation of each symbol of the mathematical concrete syntax.

Table 7.1: 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
	$ \begin{array}{cccccccccccc} \alpha & \beta & \gamma & \delta & \epsilon & \zeta & \eta & \theta & \iota & \kappa & \lambda & \mu \\ a & b & g & d & e & z & h & q & i & k & l & m \\ \nu & \xi & o & \pi & \rho & \sigma & \tau & v & \phi & \chi & \psi & \omega \\ n & x & o & p & r & s & t & u & f & c & y & w \end{array} $ $ \begin{array}{cccccccccccc} A & B & \Gamma & \Delta & E & Z & H & \Theta & I & K & \Lambda & M \\ A & B & G & D & E & Z & H & Q & I & K & L & M \\ N & \Xi & O & \Pi & P & \Sigma & T & \Upsilon & \Phi & X & \Psi & \Omega \\ N & X & O & P & R & S & T & U & F & C & Y & W \end{array} $
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	corresponding small letters
a to z	
(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	corresponding capital and lower-case letters
A to z	
A to Z	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)

Table 7.1: (continued)

mathematical syntax	interchange syntax
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 annotation	from -- (minus sign, minus sign) to next carriage return from annotation to next end annotation
(7) Other symbols	
,	,
:	:
;	;
=	= (equals sign, 3/13)
)) (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)
.	.
.	.
x	& (ampersand, 2/6)
'	*
≤	' (grave accent, 6/0)
≥	<=
≠	>=
→	<>
→	==>
t	-t>
⇒	=>
↔	<=>
↔	->
△	==
:=	:=
...	...
::	::
↑	**
†	++
△	<:
▷	:>

Table 7.1: (continued)

mathematical syntax	interchange syntax
\triangleleft	<:
\triangleright	:>
\sqsubseteq	munion
\sqcup	psubset
\sqcap	subset
\sim	~
\sqcap_1	dinter
\sqcup_1	dunion
\mathcal{F}	inverse [see note 3]
*	power
+	seq of .. [see note 2]
\xrightarrow{m}	seq1 of .. [see note 2]
\xrightarrow{m}	inmap .. to .. [see note 2]
ι	map .. to .. [see note 2]
λ	iota
μ	lambda
\mathbb{B}	mu
\mathbb{N}	bool
\mathbb{N}_1	nat
\mathbb{Z}	nat1
\mathbb{Q}	int
\mathbb{R}	rat
\sqsubset	real
\sqsupset	not
\sqcap	not
\sqcup	inter
\in	union
\notin	in set
\circ	not in set
\wedge	comp
\vee	and
\forall	or
\exists	forall
$\exists!$	exists
	exists1

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 $T_1 \xrightarrow{m} T_2$ and $T_1 \xleftarrow{m} T_2$ 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.

Chapter 8

The Syntax Mapping

8.1 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.

8.1.1 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 8.1, with arrows denoting the import relations between the modules.

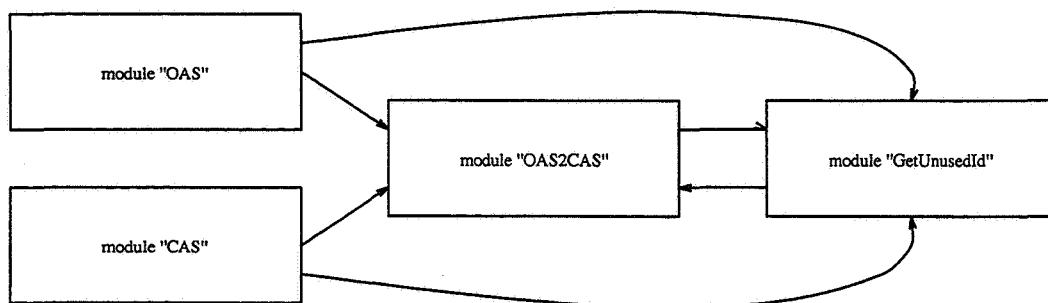


Figure 8.1: Structure of the syntax mapping

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.

Module “OAS”

This module contains type definitions for the standard abstract syntax (OAS). These type definitions can be found in chapter ??.

Module “CAS”

This module contains type definitions for the core abstract syntax (CAS). These type definitions can be found in chapter 3.

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 to the space which a complete formal definition of this function would take. See section 8.1.3 for a description of the situations in which the function is used.

8.1.2 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.

8.1.3 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:

```
Definitions:: typem    : ValueId  $\xrightarrow{m}$  TypeDef
            expofnm : Id  $\xrightarrow{m}$  ExplPolyFnDef
            exmofnm : Id  $\xrightarrow{m}$  ExplFnDef
            impofnm : Id  $\xrightarrow{m}$  ImplPolyFnDef
            immofnm: Id  $\xrightarrow{m}$  ImplFnDef
            valuem   : Pattern  $\xrightarrow{m}$  ValDef
            explopm  : ValueId  $\xrightarrow{m}$  ExplOpDef
            implopm  : ValueId  $\xrightarrow{m}$  ImplOpDef
            state     : [StateDef]
```

Information on how each part is transformed now follows.

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*:

1.0 $fn_{OAS} : I_1 \times \dots \times I_n \rightarrow \dots$
 .1 $fn_{OAS}(i_1, \dots, i_n) \triangleq$
 .2 ...

The equivalent of this function in the CAS is:

2.0 $fn_{CAS} : (I_1 \times \dots \times I_n) \rightarrow \dots$
 .1 $fn_{CAS}(i) \triangleq$
 .2 let $mk-(i_1, \dots, i_n) = i$ in
 .3 ...

where i is a newly introduced identifier. As already explained, the function *GetUnusedId* is used to produce such identifiers.

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 $fn : I \rightarrow O$
 .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 $pre-fn : I \rightarrow \mathbb{B}$
 .1 $pre-fn(i) \triangleq$
 .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 $post-fn : I \times O \rightarrow \mathbb{B}$
 .1 $post-fn(i, o) \triangleq$
 .2 $Post_{fn}(i, o)$

The Generation of Quoted Pre- and Post-conditions for Operations

For all operations op in the specification, defined as:

8.0 $op : I \xrightarrow{\sigma} O$
 .1 $op(i) \dots$
 .2 pre $Pre_{op}(i, \sigma)$

and

```
9.0  op (i : I) o : O
    .1  ext rd σ1 : Σ1
    .2      wr σ2 : Σ2
    .3  pre Preop(i, σ)
    .4  post ...
```

given a state definition

```
10.0 state Σ of
    .1      σ1 : Σ1
    .2      σ2 : Σ2
    .3      σ3 : Σ3
    .4  end
```

a definition for a function called *pre-op* is generated, which is defined as:

```
11.0  pre-op : I × Σ → B
    .1  pre-op (i, σ) △
    .2      Preop(i, σ)
```

For all operations *op* in the specification, defined as:

```
12.0  op (i : I) o : O
    .1  ext rd σ1 : Σ1
    .2      wr σ2 : Σ2
    .3  pre ...
    .4  post Postop(i, o, σ̄1, σ̄2, σ1, σ2)
```

a definition for a function called *post-op* is generated, which is defined as:

```
13.0  post-op : I × O × Σ × Σ → B
    .1  post-op (i, o, σ', σ) △
    .2      let mk-Σ(σ'1, σ'2, σ'3) = σ',
    .3          mk-Σ(σ1, σ2, σ3) = σ in
    .4      Postop(i, o, σ'1, σ'2, σ1, σ2)
```

The references to ‘old values’ in the post-condition (σ_1 and σ_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).

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.

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 \bar{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.

Guards of Error Handlers

A similar situation as in the previous subsubsection occurs 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.

Quoted Post-conditions of Implicit Operations

The construction of the body for the function $\text{post-}op$, where op is an implicitly defined operation, requires the systematic replacement of references to old values of state components by new identifiers (see also section 8.1.3).

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

$$14.0 \quad Env = OAS.Id \xrightarrow{m} CAS.Id$$

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 subsubsections above ρ is empty ($\{\mapsto\}$).

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.

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.

The Transformation of the State

The syntax mapping must convert the state from its OAS form; name_{OAS} , $\text{field reference}_{OAS}$, and $\text{map or sequence reference}_{OAS}$, to the CAS representation of; state_{CAS} , fieldlist_{CAS} , invariant_{CAS} , and $\text{initialization}_{CAS}$. Transforming the state results in the following maps:

- $CAS.Id \xrightarrow{m} CAS.TypeDef$
- $CAS.Id \xrightarrow{m} CAS.ExplFnDef$

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, re-configured 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.

8.1.4 Notational Conventions

The syntax mapping has been defined using standard VDM-SL, the only exception being the use of a provisional structuring mechanism (see section 8.1.1).

8.2 Syntaxes and Auxiliary Functions

8.2.1 Module “OAS”

```
module OAS
  exports
15.0  types all

  definitions
    types
      .1   — See chapter ?? on page ??, in which the Outer Abstract Syntax has been defined.
      .2

end OAS
```

8.2.2 Module “CAS”

```
module CAS
  exports
16.0  types all

  definitions
    types
      .1   — See chapter 3 on page 18, in which the Core Abstract Syntax has been defined.
      .2

end CAS
```

8.2.3 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 : () → OAS‘Id × CAS‘Id
        .1  GetUnusedId () △
        .2  let idOAS : OAS‘Id be st IsUniqueId (idOAS) in
        .3  let idCAS = Id2Id (idOAS, {↑}) in
        .4  mk- (idOAS, idCAS)
    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 → B
    .1  IsUniqueId (-) △
    .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
```

8.3 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

definitions
types
30.0   Env = OAS`Id  $\xrightarrow{m}$  CAS`Id

annotations
30.0   See section 8.1.3 for a description of the use of type Env.

end annotations

```

8.3.1 Document

functions

```

31.0   Document2Definitions : OAS`Document → CAS`Definitions
.1     Document2Definitions (dbs)  $\triangleq$ 
.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  $\sqcup$  tdm,
.12        expofnm                         = def-expofnm  $\sqcup$  quote-expofnm  $\sqcup$  quote-impofnm,
.13        exmofnm                         = def-exmofnm  $\sqcup$  quote-exmofnm  $\sqcup$  quote-immofnm  $\sqcup$ 
.14                                     quote-explopm  $\sqcup$  quote-implopm  $\sqcup$  quote-typem  $\sqcup$ 
.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 invariant, and invariant initial function of the state.)

end annotations ;

32.0 *DefinitionBlocks2TypeDefMap* :

- .1 $OAS^{\text{DefinitionBlock}}^+ \rightarrow$
- .2 $(CAS^{\text{Id}} \xrightarrow{m} CAS^{\text{TypeDef}}) \times (CAS^{\text{Id}} \xrightarrow{m} CAS^{\text{ExplFnDef}})$
- .3 $\text{DefinitionBlocks2TypeDefMap}(dbs) \triangleq$
 let $\text{defs} = \{ \text{TypeDefinitions2TypeDefMap}(tds) \mid$
 $\quad \text{mk- } OAS^{\text{TypeDefinitions}}(tds) \in \text{elems } dbs \} \text{ in}$
 $\quad \text{mk- } (\text{merge } \{ \text{tdm} \mid \text{mk- } (\text{tdm}, -) \in \text{defs} \}, \text{merge } \{ \text{tdm}_q \mid \text{mk- } (-, \text{tdm}_q) \in \text{defs} \})$
- .7 pre let $\text{defs} = \{ \text{TypeDefinitions2TypeDefMap}(tds) \mid$
 $\quad \text{mk- } OAS^{\text{TypeDefinitions}}(tds) \in \text{elems } dbs \} \text{ in}$
 $\quad \forall \text{mk- } (\text{tdm}, \text{tdm}_q) \in \text{defs} \cdot$
 $\quad \forall \text{mk- } (\text{tdm}', \text{tdm}_q) \in \text{defs} \setminus \{ \text{mk- } (\text{tdm}, \text{tdm}_q) \} \cdot \text{dom } \text{tdm} \cap \text{dom } \text{tdm}' = \{ \}$
- .10 annotations

.7 The pre-condition ensures that all type identifiers are unique.

end annotations ;

33.0 *DefinitionBlocks2ExplPolyFnDefMap* :

- .1 $OAS^{\text{DefinitionBlock}}^+ \rightarrow$
- .2 $(CAS^{\text{Id}} \xrightarrow{m} CAS^{\text{ExplPolyFnDef}}) \times (CAS^{\text{Id}} \xrightarrow{m} CAS^{\text{ExplPolyFnDef}})$
- .3 $\text{DefinitionBlocks2ExplPolyFnDefMap}(dbs) \triangleq$
 let $\text{defs} = \{ \text{FunctionDefinitions2ExplPolyFnDefMap}(fds) \mid$
 $\quad \text{mk- } OAS^{\text{FunctionDefinitions}}(fds) \in \text{elems } dbs \} \text{ in}$
 $\quad \text{mk- } (\text{merge } \{ \text{fdm} \mid \text{mk- } (\text{fdm}, -) \in \text{defs} \}, \text{merge } \{ \text{fdm}_q \mid \text{mk- } (-, \text{fdm}_q) \in \text{defs} \})$
- .7 pre let $\text{defs} = \{ \text{FunctionDefinitions2ExplPolyFnDefMap}(fds) \mid$
 $\quad \text{mk- } OAS^{\text{FunctionDefinitions}}(fds) \in \text{elems } dbs \} \text{ in}$
 $\quad \forall \text{mk- } (\text{fdm}, \text{fdm}_q) \in \text{defs} \cdot$
 $\quad \forall \text{mk- } (\text{fdm}', \text{fdm}_q) \in \text{defs} \setminus \{ \text{mk- } (\text{fdm}, \text{fdm}_q) \} \cdot \text{dom } \text{fdm} \cap \text{dom } \text{fdm}' = \{ \}$
- .10 annotations

.7 The pre-condition ensures that all explicit polymorphic function identifiers are unique.

end annotations ;

34.0 *DefinitionBlocks2ExplFnDefMap* :

- .1 $OAS^{\text{DefinitionBlock}}^+ \rightarrow$
- .2 $(CAS^{\text{Id}} \xrightarrow{m} CAS^{\text{ExplFnDef}}) \times (CAS^{\text{Id}} \xrightarrow{m} CAS^{\text{ExplFnDef}})$
- .3 $\text{DefinitionBlocks2ExplFnDefMap}(dbs) \triangleq$
 let $\text{defs} = \{ \text{FunctionDefinitions2ExplFnDefMap}(fds) \mid$
 $\quad \text{mk- } OAS^{\text{FunctionDefinitions}}(fds) \in \text{elems } dbs \} \text{ in}$
 $\quad \text{mk- } (\text{merge } \{ \text{fdm} \mid \text{mk- } (\text{fdm}, -) \in \text{defs} \}, \text{merge } \{ \text{fdm}_q \mid \text{mk- } (-, \text{fdm}_q) \in \text{defs} \})$
- .7 pre let $\text{defs} = \{ \text{FunctionDefinitions2ExplFnDefMap}(fds) \mid$
 $\quad \text{mk- } OAS^{\text{FunctionDefinitions}}(fds) \in \text{elems } dbs \} \text{ in}$
 $\quad \forall \text{mk- } (\text{fdm}, \text{fdm}_q) \in \text{defs} \cdot$
 $\quad \forall \text{mk- } (\text{fdm}', \text{fdm}_q) \in \text{defs} \setminus \{ \text{mk- } (\text{fdm}, \text{fdm}_q) \} \cdot \text{dom } \text{fdm} \cap \text{dom } \text{fdm}' = \{ \}$
- .10 annotations

.7 The pre-condition ensures that all explicit function identifiers are unique.

end annotations ;

35.0 *DefinitionBlocks2ImplPolyFnDefMap* :
 .1 *OAS[‘]DefinitionBlock⁺* →
 .2 (*CAS[‘]Id* \xrightarrow{m} *CAS[‘]ImplPolyFnDef*) × (*CAS[‘]Id* \xrightarrow{m} *CAS[‘]ExplPolyFnDef*)
 .3 *DefinitionBlocks2ImplPolyFnDefMap* (*dbs*) \triangleq
 .4 let *defs* = {*FunctionDefinitions2ImplPolyFnDefMap* (*fds*) |
 .5 *mk-OAS[‘]FunctionDefinitions* (*fds*) ∈ *elems* *dbs*} in
 .6 *mk-* (merge {*fdm* | *mk-* (*fdm*, -) ∈ *defs*}, merge {*fdm_q* | *mk-* (-, *fdm_q*) ∈ *defs*})
 .7 pre let *defs* = {*FunctionDefinitions2ImplPolyFnDefMap* (*fds*) |
 .8 *mk-OAS[‘]FunctionDefinitions* (*fds*) ∈ *elems* *dbs*} in
 .9 \forall *mk-* (*fdm*, *fdm_q*) ∈ *defs* ·
 .10 \forall *mk-* (*fdm*', -) ∈ *defs* \ {*mk-* (*fdm*, *fdm_q*)} · *dom* *fdm* ∩ *dom* *fdm*' = {}

annotations

.7 The pre-condition ensures that all implicit polymorphic function identifiers are unique.

end annotations ;

36.0 *DefinitionBlocks2ImplFnDefMap* :
 .1 *OAS[‘]DefinitionBlock⁺* →
 .2 (*CAS[‘]Id* \xrightarrow{m} *CAS[‘]ImplFnDef*) × (*CAS[‘]Id* \xrightarrow{m} *CAS[‘]ExplFnDef*)
 .3 *DefinitionBlocks2ImplFnDefMap* (*dbs*) \triangleq
 .4 let *defs* = {*FunctionDefinitions2ImplFnDefMap* (*fds*) |
 .5 *mk-OAS[‘]FunctionDefinitions* (*fds*) ∈ *elems* *dbs*} in
 .6 *mk-* (merge {*fdm* | *mk-* (*fdm*, -) ∈ *defs*}, merge {*fdm_q* | *mk-* (-, *fdm_q*) ∈ *defs*})
 .7 pre let *defs* = {*FunctionDefinitions2ImplFnDefMap* (*fds*) |
 .8 *mk-OAS[‘]FunctionDefinitions* (*fds*) ∈ *elems* *dbs*} in
 .9 \forall *mk-* (*fdm*, *fdm_q*) ∈ *defs* ·
 .10 \forall *mk-* (*fdm*', -) ∈ *defs* \ {*mk-* (*fdm*, *fdm_q*)} · *dom* *fdm* ∩ *dom* *fdm*' = {}

annotations

.7 The pre-condition ensures that all implicit function identifiers are unique.

end annotations ;

37.0 *DefinitionBlocks2ValDefMap* :
 .1 *OAS[‘]DefinitionBlock⁺* → (*CAS[‘]Pattern* \xrightarrow{m} *CAS[‘]ValDef*)
 .2 *DefinitionBlocks2ValDefMap* (*dbs*) \triangleq
 .3 merge {*ValueDefinitions2ValDefMap* (*vds*) |
 .4 *mk-OAS[‘]ValueDefinitions* (*vds*) ∈ *elems* *dbs*}
 .5 pre let *defs* = {*ValueDefinitions2ValDefMap* (*vds*) |
 .6 *mk-OAS[‘]ValueDefinitions* (*vds*) ∈ *elems* *dbs*} in
 .7 \forall *vdm* ∈ *defs* · \forall *vdm*' ∈ *defs* \ {*vdm*} · *dom* *vdm* ∩ *dom* *vdm*' = {}

annotations

.5 The pre-condition ensures that all value patterns are unique.

end annotations ;

38.0 *DefinitionBlocks2ExplOpDefMap* :
 .1 *OAS[‘]DefinitionBlock⁺* × *CAS[‘]TypeDef* →
 .2 (*CAS[‘]Id* \xrightarrow{m} *CAS[‘]ExplOpDef*) × (*CAS[‘]Id* \xrightarrow{m} *CAS[‘]ExplFnDef*)
 .3 *DefinitionBlocks2ExplOpDefMap* (*dbs*, σ) \triangleq
 .4 let *defs* = {*OperationDefinitions2ExplOpDefMap* (*ods*, σ) |
 .5 *mk-OAS[‘]OperationDefinitions* (*ods*) ∈ *elems* *dbs*} in
 .6 *mk-* (merge {*odm* | *mk-* (*odm*, -) ∈ *defs*}, merge {*odm_q* | *mk-* (-, *odm_q*) ∈ *defs*})

```

.7   pre let defs = {OperationDefinitions2ExplOpDefMap(ods,  $\sigma$ ) |
.8     mk-OAS‘OperationDefinitions(ods)  $\in$  elems dbs} in
.9      $\forall$  mk-(odm, odmq)  $\in$  defs .
.10     $\forall$  mk-(odm', -)  $\in$  defs \ {mk-(odm, odmq)} · dom odm  $\cap$  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  $\rightarrow$ 
.2     (CAS‘Id  $\xrightarrow{m}$  CAS‘ImplOpDef)  $\times$  (CAS‘Id  $\xrightarrow{m}$  CAS‘ExplFnDef)
.3   DefinitionBlocks2ImplOpDefMap(dbs,  $\sigma$ )  $\triangleq$ 
.4     let defs = {OperationDefinitions2ImplOpDefMap(ods,  $\sigma$ ) |
.5       mk-OAS‘OperationDefinitions(ods)  $\in$  elems dbs} in
.6       mk-(merge {odm | mk-(odm, -)  $\in$  defs}, merge {odmq | mk-(-, odmq)  $\in$  defs})
.7     pre let defs = {OperationDefinitions2ImplOpDefMap(ods,  $\sigma$ ) |
.8       mk-OAS‘OperationDefinitions(ods)  $\in$  elems dbs} in
.9        $\forall$  mk-(odm, odmq)  $\in$  defs .
.10       $\forall$  mk-(odm', -)  $\in$  defs \ {mk-(odm, odmq)} · dom odm  $\cap$  dom odm' = { }

```

annotations

.7 The pre-condition ensures that all implicit operation identifiers are unique.

end annotations ;

```

40.0   DefinitionBlocks2StateDef :
.1     OAS‘DefinitionBlock+  $\rightarrow$ 
.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       -  $\curvearrowright$  [mk-OAS‘StateDef(sd)]  $\curvearrowright$  -  $\rightarrow$  StateDef2StateDef(mk-OAS‘StateDef(stid, fields, stinv, stinit)),
.6       others  $\rightarrow$  mk-(nil, {→}, {→})
.7     end
.8   pre len [dbs(i) | i  $\in$  inds dbs · is-OAS‘StateDef(dbs(i))]  $\leq$  1

```

annotations

.8 The pre-condition ensures that a specification contains at most one state definition.

end annotations

8.3.2 Definitions

Type Definitions

functions

```

41.0   TypeDefinitions2TypeDefMap :
.1     OAS‘TypeDef+  $\rightarrow$ 
.2     (CAS‘Id  $\xrightarrow{m}$  CAS‘TypeDef)  $\times$  (CAS‘Id  $\xrightarrow{m}$  CAS‘ExplFnDef)
.3   TypeDefinitions2TypeDefMap(tds)  $\triangleq$ 
.4     let defs = {TypeDefinition2TypeDefMap(td) | td  $\in$  elems tds} in
.5       mk-(merge {tdm | mk-(tdm, -)  $\in$  defs}, merge {tdmq | mk-(-, tdmq)  $\in$  defs})
.6     pre let defs = {TypeDefinition2TypeDefMap(td) | td  $\in$  elems tds} in
.7        $\forall$  mk-(tdm, tdmq)  $\in$  defs .
.8        $\forall$  mk-(tdm', -)  $\in$  defs \ {mk-(tdm, tdmq)} · dom tdm  $\cap$  dom tdm' = { }

```

annotations

.6 The pre-condition ensures that all type identifiers are unique.

end annotations ;

42.0 $TypeDefinition2TypeDefMap :$

.1 $OAS^{\prime} TypeDef \rightarrow$

.2 $(CAS^{\prime} ValueId \xrightarrow{m} CAS^{\prime} TypeDef) \times (CAS^{\prime} Id \xrightarrow{m} CAS^{\prime} ExplFnDef)$

.3 $TypeDefinition2TypeDefMap(td) \triangleq$

.4 let $mk\text{-}(id_{OAS}, shape_{CAS}, typeinv_{OAS}) =$

.5 cases $td :$

.6 $mk\text{-}OAS^{\prime} UnTaggedTypeDef(id, shape, typeinv) \rightarrow$

.7 $mk\text{-}(id, Type2Type(shape), typeinv),$

.8 $mk\text{-}OAS^{\prime} TaggedTypeDef(id, fields, typeinv) \rightarrow$

.9 $mk\text{-}(id, Type2Type(mk\text{-}OAS^{\prime} CompositeType(id, fields)), typeinv)$

.10 end,

.11 $lambda_{CAS} = InvInitFn2Lambda(typeinv_{OAS}, shape_{CAS}),$

.12 $typeid_{CAS} = Id2Id(id_{OAS}, \{ \mapsto \}),$

.13 $inv\text{-}id_{CAS} = \text{let } mk\text{-}OAS^{\prime} ValueId(id) = id_{OAS} \text{ in}$

.14 $Id2Id(mk\text{-}OAS^{\prime} InvId(id), \{ \mapsto \}),$

.15 $typedef_{CAS} = mk\text{-}CAS^{\prime} TypeDef(shape_{CAS}, mk\text{-}(inv\text{-}id_{CAS}, lambda_{CAS})),$

.16 $explfndef_{CAS} = mk\text{-}CAS^{\prime} ExplFnDef(shape_{CAS}, mk\text{-}CAS^{\prime} BasicType(BOOLEAN),$

.17 $mk\text{-}CAS^{\prime} BoolLit(true), lambda_{CAS}, TOTAL) \text{ in}$

.18 $mk\text{-}(\{ typeid_{CAS} \mapsto typedef_{CAS} \}, \{ inv\text{-}id_{CAS} \mapsto explfndef_{CAS} \})$

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 $Type2Type(mk\text{-}OAS^{\prime} CompositeType(id, fields))$ 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 $InvInitFn2Lambda : [OAS^{\prime} InvInitFn] \times CAS^{\prime} Type \rightarrow CAS^{\prime} Lambda$

.1 $InvInitFn2Lambda(invinitfn, type_{CAS}) \triangleq$

.2 let $mk\text{-}(id_{OAS}, id_{CAS}) = GetUnusedId(),$

.3 $parm_{CAS} = mk\text{-}CAS^{\prime} Par(id_{CAS}, type_{CAS}),$

.4 $body_{CAS} = \text{if } invinitfn = \text{nil} \text{ then } mk\text{-}CAS^{\prime} BoolLit(true)$

.5 $\text{else let } mk\text{-}OAS^{\prime} InvInitFn(pat, expr) = invinitfn \text{ in}$

.6 $MakeLetExpr(pat, nil, id_{OAS}, expr, \{ \mapsto \}) \text{ in}$

.7 $mk\text{-}CAS^{\prime} Lambda(parm_{CAS}, body_{CAS})$

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);

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  valCAS = [mk-CAS‘CharLit (val(i)) | i ∈ inds val],
.4  litCAS = mk-CAS‘QuoteLit (valCAS) in
.5  mk-CAS‘QuoteType (litCAS);

```

47.0 *CompositeType2CompositeType* : OAS‘CompositeType → CAS‘CompositeType

```

.1  CompositeType2CompositeType (mk-OAS‘CompositeType (id, fields)) △
.2  let idCAS = Id2Id (id, {→}),
.3  fieldsCAS = [Field2Field (fields(i)) | i ∈ inds fields] in
.4  mk-CAS‘CompositeType (idCAS, fieldsCAS)
.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 selCAS = if sel = nil then nil
.3  else Id2Id (sel, {→}),
.4  typeCAS = Type2Type (type) in
.5  mk-CAS‘Field (selCAS, typeCAS)

```

```

.6      pre  is-OAS'ValueId (sel)

annotations
.6  The pre-condition ensures that the field name has no reserved prefix.

end annotations ;

49.0    UnionType2UnionType : OAS'UnionType → CAS'UnionType
.1      UnionType2UnionType (mk-OAS'UnionType (summands)) △
.2      let  $tp_{CAS} = \{ Type2Type (summands(i)) \mid i \in \text{inds } summands \}$  in
.3      mk-CAS'UnionType ( $tp_{CAS}$ )
.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  $fields_{CAS} = [Type2Type (factors(i)) \mid i \in \text{inds } factors]$  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 $FnType2FnType : OAS^{\text{FnType}} \rightarrow CAS^{\text{FnType}}$

.1 $FnType2FnType(fntype) \triangleq$
 let dom_{CAS} = if $fntype.fndom = \text{UNITTYPE}$
 then $mk-CAS^{\text{BasicType}}(\text{UNIT})$
 else $Type2Type(fntype.fndom)$,
 $rng_{CAS} = Type2Type(fntype.fnrng)$ in
 $mk-CAS^{\text{FnType}}(dom_{CAS}, rng_{CAS})$;

58.0 $TypeName2TypeId : OAS^{\text{TypeName}} \rightarrow CAS^{\text{TypeId}}$

.1 $TypeName2TypeId(mk-OAS^{\text{Name}}(name)) \triangleq$
 let $id_{CAS} = Id2Id(name, \{\mapsto\})$ in
 $mk-CAS^{\text{TypeId}}(id_{CAS})$
 pre $\text{is-}OAS^{\text{ValueId}}(name)$

Annotations

.4 The pre-condition ensures that the type name has no reserved prefix.

end annotations ;

59.0	$TypeVar2TypeVar : OAS`TypeVar \rightarrow CAS`TypeVar$
.1	$TypeVar2TypeVar (mk-OAS`TypeVarId (id)) \triangleq$
.2	$\text{let } id' = Id2Id (id, \{\mapsto\}),$
.3	$mk-OAS`ValueId (id_{CAS}) = id' \text{ in}$
.4	$mk-CAS`TypeVar (mk-CAS`TypeVarId (id_{CAS}))$
.5	pre $is-OAS`ValueId (id)$

Annotations

.5 The pre-condition ensures that the type variable identifier has no reserved prefix.

end annotations

State Definition

functions

```

60.0 StateDef2StateDef : OAS`StateDef → CAS`StateDef ×
.1 (CAS`ValueId ↗ CAS`TypeDef) × (CAS`Id ↗ CAS`ExplFnDef)
.2 StateDef2StateDef (mk-OAS`StateDef (stid, fields, stinv, stinit)) △
.3 let mk- (typedefmapCAS, inv-stidmCAS) =
.4 TypeDefinition2TypeDefMap (mk-OAS`TaggedTypeDef (stid, fields, stinv)),
.5 stidCAS = Id2Id (stid, {↑}),  

.6 tpCAS = stidCAS,  

.7 init-stidCAS = let mk-OAS`ValueId (id) = stid in  

.8 Id2Id (mk-OAS`InitId (id), {↑}),  

.9 lambdaCAS = InvInitFn2Lambda (stinit, tpCAS),  

.10 initCAS = mk- (init-stidCAS, lambdaCAS),  

.11 statedefCAS = mk-CAS`StateDef (stidCAS, tpCAS, initCAS),  

.12 explfndefCAS = mk-CAS`ExplFnDef (tpCAS, mk-CAS`BasicType(BOOL),
.13 mk-CAS`BoolLit(true), lambdaCAS, TOTAL),  

.14 init-stidmCAS = {init-stidCAS ↪ explfndefCAS} in  

.15 mk- (statedefCAS, typedefmapCAS, inv-stidmCAS ⊔ init-stidmCAS)
.16 pre is-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 *TypeDefiton2TypeDefMap*. 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

Value Definitions

functions

61.0 $\text{ValueDefinitions2ValDefMap} : \text{OAS}^{\text{ValueDef}^+} \rightarrow (\text{CAS}^{\text{Pattern}} \xrightarrow{m} \text{CAS}^{\text{ValDef}})$

.1 $\text{ValueDefinitions2ValDefMap}(\text{vds}) \triangleq$
.2 $\text{merge} \{ \text{ValueDef2ValDefMap}(\text{vd}) \mid \text{vd} \in \text{elems vds} \}$
.3 $\text{pre let } \text{defs} = \{ \text{ValueDef2ValDefMap}(\text{vd}) \mid \text{vd} \in \text{elems vds} \} \text{ in}$
.4 $\forall \text{vdm} \in \text{defs} \cdot \forall \text{vdm}' \in \text{defs} \setminus \{\text{vdm}\} \cdot \text{dom vdm} \cap \text{dom vdm}' = \{ \}$

annotations

.3 The pre-condition ensures that all value patterns are unique.

end annotations ;

62.0 $\text{ValueDef2ValDefMap} : \text{OAS}^{\text{ValueDef}} \rightarrow (\text{CAS}^{\text{Pattern}} \xrightarrow{m} \text{CAS}^{\text{ValDef}})$

.1 $\text{ValueDef2ValDefMap}(\text{mk-OAS}^{\text{ValueDef}}(\text{pat}, \text{type}, \text{expr})) \triangleq$
.2 $\text{let } \text{pat}_{\text{CAS}} = \text{Pattern2Pattern}(\text{pat}, \{ \mapsto \}),$
.3 $\text{type}_{\text{CAS}} = \text{if type} = \text{nil} \text{ then nil}$
.4 $\text{else Type2Type}(\text{type}),$
.5 $\text{val}_{\text{CAS}} = \text{Expr2Expr}(\text{expr}, \{ \mapsto \}) \text{ in}$
.6 $\{ \text{pat}_{\text{CAS}} \mapsto \text{mk-CAS}^{\text{ValDef}}(\text{type}_{\text{CAS}}, \text{val}_{\text{CAS}}) \}$

Function Definitions

functions

63.0 $\text{FunctionDefinitions2ExplPolyFnDefMap} :$
.1 $\text{OAS}^{\text{FunctionDef}^+} \rightarrow$
.2 $(\text{CAS}^{\text{Id}} \xrightarrow{m} \text{CAS}^{\text{ExplPolyFnDef}}) \times (\text{CAS}^{\text{Id}} \xrightarrow{m} \text{CAS}^{\text{ExplPolyFnDef}})$
.3 $\text{FunctionDefinitions2ExplPolyFnDefMap}(\text{fds}) \triangleq$
.4 $\text{let } \text{defs} = \{ \text{FunctionDef2ExplPolyFnDefMap}(\text{fd}) \mid$
.5 $\text{fd} : \text{OAS}^{\text{ExplFnDef}} \cdot \text{fd} \in \text{elems fds} \wedge \text{IsPolymorphic}(\text{fd}) \} \text{ in}$
.6 $\text{mk-}(\text{merge} \{ \text{fnm} \mid \text{mk-}(\text{fnm}, -) \in \text{defs} \}, \text{merge} \{ \text{fnm}_q \mid \text{mk-}(-, \text{fnm}_q) \in \text{defs} \})$
.7 $\text{pre let } \text{defs} = \{ \text{FunctionDef2ExplPolyFnDefMap}(\text{fd}) \mid$
.8 $\text{fd} : \text{OAS}^{\text{ExplFnDef}} \cdot \text{fd} \in \text{elems fds} \wedge \text{IsPolymorphic}(\text{fd}) \} \text{ in}$
.9 $\forall \text{mk-}(\text{tdm}, \text{tdm}_q) \in \text{defs} \cdot$
.10 $\forall \text{mk-}(\text{tdm}', -) \in \text{defs} \setminus \{ \text{mk-}(\text{tdm}, \text{tdm}_q) \} \cdot$
.11 $\text{dom tdm} \cap \text{dom tdm}' = \{ \}$

annotations

- .7 The pre-condition ensures that all explicit polymorphic function identifiers are unique.
end annotations ;

64.0 *IsPolymorphic* : OAS‘FunctionDef → B

- .1 *IsPolymorphic* (fd) △
.2 fd.tpparms ≠ []

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 ;

65.0 *FunctionDef2ExplFnDefMap* :

- .1 OAS‘ExplFnDef → (CAS‘Id \xrightarrow{m} CAS‘ExplFnDef) × (CAS‘Id \xrightarrow{m} CAS‘ExplFnDef)
.2 *FunctionDef2ExplFnDefMap* (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 $\text{MakeLambda} : \text{OAS}'\text{ParametersList} \times \text{OAS}'\text{FnType} \times \text{OAS}'\text{Expr} \times \text{Env} \rightarrow \text{CAS}'\text{Lambda}$

.1 $\text{MakeLambda}(\text{parms}, \text{type}, \text{body}, \rho) \triangleq$
 .2 let $\text{patternlist} = \text{hd } \text{parms}$,
 .3 $\text{mk-}(-, \text{id}_{\text{CAS}}) = \text{GetUnusedId}()$,
 .4 $\text{type}_{\text{CAS}} = \text{Type2Type}(\text{type}.fndom)$,
 .5 $\text{par}_{\text{CAS}} = \text{mk-CAS}'\text{Par}(\text{id}_{\text{CAS}}, \text{type}_{\text{CAS}})$,
 .6 $\text{body}_{\text{CAS}} = \text{let } \text{bodyLet} = \begin{cases} \text{if len } \text{parms} = 1 \text{ then } \text{Expr2Expr}(\text{body}, \rho) \\ \text{else } \text{MakeLambda}(\text{tl } \text{parms}, \text{type}.fnrng, \text{body}, \rho), \end{cases}$
 .7 $\text{value}_{\text{Let}} = \text{mk-CAS}'\text{ValDef}(\text{type}_{\text{CAS}}, \text{id}_{\text{CAS}})$,
 .8 $\text{pat}_{\text{Let}} = \text{if len } \text{patternlist} = 1$
 .9 $\text{then Pattern2Pattern}(\text{hd } \text{patternlist}, \rho)$
 .10 $\text{else Pattern2Pattern}(\text{mk-OAS}'\text{TuplePattern}(\text{patternlist}), \rho)$,
 .11 $\text{vals}_{\text{Let}} = \{\text{pat}_{\text{Let}} \mapsto \text{value}_{\text{Let}}\} \text{ in}$
 .12 $\text{mk-CAS}'\text{LetExpr}(\text{vals}_{\text{Let}}, \{\mapsto\}, \{\mapsto\}, \text{body}_{\text{Let}}) \text{ in}$
 .13 $\text{mk-CAS}'\text{Lambda}(\text{par}_{\text{CAS}}, \text{body}_{\text{CAS}})$
 .14 pre $\text{NumberOfParsMatchesFnType}(\text{parms}, \text{type})$

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.

.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 $\text{NumberOfParsMatchesFnType} : \text{OAS}'\text{ParametersList} \times \text{OAS}'\text{FnType} \rightarrow \mathbb{B}$

.1 $\text{NumberOfParsMatchesFnType}(\text{parms}, \text{fntype}) \triangleq$
 .2 let $\text{FirstPairOK} = \text{cases } \text{len } \text{hd } \text{parms} :$
 .3 0 $\rightarrow \text{fntype}.fndom = \text{UNITTYPE}$,
 .4 1 $\rightarrow \neg \text{is-OAS}'\text{ProductType}(\text{fntype}.fndom)$,
 .5 others $\rightarrow \text{is-OAS}'\text{ProductType}(\text{fntype}.fndom) \wedge$
 .6 $\text{len } \text{hd } \text{parms} = \text{len } \text{fntype}.fndom.\text{factors}$
 .7 end in
 .8 if $\text{len } \text{parms} = 1$ then FirstPairOK
 .9 else $\text{FirstPairOK} \wedge$
 .10 $(\text{is-OAS}'\text{PartialFnType}(\text{fntype}.fnrng) \vee \text{is-OAS}'\text{TotalFnType}(\text{fntype}.fnrng)) \wedge$
 .11 $\text{NumberOfParsMatchesFnType}(\text{tl } \text{parms}, \text{fntype}.fnrng)$

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* \xrightarrow{m} *CAS‘ExplFnDef*) × (*CAS‘Id* \xrightarrow{m} *CAS‘ExplFnDef*)
.2 *FunctionDefinitions2ExplFnDefMap* (*fds*) △
.3 let *defs* = { *FunctionDef2ExplFnDefMap* (*fd*) |
.4 *fd* : *OAS‘ExplFnDef* · *fd* ∈ *elems fds* ∧ \neg *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* ∧ \neg *IsPolymorphic* (*fd*) } in
.8 \forall *mk-* (*tdm*, *tdm_q*) ∈ *defs* ·
.9 \forall *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 ;

70.0 *FunctionDef2ExplFnDefMap* :

.1 *OAS‘ExplFnDef* → (*CAS‘Id* \xrightarrow{m} *CAS‘ExplFnDef*) × (*CAS‘Id* \xrightarrow{m} *CAS‘ExplFnDef*)
.2 *FunctionDef2ExplFnDefMap* (*efd*) △
.3 let *mk-OAS‘ExplFnDef* (*id*, -, *type*, -, *parms*, *body*, *pre*) = *efd* in
.4 let *id_{CAS}* = *Id2Id* (*id*, { \mapsto }),
.5 *explfndef_{CAS}* = *ExplFnDef2ExplFnDef* (*efd*, { \mapsto }),
.6 *pre-id_{CAS}* = let *mk-OAS‘ValueId* (*pre-id*) = *id* in
.7 *Id2Id* (*mk-OAS‘PreId* (*pre-id*), { \mapsto }),
.8 *pre-explfndef_{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}*, [], *type_{pre-id}*,
.12 *pre-id_{OAS}*, *parms*, *pre*, nil) in
.13 *ExplFnDef2ExplFnDef* (*pre-efd*, { \mapsto }) in
.14 *mk-* ({*id_{CAS}* \mapsto *explfndef_{CAS}*}, {*pre-id_{CAS}* \mapsto *pre-explfndef_{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*) ∨
.17 *is-OAS‘InvId* (*id*) ∨ *is-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      OAS‘ExplFnDef × Env → (CAS‘ExplFnDef | CAS‘ExplPolyFnDef)
    .2      ExplFnDef2ExplFnDef (mk-OAS‘ExplFnDef (-, tpparms, type, -, parms, body, pre), ρ) △
    .3      let tpparmsCAS = OTypeVarList2TypeVarList (tpparms),
    .4      domtpCAS = Type2Type (type.fndom),
    .5      restpCAS = Type2Type (type.fnrng),
    .6      polytpCAS = FnType2FnType (type),
    .7      preCAS = OExpr2TrueOrExpr (pre, {→}),
    .8      bodyCAS = MakeLambda (parms, type, body, ρ),
    .9      kindCAS = if is-OAS‘PartialFnType (type) then PARTIAL
                      else TOTAL in
    .10     if tpparmsCAS = []
    .11     then mk-CAS‘ExplFnDef (domtpCAS, restpCAS, preCAS, bodyCAS, kindCAS)
    .12     else mk-CAS‘ExplPolyFnDef (tpparmsCAS, polytpCAS, preCAS, bodyCAS, kindCAS)
    .13

```

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 *tpparms*, and *restp* with *polytp* if this is the case.

end annotations ;

```

72.0   FunctionDefinitions2ImplPolyFnDefMap :
    .1      OAS‘FunctionDef+ →
    .2      (CAS‘Id  $\xrightarrow{m}$  CAS‘ImplPolyFnDef) × (CAS‘Id  $\xrightarrow{m}$  CAS‘ExplPolyFnDef)
    .3      FunctionDefinitions2ImplPolyFnDefMap (fds) △
    .4      let defs = {FunctionDef2ImplPolyFnDefMap (fd) |
    .5          fd : OAS‘ImplFnDef · fd ∈ elems fds ∧ IsPolymorphic (fd)} in
    .6          mk- (merge {fnm | mk- (fnm, -) ∈ defs}, merge {fnmq | mk- (-, fnmq) ∈ defs})
    .7      pre let defs = {FunctionDef2ImplPolyFnDefMap (fd) |
    .8          fd : OAS‘ImplFnDef · fd ∈ elems fds ∧ IsPolymorphic (fd)} in
    .9          ∀ mk- (tdm, tdmq) ∈ defs .
    .10         ∀ mk- (tdm', -) ∈ defs \ {mk- (tdm, tdmq)} .
    .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 $OAS'ImplFnDef \rightarrow (CAS'Id \xrightarrow{m} CAS'ImplPolyFnDef) \times (CAS'Id \xrightarrow{m} CAS'ExplPolyFnDef)$
 .2 *FunctionDef2ImplPolyFnDefMap* (*ifd*) \triangleq
 .3 let $mk\text{-}OAS'ImplFnDef$ (*id*, *tpparms*, *partps*, *residtype*, *fnpref*, *fnpost*) = *ifd* in
 .4 let id_{CAS} = $Id2Id(id, \{\mapsto\})$,
 .5 $implpolyfndef_{CAS}$ = $ImplFnDef2ImplFnDef(ifd)$,
 .6 $pre-id_{CAS}$ = let $mk\text{-}OAS'ValueId(pre-id) = id$ in
 .7 $Id2Id(mk\text{-}OAS'PreId(pre-id), \{\mapsto\})$,
 .8 $pre-implpolyfndef_{CAS}$ =
 .9 let $pre-id_{OAS} = \text{let } mk\text{-}OAS'ValueId(pre-id) = id \text{ in } mk\text{-}OAS'PreId(pre-id)$,
 .10 $mk\text{-}(parms}_{pre-id}, domtype_{pre-id}) =$
 .11 $SplitOPatTypePairList(partps)$,
 .12 $type_{pre-id} = mk\text{-}OAS'TotalFnType(domtype_{pre-id}, BOOLEAN)$,
 .13 $pre-ifd = mk\text{-}OAS'ExplFnDef(pre-id_{OAS}, tpparms, type_{pre-id},$
 .14 $pre-id_{OAS}, parms_{pre-id}, pre, nil)$ in
 .15 $ExplFnDef2ExplFnDef(pre-ifd, \{\mapsto\})$,
 .16 $post-id_{CAS} = \text{let } mk\text{-}OAS'ValueId(post-id) = id \text{ in }$
 .17 $Id2Id(mk\text{-}OAS'PostId(post-id), \{\mapsto\})$,
 .18 $post-implpolyfndef_{CAS}$ =
 .19 let $post-id_{OAS} = \text{let } mk\text{-}OAS'ValueId(post-id) = id \text{ in } mk\text{-}OAS'PostId(post-id)$,
 .20 $mk\text{-}(parms}_{post-id}, domtype_{post-id}) =$
 .21 $SplitOPatTypePairList(partps)$,
 .22 $type_{post-id} = mk\text{-}OAS'TotalFnType$
 .23 $(mk\text{-}OAS'ProductType([domtype}_{post-id}, residtype.type]), BOOLEAN)$,
 .24 $post-ifd = mk\text{-}OAS'ExplFnDef(post-id_{OAS}, [], type_{post-id}, post-id_{OAS},$
 .25 $parms_{post-id} \rightsquigarrow [residtype.id], post, nil)$ in
 .26 $ExplFnDef2ExplFnDef(post-ifd, \{\mapsto\})$ in
 .27 $mk\text{-}\{id_{CAS} \mapsto implpolyfndef_{CAS}\},$
 .28 $\{pre-id_{CAS} \mapsto pre-implpolyfndef_{CAS}\} \sqcup \{post-id_{CAS} \mapsto post-implpolyfndef_{CAS}\}$)
 .29 pre let $mk\text{-}OAS'ImplFnDef(id, -, -, mk\text{-}OAS'IdType(rid, -), -, -) = ifd$ in
 .30 $is\text{-}OAS'ValueId(id) \wedge is\text{-}OAS'ValueId(rid)$

annotations

- .8-.15 Generate an explicit function for the pre-condition of the function with the name *pre-id*.
- .18-.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 ;

```

74.0 ImplFnDef2ImplFnDef : OAS‘ImplFnDef → (CAS‘ImplFnDef | CAS‘ImplPolyFnDef)
.1 ImplFnDef2ImplFnDef (ifd) △
.2   let mk-OAS‘ImplFnDef (id, tpparms, partps, residtype, fnpref, fnpost) = ifd,
.3     mk-OAS‘IdType (rid, fnrng) = residtype in
.4   let mk-(patlist, fndom) = SplitOPatTypePairList (partps),
.5     mk-(idOAS, idCAS) = GetUnusedId (),
.6     tpparmsCAS = OTypeVarList2TypeVarList (tpparms),
.7     headingCAS = let fndomCAS = 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     parCAS = mk-CAS‘Par (idCAS, fndomCAS),
.14     restpCAS = Type2Type (fnrng),
.15     resumCAS = Id2Id (rid, {→}) in
.16     mk-CAS‘ImplFnHeading (parCAS, restpCAS, resumCAS),
.17   preCAS = if fnpref = nil then mk-CAS‘BoolLit (true)
.18     else MakeLetExpr (patlist, fnrng, idOAS, fnpref, {→}),
.19   postCAS = MakeLetExpr (patlist, fnrng, idOAS, fnpost, {→}) in
.20   if tpparmsCAS = []
.21   then mk-CAS‘ImplFnDef (headingCAS, preCAS, postCAS)
.22   else mk-CAS‘ImplPolyFnDef (tpparmsCAS, headingCAS, preCAS, postCAS)

```

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 ptle = 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, -) = ptle (i) in p | i ∈ inds ptle],
.6   types = [let mk-(-, t) = ptle (i) in t | i ∈ inds ptle] 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.

end annotations ;

```
76.0   FunctionDefinitions2ImplFnDefMap :
.1      OAS'FunctionDef+ →
.2      (CAS'Id  $\xrightarrow{m}$  CAS'ImplFnDef) × (CAS'Id  $\xrightarrow{m}$  CAS'ExplFnDef)
.3      FunctionDefinitions2ImplFnDefMap(fds)  $\triangleq$ 
.4      let defs = {FunctionDef2ImplFnDefMap(fd) |
.5          fd : OAS'ImplFnDef · fd ∈ elems fds  $\wedge \neg$  IsPolymorphic(fd)} in
.6          mk-(merge {fnm | mk-(fnm, -) ∈ defs}, merge {fnmq | mk-(-, fnmq) ∈ defs})
.7      pre let defs = {FunctionDef2ImplFnDefMap(fd) |
.8          fd : OAS'ImplFnDef · fd ∈ elems fds  $\wedge \neg$  IsPolymorphic(fd)} in
.9           $\forall$  mk-(tdm, tdmq) ∈ defs .
.10          $\forall$  mk-(tdm', -) ∈ defs \ {mk-(tdm, tdmq)} .
.11         dom tdm  $\cap$  dom tdm' = { }
```

annotations

.7 The pre-condition ensures that all implicit function identifiers are unique.

end annotations ;

```
77.0   FunctionDef2ImplFnDefMap :
.1      OAS'ImplFnDef → (CAS'Id  $\xrightarrow{m}$  CAS'ImplFnDef) × (CAS'Id  $\xrightarrow{m}$  CAS'ExplFnDef)
.2      FunctionDef2ImplFnDefMap(ifd)  $\triangleq$ 
.3      let mk-OAS'ImplFnDef(id, -, partps, mk-OAS'IdType(rid, rtype), fnpre, fnpost) = ifd in
.4      let idCAS = Id2Id(id, { $\mapsto$ }),
.5      implfndefCAS = ImplFnDef2ImplFnDef(ifd),
.6      pre-idCAS = let mk-OAS'ValueId(pre-id) = id in
.7          Id2Id(mk-OAS'PreId(pre-id), { $\mapsto$ }),
.8      pre-implfndefCAS =
.9          let pre-idOAS = let mk-OAS'ValueId(pre-id) = id in mk-OAS'PreId(pre-id),
.10         mk-(parmspre-id, domtypepre-id) =
.11             SplitOPatTypePairList(partps),
.12             typepre-id = mk-OAS'TotalFnType(domtypepre-id, BOOLEAN),
.13             pre-ifd = mk-OAS'ExplFnDef(pre-idOAS, [], typepre-id,
.14                 pre-idOAS, parmspre-id, fnpre, nil) in
.15             ExplFnDef2ExplFnDef(pre-ifd, { $\mapsto$ }),
.16             post-idCAS = let mk-OAS'ValueId(post-id) = id in
.17                 Id2Id(mk-OAS'PostId(post-id), { $\mapsto$ }),
.18             post-implfndefCAS =
.19                 let post-idOAS = let mk-OAS'ValueId(post-id) = id in mk-OAS'PostId(post-id),
.20                     mk-(parmspost-id, domtypepost-id) = SplitOPatTypePairList(partps),
.21                     typepost-id = mk-OAS'TotalFnType(mk-OAS'ProductType([domtypepost-id, rtype]),,
.22                         BOOLEAN),
.23                     post-ifd = mk-OAS'ExplFnDef(post-idOAS, [], typepost-id, post-idOAS,
.24                         parmspost-id ↗ [rid], fnpost, nil) in
.25             ExplFnDef2ExplFnDef(post-ifd, { $\mapsto$ }) in
.26             mk-({idCAS ↪ implfndefCAS},
.27                 {pre-idCAS ↪ pre-implfndefCAS} ⋃ {post-idCAS ↪ post-implfndefCAS})
```

```

.28   pre let mk-OAS'ImplFnDef (id, -, -, mk-OAS'IdType (rid, -, -, -) = ifd in
.29     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.
.18-25 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.

end annotations

```

Operation Definitions

functions

```

78.0  OperationDefinitions2ExplOpDefMap :
.1    OAS'OperationDef+ × CAS'TypeDef →
.2    (CAS'Id  $\xrightarrow{m}$  CAS'ExplOpDef) × (CAS'Id  $\xrightarrow{m}$  CAS'ExplFnDef)

.3  OperationDefinitions2ExplOpDefMap (ods,  $\sigma$ )  $\triangleq$ 
.4    let defs = {OperationDef2ExplOpDefMap (od,  $\sigma$ ) |
.5      od : OAS'ExplOprtDef · od ∈ elems ods} in
.6      mk- (merge {op | mk- (op, -) ∈ defs}, merge {opq | mk- (-, opq) ∈ defs})
.7  pre let defs = {OperationDef2ExplOpDefMap (od,  $\sigma$ ) |
.8    od : OAS'ExplOprtDef · od ∈ elems ods} in
.9     $\forall$  mk-(tdm, tdmq) ∈ defs .
.10    $\forall$  mk-(tdm', -) ∈ defs \ {mk-(tdm, tdmq)} · dom tdm ∩ dom tdm' = {}

```

annotations

```

.7 The pre-condition ensures that all explicit operation identifiers are unique.

end annotations ;

```

```

79.0   OperationDef2ExplOpDefMap :
    .1      OAS`ExplOprtDef × CAS`TypeDef →
    .2      (CAS`Id  $\xrightarrow{m}$  CAS`ExplOpDef) × (CAS`Id  $\xrightarrow{m}$  CAS`ExplFnDef)

    .3   OperationDef2ExplOpDefMap (eod,  $\sigma$ )  $\triangleq$ 
    .4     let mk-OAS`ExplOprtDef (id, optype, -, params, body, oppre) = eod in
    .5     let idCAS          = Id2Id (id, { $\mapsto$ }),
    .6     explopdefCAS       = ExplOprtDef2ExplOpDef (eod,  $\sigma$ ),
    .7     pre-idCAS         = let mk-OAS`ValueId (pre-id) = id in
    .8           Id2Id (mk-OAS`PreId (pre-id), { $\mapsto$ }),
    .9     pre-explopdefCAS =
    .10       let pre-idOAS = let mk-OAS`ValueId (pre-id) = id in mk-OAS`PreId (pre-id),
    .11         typepre-id = if optype.dom = UNITTYPE
    .12           then mk-OAS`TotalFnType ( $\sigma$ , BOOLEAN)
    .13           else mk-OAS`TotalFnType
    .14             (mk-OAS`ProductType ([optype.dom,  $\sigma$ ]), BOOLEAN),
    .15     parmspre-id =
    .16       let scomps = [mk-OAS`PatternId (mk-OAS`ValueId ( $\sigma$ .shape.fields(i).sel)) |
    .17         i ∈ inds  $\sigma$ .shape.fields],
    .18         mk-OAS`ValueId (tag) =  $\sigma$ .shape.id in
    .19         if params = [] then mk-OAS`RecordPattern (tag, scomps)
    .20         else mk-OAS`TuplePattern ([params, mk-OAS`RecordPattern (tag, scomps)]),
    .21     bodypre-id = if oppre = nil then mk-OAS`BoolLit (true) else oppre,
    .22     pre-eod    = mk-OAS`ExplFnDef (pre-idOAS, [], typepre-id, pre-idOAS,
    .23       parmspre-id, bodypre-id, nil) in
    .24     ExplFnDef2ExplFnDef (pre-eod, { $\mapsto$ }) in
    .25     mk-{idCAS  $\mapsto$  explopdefCAS}, {pre-idCAS  $\mapsto$  pre-explopdefCAS})
    .26   pre let mk-OAS`ExplOprtDef (id, optype, id_repeated, params, -, -) = eod in
    .27     id = id_repeated  $\wedge$  is-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 ;

```

80.0   NumberOfParsMatchesOpType : OAS`Parameters × OAS`OpType →  $\mathbb{B}$ 
    .1      NumberOfParsMatchesOpType (params, mk-OAS`OpType (opdom, -))  $\triangleq$ 
    .2      cases len params :
    .3        0 → opdom = UNITTYPE,
    .4        1 →  $\neg$ is-OAS`ProductType (opdom),
    .5        others → is-OAS`ProductType (opdom)  $\wedge$ 
    .6          let mk-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 $\text{ExplOpDef2ExplOpDef} : \text{OAS}^{\text{ExplOpDef}} \times \text{CAS}^{\text{TypeDef}} \rightarrow \text{CAS}^{\text{ExplOpDef}}$

- .1 $\text{ExplOpDef2ExplOpDef}(\text{eod}, \sigma) \triangleq$
- .2 $\text{let } \text{mk-}\text{OAS}^{\text{ExplOpDef}}(\text{id}, \text{optype}, -, \text{params}, \text{body}, \text{oppre}) = \text{eod} \text{ in}$
- .3 $\text{let } \text{heading}_{\text{CAS}} = \text{MakeOpHeading}(\text{optype}, \text{oppre}, [], \sigma),$
- .4 $\text{body}_{\text{CAS}} = \text{Stmt2Stmt}(\text{body}) \text{ in}$
- .5 $\text{mk-}\text{CAS}^{\text{ExplOpDef}}(\text{heading}_{\text{CAS}}, \text{body}_{\text{CAS}})$

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 $\text{MakeOpHeading} :$
 $\text{OAS}^{\text{OpType}} \times [\text{OAS}^{\text{Expr}}] \times \text{OAS}^{\text{VarInf}^*} \times \text{CAS}^{\text{TypeDef}} \rightarrow \text{CAS}^{\text{OpHeading}}$

- .1 $\text{MakeOpHeading}(\text{mk-}\text{OAS}^{\text{OpType}}(\text{dom}, \text{rng}), \text{pre}, \text{extrdwr}, \sigma) \triangleq$
- .2 $\text{let } \text{mk-}(-, \text{id}_{\text{CAS}}) = \text{GetUnusedId}(),$
- .3 $\text{par}_{\text{CAS}} = \text{mk-}\text{CAS}^{\text{Par}}(\text{id}_{\text{CAS}}, \text{Type2Type}(\text{dom})),$
- .4 $\text{restp}_{\text{CAS}} = \text{Type2Type}(\text{rng}),$
- .5 $\rho = \{\text{mk-}\text{OAS}^{\text{ValueId}}(\sigma.\text{shape}.fields(i).\text{sel}) \mapsto$
- .6 $\text{mk-}\text{CAS}^{\text{OldId}}(\sigma.\text{shape}.fields(i).\text{sel}) \mid$
- .7 $i \in \text{inds } \text{stype}.\text{shape}.fields\},$
- .8 $\text{prec}_{\text{CAS}} = \text{OExpr2TrueOrExpr}(\text{pre}, \rho),$
- .9 $\text{ext}_{\text{CAS}} = \text{VarInfs2ExtVarInfs}(\text{extrdwr}, \sigma) \text{ in}$
- .10 $\text{mk-}\text{CAS}^{\text{OpHeading}}(\text{par}_{\text{CAS}}, \text{restp}_{\text{CAS}}, \text{prec}_{\text{CAS}}, \text{ext}_{\text{CAS}});$

83.0 $\text{VarInfs2ExtVarInfs} : \text{OAS}^{\text{VarInf}^*} \times \text{CAS}^{\text{TypeDef}} \rightarrow \text{CAS}^{\text{ExtVarInf-set}}$

- .1 $\text{VarInfs2ExtVarInfs}(\text{extrdwr}, \sigma) \triangleq$
- .2 $\bigcup \{\{\text{let } \text{mk-}\text{OAS}^{\text{VarInf}}(\text{mode}, \text{vars}, \text{type}) = \text{extrdwr}(j),$
- .3 $\text{rest} = \text{Id2Id}(\text{vars}(i), \{\mapsto\}),$
- .4 $\text{tp} = \text{if } \text{type} = \text{nil}$
- .5 $\text{then let } \text{tp} : \text{CAS}^{\text{Type}} \text{ be st } \text{mk-}\text{CAS}^{\text{Field}}(\text{rest}, \text{tp}) \in \text{elems } \sigma.\text{shape}.fields \text{ in } \text{tp}$
- .6 $\text{else } \text{Type2Type}(\text{type}) \text{ in}$
- .7 $\text{mk-}\text{CAS}^{\text{ExtVarInf}}(\text{mode}, \text{rest}, \text{tp}) \mid i \in \text{inds } \text{extrdwr}(j).\text{vars} \} \mid j \in \text{inds } \text{extrdwr}\}$
- .8 $\text{pre } \forall \text{id} \in \bigcup \{\{\text{extrdwr}(j) \cdot \text{vars}(i) \mid i \in \text{inds } \text{extrdwr}(j) \cdot \text{vars}\} \mid j \in \text{inds } \text{extrdwr}\} .$
- .9 $\text{is-}\text{OAS}^{\text{ValueId}}(\text{id})$

annotations

83.0 When this function is called with $\text{OAS}^{\text{VarInf}^*}$ 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.

.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 $\text{OperationDefinitions2ImplOpDefMap} :$
 $\text{OAS}^{\text{OperationDef}^+} \times \text{CAS}^{\text{TypeDef}} \rightarrow$
 $(\text{CAS}^{\text{Id}} \xrightarrow{m} \text{CAS}^{\text{ImplOpDef}}) \times (\text{CAS}^{\text{Id}} \xrightarrow{m} \text{CAS}^{\text{ExplFnDef}})$

- .1 $\text{OperationDefinitions2ImplOpDefMap}(\text{ods}, \sigma) \triangleq$
- .2 $\text{let } \text{defs} = \{\text{OperationDef2ImplOpDefMap}(\text{od}, \sigma) \mid$
- .3 $\text{od} : \text{OAS}^{\text{ImplOpDef}} \cdot \text{od} \in \text{elems } \text{ods}\} \text{ in}$
- .4 $\text{mk-}(\text{merge } \{\text{op} \mid \text{mk-}(\text{op}, -) \in \text{defs}\}, \text{merge } \{\text{op}_q \mid \text{mk-}(-, \text{op}_q) \in \text{defs}\})$

```

.7   pre let defs = {OperationDef2ImplOpDefMap(od,  $\sigma$ ) |
.8     od : OAS‘ImplOprtDef · od ∈ elems ods} in
.9      $\forall \text{mk-}(tdm, tdm_q) \in \text{defs} \cdot$ 
.10     $\forall \text{mk-}(tdm', -) \in \text{defs} \setminus \{\text{mk-}(tdm, tdm_q)\} \cdot$ 
.11      dom tdm ∩ dom tdm' = { }

```

annotations

.7 The pre-condition ensures that all implicit operation identifiers are unique.
end annotations ;

```

85.0   OperationDef2ImplOpDefMap :
.1     OAS‘ImplOprtDef × CAS‘TypeDef →
.2     (CAS‘Id  $\xrightarrow{m}$  CAS‘ImplOpDef) × (CAS‘Id  $\xrightarrow{m}$  CAS‘ExplFnDef)
.3   OperationDef2ImplOpDefMap(iod,  $\sigma$ )  $\triangleq$ 
.4     let mk-OAS‘ImplOprtDef(id, params, residtype, opext, oppire, oppost, excps) = iod,
.5        $\rho = \{\text{mk-OAS‘ValueId}(\sigma.\text{shape}.\text{fields}(i).\text{sel}) \mapsto \text{GetUnusedId}() \mid$ 
.6          $i \in \text{inds } \sigma.\text{shape}.\text{fields}\}$  in
.7     let idCAS = Id2Id(id, { $\mapsto$ }),
.8     implopdefCAS = ImplOprtDef2ImplOpDef(iod,  $\sigma$ ,  $\rho$ ),
.9     mk-(patlist, pattype) = SplitOPatTypePairList(params),
.10    mk-(pre-idOAS, pre-idCAS,
.11    post-idOAS, post-idCAS) = let mk-OAS‘ValueId(idOAS) = id in
.12      mk-(mk-OAS‘PreId(idOAS), Id2Id(pre-idOAS, { $\mapsto$ })),
.13      mk-OAS‘PostId(idOAS), Id2Id(post-idOAS, { $\mapsto$ })),
.14    scomps = [mk-OAS‘PatternId(mk-OAS‘ValueId(σ.shape.fields(i).sel)) |
.15       $i \in \text{inds } \sigma.\text{shape}.\text{fields}$ ],
.16    pre-implopdefCAS =
.17      let pre-idOAS = let mk-OAS‘ValueId(pre-id) = id in mk-OAS‘PreId(pre-id),
.18      typepre-id = if pattype = nil then mk-OAS‘TotalFnType( $\sigma$ , BOOLEAN)
.19        else mk-OAS‘TotalFnType(mk-OAS‘ProductType([pattype,  $\sigma$ ]), BOOLEAN),
.20      parmspre-id = let mk-OAS‘ValueId(tag) =  $\sigma.\text{shape}.\text{id}$  in
.21        if parms = [] then mk-OAS‘RecordPattern(tag, scomps)
.22        else mk-OAS‘TuplePattern([parms, mk-OAS‘RecordPattern(tag, scomps)]),
.23      bodypre-id = if oppire = nil then mk-OAS‘BoolLit(true) else oppire,
.24      pre-iod = mk-OAS‘ExplFnDef(pre-idOAS, [], typepre-id, pre-idOAS,
.25        parmspre-id, bodypre-id, nil) in
.26      ExplFnDef2ExplFnDef(pre-iod, { $\mapsto$ }),
.27    post-implopdefCAS =
.28      let post-idOAS = let mk-OAS‘ValueId(post-id) = id in mk-OAS‘PostId(post-id),
.29      typepost-id = let rtype = if residtype = nil then [] else [residtype.type] in
.30        mk-OAS‘TotalFnType(mk-OAS‘ProductType([pattype,  $\sigma$ ])  $\curvearrowright$ 
.31          rtype), BOOLEAN),
.32      parmspost-id = let mk-OAS‘ValueId(tag) =  $\sigma.\text{shape}.\text{id}$  in
.33        let scomps-old = [ $\rho(scomp(i)) \mid i \in \text{inds } scomp$ ],
.34        parms' = parms  $\curvearrowright$  [mk-OAS‘RecordPattern(tag, scomps)]  $\curvearrowright$ 
.35          [mk-OAS‘RecordPattern(tag, scomps-old)] in
.36        if residtype = nil then parms'
.37        else parms'  $\curvearrowright$  [mk-OAS‘PatternId(residtype.id)],
.38      post-iod = mk-OAS‘ExplFnDef(post-idOAS, [], typepost-id, post-idOAS,
.39        parmspost-id, oppost, nil) in
.40      ExplFnDef2ExplFnDef(post-iod,  $\rho$ ) in
.41    mk-{idCAS  $\mapsto$  implopdefCAS},
.42    {pre-idCAS  $\mapsto$  pre-implopdefCAS}  $\sqcup$  {post-idCAS  $\mapsto$  post-implopdefCAS}

```

.43 pre let $\text{mk-OAS}^{\text{'}}\text{ImplOprtDef}(id, -, \text{mk-OAS}^{\text{'}}\text{IdType}(rid, -), -, -, -, -) = iod$ in
.44 $\text{is-OAS}^{\text{'}}\text{ValueId}(id) \wedge \text{is-OAS}^{\text{'}}\text{ValueId}(rid)$

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 ;

86.0 $\text{ImplOprtDef2ImplOpDef} : \text{OAS}^{\text{'}}\text{ImplOprtDef} \times \text{CAS}^{\text{'}}\text{TypeDef} \times \text{Env} \rightarrow \text{CAS}^{\text{'}}\text{ImplOpDef}$

.1 $\text{ImplOprtDef2ImplOpDef}(iod, \sigma, \rho) \triangleq$
.2 let $\text{mk-OAS}^{\text{'}}\text{ImplOprtDef}(id, \text{params}, \text{residtype}, \text{opext}, \text{oppre}, \text{oppst}, \text{excps}) = iod$ in
.3 let $\text{mk-(patlist, pattype)} = \text{SplitOPatTypePairList}(\text{params})$,
.4 $\text{optype} = \text{if residtype} = \text{nil}$
.5 then $\text{mk-OAS}^{\text{'}}\text{OpType}(\text{pattype}, \text{UNITTYPE})$
.6 else let $\text{mk-OAS}^{\text{'}}\text{IdType}(-, \text{type}) = \text{residtype}$ in
.7 $\text{mk-OAS}^{\text{'}}\text{OpType}(\text{pattype}, \text{type})$,
.8 $\text{heading}_{\text{CAS}} = \text{MakeOpHeading}(\text{optype}, \text{oppre}, \text{excps}, \sigma)$,
.9 $\text{body}_{\text{CAS}} = \text{MakePostDisjExcps}(\text{residtype}, \text{oppst}, \text{excps}, \text{patlist}, \rho)$ in
.10 $\text{mk-CAS}^{\text{'}}\text{ImplOpDef}(\text{heading}_{\text{CAS}}, \text{body}_{\text{CAS}})$

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 $\text{MakePostDisjExcps} :$

.1 $[\text{OAS}^{\text{'}}\text{IdType}] \times \text{OAS}^{\text{'}}\text{Expr} \times \text{OAS}^{\text{'}}\text{OExceptions} \times \text{OAS}^{\text{'}}\text{ParameterTypes} \times \text{Env} \rightarrow$
.2 $\text{CAS}^{\text{'}}\text{OpPost}$

.3 $\text{MakePostDisjExcps}(\text{post}, \text{excps}, \text{patlist}, \rho) \triangleq$
.4 let $\text{vid}_{\text{CAS}} = \text{if residtype} = \text{nil}$ then nil
.5 else let $\text{mk-OAS}^{\text{'}}\text{IdType}(id, -) = \text{residtype}$,
.6 $\text{id}_{\text{CAS}} = \text{Id2Id}(id, \rho)$ in
.7 $\text{mk-CAS}^{\text{'}}\text{ValueId}(\text{id}_{\text{CAS}})$,
.8 $\text{excps}_{\text{CAS}} = \text{MakeDisjExcps}(\text{excps}, \rho)$,
.9 $\text{post}_{\text{CAS}} = \text{Expr2Expr}(\text{post}, \{\mapsto\})$,
.10 $\text{post}_{\text{CAS-excps}} = \text{if } \text{excps}_{\text{CAS}} = \text{nil}$ then post_{CAS}
.11 else $\text{mk-CAS}^{\text{'}}\text{BinaryExpr}(\text{post}_{\text{CAS}}, \text{OR}, \text{excps}_{\text{CAS}})$,
.12 $\text{postcond}_{\text{CAS}} = \text{if patlist} = []$
.13 then $\text{post}_{\text{CAS-excps}}$
.14 else let $\text{pat} = \text{if len patlist} = 1$
.15 then $\text{Pattern2Pattern}(\text{hd patlist}, \{\mapsto\})$
.16 else let $\text{tpat} = \text{mk-OAS}^{\text{'}}\text{TuplePattern}(\text{patlist})$ in
.17 $\text{TuplePattern2TuplePattern}(\text{tpat}, \{\mapsto\})$,
.18 $v_{\text{CAS}} = \text{ValueDef2ValDefMap}(\text{mk-OAS}^{\text{'}}\text{ValueDef}(\text{pat}, \text{pattype}, \text{nid}))$ in
.19 $\text{mk-CAS}^{\text{'}}\text{LetExpr}(v_{\text{CAS}}, \{\mapsto\}, \{\mapsto\}, \text{post}_{\text{CAS-excps}})$ in
.20 $\text{mk-CAS}^{\text{'}}\text{OpPost}(\text{vid}_{\text{CAS}}, \text{postcond}_{\text{CAS}})$

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 $post_{CAS_excps}$; otherwise generate a value definition from $patlist$ and combine with $post_{CAS_excps}$ to form a let expression.

end annotations ;

88.0 $MakeDisjExcps : OAS^{\prime}OExceptions \times Env \rightarrow [CAS^{\prime}Expr]$

- .1 $MakeDisjExcps(excps, \rho) \triangleq$
- .2 $\text{let } mk\text{-}OAS^{\prime}\text{Error}(-, cond, action) = \text{hd } excps \text{ in}$
- .3 $\text{let } left_{CAS} = Expr2Expr(cond, \rho),$
- .4 $right_{CAS} = Expr2Expr(action, \{\mapsto\}),$
- .5 $error_{CAS} = mk\text{-}CAS^{\prime}\text{BinaryExpr}(left_{CAS}, \text{AND}, right_{CAS}) \text{ in}$
- .6 $\text{if len } excps = 1$
- .7 $\text{then } error_{CAS}$
- .8 $\text{else } mk\text{-}CAS^{\prime}\text{BinaryExpr}(error_{CAS}, \text{OR}, MakeDisjExcps(\text{tl } excps, \rho))$

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

8.3.3 Expressions

functions

89.0 $OExpr2TrueOrExpr : [OAS^{\prime}Expr] \times Env \rightarrow CAS^{\prime}Expr$

- .1 $OExpr2TrueOrExpr(expr, \rho) \triangleq$
- .2 $\text{if } expr = \text{nil} \text{ then } mk\text{-}CAS^{\prime}\text{BoolLit}(\text{true})$
- .3 $\text{else } Expr2Expr(expr, \rho);$

90.0 $\text{Expr2Expr} : [\text{OAS}^{\text{Expr}}] \times \text{Env} \rightarrow [\text{CAS}^{\text{Expr}}]$

.1 $\text{Expr2Expr}(\text{expr}, \rho) \triangleq$

.2 cases expr :

.3 $\text{mk-OAS}^{\text{BracketedExpr}}(\text{expr}) \rightarrow \text{Expr2Expr}(\text{expr}, \rho),$
 .4 $\text{mk-OAS}^{\text{LetExpr}}(-, -) \rightarrow \text{LetExpr2LetExpr}(\text{expr}, \rho),$
 .5 $\text{mk-OAS}^{\text{LetBeSTExpr}}(-, -, -) \rightarrow \text{LetBeSTExpr2LetBeSTExpr}(\text{expr}, \rho),$
 .6 $\text{mk-OAS}^{\text{DefExpr}}(-, -) \rightarrow \text{DefExpr2DefExpr}(\text{expr}, \rho),$
 .7 $\text{mk-OAS}^{\text{IfExpr}}(-, -, -, -) \rightarrow \text{IfExpr2IfExpr}(\text{expr}, \rho),$
 .8 $\text{mk-OAS}^{\text{CasesExpr}}(-, -, -) \rightarrow \text{CasesExpr2CasesExpr}(\text{expr}, \rho),$
 .9 $\text{mk-OAS}^{\text{PrefixExpr}}(-, -) \rightarrow \text{PrefixExpr2UnaryExpr}(\text{expr}, \rho),$
 .10 $\text{mk-OAS}^{\text{MapInverseExpr}}(-) \rightarrow \text{MapInverseExpr2UnaryExpr}(\text{expr}, \rho),$
 .11 $\text{mk-OAS}^{\text{BinaryExpr}}(-, -, -) \rightarrow \text{BinaryExpr2BinaryExpr}(\text{expr}, \rho),$
 .12 $\text{mk-OAS}^{\text{AllExpr}}(-, -) \rightarrow \text{AllExpr2AllOrExistsExpr}(\text{expr}, \rho),$
 .13 $\text{mk-OAS}^{\text{ExistsExpr}}(-, -) \rightarrow \text{ExistsExpr2AllOrExistsExpr}(\text{expr}, \rho),$
 .14 $\text{mk-OAS}^{\text{ExistsUniqueExpr}}(-, -) \rightarrow \text{ExistsUniqueExpr2ExistsUniqueExpr}(\text{expr}, \rho),$
 .15 $\text{mk-OAS}^{\text{IotaExpr}}(-, -) \rightarrow \text{IotaExpr2IotaExpr}(\text{expr}, \rho),$
 .16 $\text{mk-OAS}^{\text{SetEnumeration}}(-) \rightarrow \text{SetEnumeration2SetEnumeration}(\text{expr}, \rho),$
 .17 $\text{mk-OAS}^{\text{SetComprehension}}(-, -, -) \rightarrow \text{SetComprehension2SetComprehension}(\text{expr}, \rho),$
 .18 $\text{mk-OAS}^{\text{SetRange}}(-, -) \rightarrow \text{SetRange2SetRange}(\text{expr}, \rho),$
 .19 $\text{mk-OAS}^{\text{SeqEnumeration}}(-) \rightarrow \text{SeqEnumeration2SeqEnumeration}(\text{expr}, \rho),$
 .20 $\text{mk-OAS}^{\text{SeqComprehension}}(-, -, -) \rightarrow \text{SeqComprehension2SeqComprehension}(\text{expr}, \rho),$
 .21 $\text{mk-OAS}^{\text{SubSequence}}(-, -, -) \rightarrow \text{SubSequence2SubSequence}(\text{expr}, \rho),$
 .22 $\text{mk-OAS}^{\text{MapEnumeration}}(-) \rightarrow \text{MapEnumeration2MapEnumeration}(\text{expr}, \rho),$
 .23 $\text{mk-OAS}^{\text{MapComprehension}}(-, -, -) \rightarrow \text{MapComprehension2MapComprehension}(\text{expr}, \rho),$
 .24 $\text{mk-OAS}^{\text{TupleConstructor}}(-) \rightarrow \text{TupleConstructor2TupleConstructor}(\text{expr}, \rho),$
 .25 $\text{mk-OAS}^{\text{RecordConstructor}}(-, -) \rightarrow \text{RecordConstructor2RecordConstructor}(\text{expr}, \rho),$
 .26 $\text{mk-OAS}^{\text{RecordModifier}}(-, -) \rightarrow \text{RecordModifier2RecordModifier}(\text{expr}, \rho),$
 .27 $\text{mk-OAS}^{\text{Apply}}(-, -) \rightarrow \text{Apply2Apply}(\text{expr}, \rho),$
 .28 $\text{mk-OAS}^{\text{FieldSelect}}(-, -) \rightarrow \text{FieldSelect2FieldSelect}(\text{expr}, \rho),$
 .29 $\text{mk-OAS}^{\text{FctTypeInst}}(-, -) \rightarrow \text{FctTypeInst2FctTypeInst}(\text{expr}, \rho),$
 .30 $\text{mk-OAS}^{\text{Lambda}}(-, -) \rightarrow \text{Lambda2Lambda}(\text{expr}, \rho),$
 .31 $\text{mk-OAS}^{\text{IsDefTypeExpr}}(-, -) \rightarrow \text{IsDefTypeExpr2IsExpr}(\text{expr}, \rho),$
 .32 $\text{mk-OAS}^{\text{IsBasicTypeExpr}}(-, -) \rightarrow \text{IsBasicTypeExpr2IsExpr}(\text{expr}, \rho),$
 .33 $\text{mk-OAS}^{\text{NumLit}}(\text{val}) \rightarrow \text{mk-CAS}^{\text{NumLit}}(\text{val}),$
 .34 $\text{mk-OAS}^{\text{BoolLit}}(\text{val}) \rightarrow \text{mk-CAS}^{\text{BoolLit}}(\text{val}),$
 .35 $(\text{is-OAS}^{\text{NilLit}}(\text{expr})) \rightarrow \text{mk-CAS}^{\text{NilLit}}(),$
 .36 $\text{mk-OAS}^{\text{CharLit}}(\text{val}) \rightarrow \text{mk-CAS}^{\text{CharLit}}(\text{val}),$
 .37 $\text{mk-OAS}^{\text{TextLit}}(\text{val}) \rightarrow \text{mk-CAS}^{\text{TextLit}}([mk-CAS^{\text{CharLit}}(\text{val}(i)) \mid i \in \text{inds val}]),$
 .38 $\text{mk-OAS}^{\text{QuoteLit}}(\text{val}) \rightarrow \text{mk-CAS}^{\text{QuoteLit}}(\text{val}),$
 .39 $\text{mk-OAS}^{\text{Name}}(\text{name}) \rightarrow \text{Id2Id}(\text{name}, \rho),$
 .40 $\text{mk-OAS}^{\text{OldName}}(\text{name}) \rightarrow \text{OldName2OldId}(\text{name}),$
 .41 others
 .42 $\rightarrow \text{nil}$
 .43 end

annotations

.1 See section 8.1.3 for a description of the use of argument ρ .

end annotations ;

91.0 $\text{IsExpr} : \text{OAS}^{\text{Expr}} \rightarrow \mathbb{B}$

- .1 $\text{IsExpr}(\text{expr}) \triangleq$
- .2 $\text{is-OAS}^{\text{BracketedExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{LetExpr}}(\text{expr}) \vee$
- .3 $\text{is-OAS}^{\text{LetBeSTEExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{DefExpr}}(\text{expr}) \vee$
- .4 $\text{is-OAS}^{\text{IfExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{CasesExpr}}(\text{expr}) \vee$
- .5 $\text{is-OAS}^{\text{PrefixExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{MapInverseExpr}}(\text{expr}) \vee$
- .6 $\text{is-OAS}^{\text{BinaryExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{AllExpr}}(\text{expr}) \vee$
- .7 $\text{is-OAS}^{\text{ExistsExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{ExistsUniqueExpr}}(\text{expr}) \vee$
- .8 $\text{is-OAS}^{\text{IotaExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{SetEnumeration}}(\text{expr}) \vee$
- .9 $\text{is-OAS}^{\text{SetComprehension}}(\text{expr}) \vee \text{is-OAS}^{\text{SetRange}}(\text{expr}) \vee$
- .10 $\text{is-OAS}^{\text{SeqEnumeration}}(\text{expr}) \vee \text{is-OAS}^{\text{SeqComprehension}}(\text{expr}) \vee$
- .11 $\text{is-OAS}^{\text{SubSequence}}(\text{expr}) \vee \text{is-OAS}^{\text{MapEnumeration}}(\text{expr}) \vee$
- .12 $\text{is-OAS}^{\text{MapComprehension}}(\text{expr}) \vee \text{is-OAS}^{\text{TupleConstructor}}(\text{expr}) \vee$
- .13 $\text{is-OAS}^{\text{RecordConstructor}}(\text{expr}) \vee \text{is-OAS}^{\text{RecordModifier}}(\text{expr}) \vee$
- .14 $\text{is-OAS}^{\text{Apply}}(\text{expr}) \vee \text{is-OAS}^{\text{FieldSelect}}(\text{expr}) \vee$
- .15 $\text{is-OAS}^{\text{FctTypeInst}}(\text{expr}) \vee \text{is-OAS}^{\text{Lambda}}(\text{expr}) \vee$
- .16 $\text{is-OAS}^{\text{IsDefTypeExpr}}(\text{expr}) \vee \text{is-OAS}^{\text{IsBasicTypeExpr}}(\text{expr}) \vee$
- .17 $\text{is-OAS}^{\text{NumLit}}(\text{expr}) \vee \text{is-OAS}^{\text{BoolLit}}(\text{expr}) \vee$
- .18 $\text{is-OAS}^{\text{NilLit}}(\text{expr}) \vee \text{is-OAS}^{\text{CharLit}}(\text{expr}) \vee$
- .19 $\text{is-OAS}^{\text{TextLit}}(\text{expr}) \vee \text{is-OAS}^{\text{QuoteLit}}(\text{expr}) \vee$
- .20 $\text{is-OAS}^{\text{Name}}(\text{expr}) \vee \text{is-OAS}^{\text{OldName}}(\text{expr})$

Bracketed Expressions

This section is intentionally left empty.

Local Binding Expressions

functions

92.0 $\text{LetExpr2LetExpr} : \text{OAS}^{\text{LetExpr}} \times \text{Env} \rightarrow \text{CAS}^{\text{LetExpr}}$

- .1 $\text{LetExpr2LetExpr}(\text{mk-OAS}^{\text{LetExpr}}(\text{localdefs}, \text{body}), \rho) \triangleq$
- .2 $\text{let } \text{vals}_{\text{CAS}} = \text{merge} \{ \text{ValueDefinitions2ValDefMap}(\text{vds}) \mid$
 $\text{mk-OAS}^{\text{ValueDefinitions}}(\text{vds}) \in \text{elems localdefs} \},$
- .3 $\text{mk-}(\text{explfns}_{\text{CAS}}, -) = \{ \text{FunctionDef2ExplFnDefMap}(\text{fd}) \mid$
 $\text{fd: OAS}^{\text{ExplFnDef}} \cdot \text{fd} \in \text{elems localdefs} \wedge \neg \text{IsPolymorphic}(\text{fd}) \},$
- .4 $\text{mk-}(\text{implfns}_{\text{CAS}}, -) = \{ \text{FunctionDef2ImplFnDefMap}(\text{fd}) \mid$
 $\text{fd: OAS}^{\text{ImplFnDef}} \cdot \text{fd} \in \text{elems localdefs} \wedge \neg \text{IsPolymorphic}(\text{fd}) \},$
- .5 $\text{in}_{\text{CAS}} = \text{Expr2Expr}(\text{body}, \rho) \text{ in }$
- .6 $\text{mk-CAS}^{\text{LetExpr}}(\text{vals}_{\text{CAS}}, \text{explfns}_{\text{CAS}}, \text{implfns}_{\text{CAS}}, \text{in}_{\text{CAS}})$
- .7 $\text{pre let } \text{mk-}(\text{explfns}_{\text{CAS}}, -) = \{ \text{FunctionDef2ExplFnDefMap}(\text{fd}) \mid$
 $\text{fd: OAS}^{\text{ExplFnDef}} \cdot \text{fd} \in \text{elems localdefs} \wedge \neg \text{IsPolymorphic}(\text{fd}) \},$
- .8 $\text{mk-}(\text{implfns}_{\text{CAS}}, -) = \{ \text{FunctionDef2ImplFnDefMap}(\text{fd}) \mid$
 $\text{fd: OAS}^{\text{ImplFnDef}} \cdot \text{fd} \in \text{elems localdefs} \wedge \neg \text{IsPolymorphic}(\text{fd}) \} \text{ in }$
- .9 $\forall \text{id} \in \text{dom explfns}_{\text{CAS}} \cup \text{dom implfns}_{\text{CAS}} \cdot \text{is-OAS}^{\text{ValueId}}(\text{id})$

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 ;

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}*)

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* (*testElsifExpr*, *consElsifExpr*) = hd *elsifaltn* in
 .6 *IfExpr2IfExpr* (mk-OAS‘*IfExpr* (*testElsifExpr*, *consElsifExpr*,
 .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* : $OAS^{\epsilon} Pattern \times OAS^{\epsilon} Expr \times Env \rightarrow CAS^{\epsilon} CaseAltn$
 .1 *CaseAltn2CaseAltn* (*match*, *body*, ρ) \triangleq
 .2 let $match_{CAS} = Pattern2Pattern (match, \rho)$,
 .3 $body_{CAS} = Expr2Expr (body, \rho)$ in
 .4 *mk-CAS* $^{\epsilon}$ *CaseAltn* ($match_{CAS}$, $body_{CAS}$)

Unary Expressions

functions

98.0 *PrefixExpr2UnaryExpr* : $OAS^{\epsilon} PrefixExpr \times Env \rightarrow CAS^{\epsilon} Expr$
 .1 *PrefixExpr2UnaryExpr* (*mk-OAS* $^{\epsilon}$ *PrefixExpr* (*op*, *expr*), ρ) \triangleq
 .2 let $arg_{CAS} = Expr2Expr (expr, \rho)$ in
 .3 *mk-CAS* $^{\epsilon}$ *UnaryExpr* (*op*, arg_{CAS}) ;
 99.0 *MapInverseExpr2UnaryExpr* : $OAS^{\epsilon} MapInverseExpr \times Env \rightarrow CAS^{\epsilon} UnaryExpr$
 .1 *MapInverseExpr2UnaryExpr* (*mk-OAS* $^{\epsilon}$ *MapInverseExpr* (*expr*), ρ) \triangleq
 .2 let $arg_{CAS} = Expr2Expr (expr, \rho)$ in
 .3 *mk-CAS* $^{\epsilon}$ *UnaryExpr* (MAPINVERSE, arg_{CAS})

Binary Expressions

functions

100.0 *BinaryExpr2BinaryExpr* : $OAS^{\epsilon} BinaryExpr \times Env \rightarrow CAS^{\epsilon} Expr$
 .1 *BinaryExpr2BinaryExpr* (*mk-OAS* $^{\epsilon}$ *BinaryExpr* (*left*, *op*, *right*), ρ) \triangleq
 .2 let $left_{CAS} = Expr2Expr (left, \rho)$,
 .3 $right_{CAS} = Expr2Expr (right, \rho)$ in
 .4 *mk-CAS* $^{\epsilon}$ *BinaryExpr* ($left_{CAS}$, *op*, $right_{CAS}$)

Quantified Expressions

functions

101.0 *AllExpr2AllOrExistsExpr* : $OAS^{\epsilon} AllExpr \times Env \rightarrow CAS^{\epsilon} AllOrExistsExpr$
 .1 *AllExpr2AllOrExistsExpr* (*mk-OAS* $^{\epsilon}$ *AllExpr* (*bind*, *pred*), ρ) \triangleq
 .2 let $bind_{CAS} = BindList2BindSet (bind, \rho)$,
 .3 $pred_{CAS} = Expr2Expr (pred, \rho)$ in
 .4 *mk-CAS* $^{\epsilon}$ *AllOrExistsExpr* (ALL, $bind_{CAS}$, $pred_{CAS}$) ;
 102.0 *ExistsExpr2AllOrExistsExpr* : $OAS^{\epsilon} ExistsExpr \times Env \rightarrow CAS^{\epsilon} AllOrExistsExpr$
 .1 *ExistsExpr2AllOrExistsExpr* (*mk-OAS* $^{\epsilon}$ *ExistsExpr* (*bind*, *pred*), ρ) \triangleq
 .2 let $bind_{CAS} = BindList2BindSet (bind, \rho)$,
 .3 $pred_{CAS} = Expr2Expr (pred, \rho)$ in
 .4 *mk-CAS* $^{\epsilon}$ *AllOrExistsExpr* (EXISTS, $bind_{CAS}$, $pred_{CAS}$) ;
 103.0 *ExistsUniqueExpr2ExistsUniqueExpr* :
 $OAS^{\epsilon} ExistsUniqueExpr \times Env \rightarrow CAS^{\epsilon} ExistsUniqueExpr$
 .1 *ExistsUniqueExpr2ExistsUniqueExpr* (*mk-OAS* $^{\epsilon}$ *ExistsUniqueExpr* (*bind*, *pred*), ρ) \triangleq
 .2 let $bind_{CAS} = Bind2Bind (bind, \rho)$,
 .3 $pred_{CAS} = Expr2Expr (pred, \rho)$ in
 .4 *mk-CAS* $^{\epsilon}$ *ExistsUniqueExpr* ($bind_{CAS}$, $pred_{CAS}$)

Iota Expression

functions

- 104.0 $IotaExpr2IotaExpr : OAS'IotaExpr \times Env \rightarrow CAS'IotaExpr$
.1 $IotaExpr2IotaExpr(mk-OAS'IotaExpr(bind, pred), \rho) \triangleq$
.2 let $bind_{CAS} = Bind2Bind(bind, \rho)$,
.3 $pred_{CAS} = Expr2Expr(pred, \rho)$ in
.4 $mk-CAS'IotaExpr(bind_{CAS}, pred_{CAS})$

Set Expressions

functions

- 105.0 $SetEnumeration2SetEnumeration : OAS'SetEnumeration \times Env \rightarrow CAS'SetEnumeration$
.1 $SetEnumeration2SetEnumeration(mk-OAS'SetEnumeration(els), \rho) \triangleq$
.2 let $els_{CAS} = [Expr2Expr(els(i), \rho) | i \in \text{inds } els]$ in
.3 $mk-CAS'SetEnumeration(els_{CAS})$;
- 106.0 $SetComprehension2SetComprehension :$
 $OAS'SetComprehension \times Env \rightarrow CAS'SetComprehension$
.2 $SetComprehension2SetComprehension(mk-OAS'SetComprehension(elem, bind, pred), \rho) \triangleq$
.3 let $elem_{CAS} = Expr2Expr(elem, \rho)$,
.4 $binds_{CAS} = BindList2BindSet(bind, \rho)$,
.5 $pred_{CAS} = OExpr2TrueOrExpr(pred, \rho)$ in
.6 $mk-CAS'SetComprehension(elem_{CAS}, binds_{CAS}, pred_{CAS})$;
- 107.0 $SetRange2SetRange : OAS'SetRange \times Env \rightarrow CAS'SetRange$
.1 $SetRange2SetRange(mk-OAS'SetRange(lb, ub), \rho) \triangleq$
.2 let $lb_{CAS} = Expr2Expr(lb, \rho)$,
.3 $ub_{CAS} = Expr2Expr(ub, \rho)$ in
.4 $mk-CAS'SetRange(lb_{CAS}, ub_{CAS})$

Sequence Expressions

functions

- 108.0 $SeqEnumeration2SeqEnumeration : OAS'SeqEnumeration \times Env \rightarrow CAS'SeqEnumeration$
.1 $SeqEnumeration2SeqEnumeration(mk-OAS'SeqEnumeration(els), \rho) \triangleq$
.2 let $els_{CAS} = [Expr2Expr(els(i), \rho) | i \in \text{inds } els]$ in
.3 $mk-CAS'SeqEnumeration(els_{CAS})$;
- 109.0 $SeqComprehension2SeqComprehension :$
 $OAS'SeqComprehension \times Env \rightarrow CAS'SeqComprehension$
.2 $SeqComprehension2SeqComprehension(mk-OAS'SeqComprehension(elem, bind, pred), \rho) \triangleq$
.3 let $elem_{CAS} = Expr2Expr(elem, \rho)$,
.4 $binds_{CAS} = Bind2Bind(bind, \rho)$,
.5 $pred_{CAS} = OExpr2TrueOrExpr(pred, \rho)$ in
.6 $mk-CAS'SeqComprehension(elem_{CAS}, binds_{CAS}, pred_{CAS})$;

110.0 *SubSequence2SubSequence* : $OAS^{\prime} SubSequence \times Env \rightarrow CAS^{\prime} SubSequence$
 .1 *SubSequence2SubSequence* (*mk-OAS[′]SubSequence* (*sequence*, *frompos*, *topos*), ρ) \triangleq
 .2 let *sequence_{CAS}* = *Expr2Expr* (*sequence*, ρ),
 .3 *from_{CAS}* = *Expr2Expr* (*frompos*, ρ),
 .4 *to_{CAS}* = *Expr2Expr* (*topos*, ρ) in
 .5 *mk-CAS[′]SubSequence* (*sequence_{CAS}*, *from_{CAS}*, *to_{CAS}*)

Map Expressions

functions

111.0 *MapEnumeration2MapEnumeration* : $OAS^{\prime} MapEnumeration \times Env \rightarrow CAS^{\prime} MapEnumeration$
 .1 *MapEnumeration2MapEnumeration* (*mk-OAS[′]MapEnumeration* (*els*), ρ) \triangleq
 .2 let *els_{CAS}* = [*Maplet2Maplet* (*els*(*i*), ρ) | *i* \in *inds els*] in
 .3 *mk-CAS[′]MapEnumeration* (*els_{CAS}*);
 112.0 *Maplet2Maplet* : $OAS^{\prime} Maplet \times Env \rightarrow CAS^{\prime} Maplet$
 .1 *Maplet2Maplet* (*mk-OAS[′]Maplet* (*mapdom*, *maprng*), ρ) \triangleq
 .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* :
 OAS[′]MapComprehension \times *Env* \rightarrow *CAS[′]MapComprehension*
 .2 *MapComprehension2MapComprehension* (*mk-OAS[′]MapComprehension* (*elem*, *bind*, *pred*), ρ) \triangleq
 .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}*)

Tuple Constructor Expression

functions

114.0 *TupleConstructor2TupleConstructor* : $OAS^{\prime} TupleConstructor \times Env \rightarrow CAS^{\prime} TupleConstructor$
 .1 *TupleConstructor2TupleConstructor* (*mk-OAS[′]TupleConstructor* (*fields*), ρ) \triangleq
 .2 let *fields_{CAS}* = [*Expr2Expr* (*fields*(*i*), ρ) | *i* \in *inds fields*] in
 .3 *mk-CAS[′]TupleConstructor* (*fields_{CAS}*)

Record Expressions

functions

115.0 *RecordConstructor2RecordConstructor* :
 OAS[′]RecordConstructor \times *Env* \rightarrow *CAS[′]RecordConstructor*
 .2 *RecordConstructor2RecordConstructor* (*mk-OAS[′]RecordConstructor* (*tag*, *fields*), ρ) \triangleq
 .3 let *tag_{CAS}* = *Id2Id* (*tag*, { \mapsto }),
 .4 *fields_{CAS}* = [*Expr2Expr* (*fields*(*i*), ρ) | *i* \in *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) \triangleq
 .2 let rec_{CAS} = *Expr2Expr* (*rec*, ρ),
 .3 *modifiers_{CAS}* = {*Id2Id* (*modifier.field*, ρ) \mapsto *Expr2Expr* (*modifier.new*, ρ) |
 .4 *modifier* ∈ *elems modifiers*} in
 .5 *mk-CAS‘RecordModifier* (rec_{CAS} , *modifiers_{CAS}*)

Apply Expressions

functions

117.0 *Apply2Apply* : OAS‘Apply × Env → CAS‘Apply
 .1 *Apply2Apply* (*mk-OAS‘Apply* (*fct*, *arg*), ρ) \triangleq
 .2 let fct_{CAS} = *Expr2Expr* (*fct*, ρ),
 .3 *arg_{CAS}* = cases len *arg* :
 .4 0 \rightarrow *mk-CAS‘NilLit* (),
 .5 1 \rightarrow *Expr2Expr* (*hd arg*, ρ),
 .6 others \rightarrow 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) \triangleq
 .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) \triangleq
 .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}*)

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    typeCAS = 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, typeCAS), Expr2Expr (body, ρ)),
.10     others → let mk- (idOAS, idCAS) = GetUnusedId () in
.11         mk-CAS'Lambda (mk-CAS'Par (idCAS, typeCAS),
.12             MakeLetExpr (parm, nil, idOAS, body, ρ))
.13   end
```

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

Is Expressions

functions

```
121.0  IsDefTypeExpr2IsExpr : OAS'IsDefTypeExpr × Env → CAS'IsExpr
.1    IsDefTypeExpr2IsExpr (mk-OAS'IsDefTypeExpr (deftype, arg), ρ) △
.2    let idOAS = let mk-OAS'IsId (id) = deftype in mk-OAS'ValueId (id),
.3    tagCAS = Id2Id (idOAS, {→}),
.4    argCAS = Expr2Expr (arg, ρ) in
.5    mk-CAS'IsExpr (tagCAS, argCAS)
.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 tagCAS = mk-CAS'BasicType (type),
.3    argCAS = Expr2Expr (arg, ρ) in
.4    mk-CAS'IsExpr (tagCAS, argCAS)
```

Names

functions

- 123.0 $Name2Id : OAS^{\text{Name}} \times Env \rightarrow CAS^{\text{Id}}$
.1 $Name2Id(mk-OAS^{\text{Name}}(name), \rho) \triangleq Id2Id(name, \rho)$;
.2
124.0 $OldName2OldId : OAS^{\text{ValueId}} \rightarrow CAS^{\text{OldId}}$
.1 $OldName2OldId(mk-OAS^{\text{ValueId}}(id)) \triangleq mk-CAS^{\text{OldId}}(id)$
.2

8.3.4 State Designators

functions

- 125.0 $StateDesignator2StateDesignator : OAS^{\text{StateDesignator}} \rightarrow CAS^{\text{StateDesignator}}$
.1 $StateDesignator2StateDesignator(sd) \triangleq$
.2 cases sd :
.3 $mk-OAS^{\text{Name}}(name) \rightarrow Id2Id(name, \{\mapsto\})$,
.4 $mk-OAS^{\text{FieldRef}}(var, sel) \rightarrow$
.5 let $sd_{CAS} = StateDesignator2StateDesignator(var)$,
.6 $sel_{CAS} = Id2Id(sel, \{\mapsto\})$ in
.7 $mk-CAS^{\text{FieldRef}}(sd_{CAS}, sel_{CAS})$,
.8 $mk-OAS^{\text{MapOrSeqRef}}(var, arg) \rightarrow$
.9 let $sd_{CAS} = StateDesignator2StateDesignator(var)$,
.10 $sel_{CAS} = Expr2Expr(arg, \{\mapsto\})$ in
.11 $mk-CAS^{\text{MapOrSeqRef}}(sd_{CAS}, sel_{CAS})$
.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

8.3.5 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

```

Local Binding Statements

functions

```

127.0   LetStmt2LetStmt : OAS'LetStmt → CAS'LetStmt
.1      LetStmt2LetStmt (mk-OAS'LetStmt (letdefs, body)) △
.2      let defsCAS      = merge { ValueDefinitions2ValDefMap (vds) |
.3                                mk-OAS'ValueDefinitions (vds) ∈ elems letdefs },
.4      mk- (explfnsCAS, -) = {FunctionDef2ExplFnDefMap (fd) |
.5                                fd: OAS'ExplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)},
.6      mk- (implfnsCAS, -) = {FunctionDef2ImplFnDefMap (fd) |
.7                                fd: OAS'ImplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)},
.8      bodyCAS          = Stmt2Stmt (body) in
.9      mk-CAS'LetStmt (defsCAS, explfnsCAS, implfnsCAS, bodyCAS)
.10     pre let mk- (explfnsCAS, -) = {FunctionDef2ExplFnDefMap (fd) |
.11                                fd: OAS'ExplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)},
.12     mk- (implfnsCAS, -) = {FunctionDef2ImplFnDefMap (fd) |
.13                                fd: OAS'ImplFnDef · fd ∈ elems letdefs ∧ ¬ IsPolymorphic (fd)} in
.14     ∀ id ∈ dom explfnsCAS ∪ dom implfnsCAS · 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

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 *BlockStatement2BlockOrSequence* 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*, { \mapsto })
 .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

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*, { \mapsto }),
 .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}*) ;

133.0 *CasesStmt2CasesStmt* : *OAS'CasesStmt* → *CAS'CasesStmt*
 .1 *CasesStmt2CasesStmt* (*mk-OAS'CasesStmt* (*sel*, *altns*, *other*)) △
 .2 let *sel_{CAS}* = *Expr2Expr* (*sel*, { \mapsto }),
 .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 $\text{CaseStmtAltn2CaseStmtAltn} : \text{OAS}'\text{Pattern} \times \text{OAS}'\text{Stmt} \rightarrow \text{CAS}'\text{CaseStmtAltn}$

- .1 $\text{CaseStmtAltn2CaseStmtAltn}(\text{match}, \text{body}) \triangleq$
- .2 $\text{let } \text{match}_{\text{CAS}} = \text{Pattern2Pattern}(\text{match}, \{\mapsto\}),$
- .3 $\text{body}_{\text{CAS}} = \text{Stmt2Stmt}(\text{body}) \text{ in}$
- .4 $\text{mk-CAS}'\text{CaseStmtAltn}(\text{match}_{\text{CAS}}, \text{body}_{\text{CAS}})$

Loop Statements

functions

135.0 $\text{SeqForLoop2SeqForLoop} : \text{OAS}'\text{SeqForLoop} \rightarrow \text{CAS}'\text{SeqForLoop}$

- .1 $\text{SeqForLoop2SeqForLoop}(\text{mk-OAS}'\text{SeqForLoop}(\text{cv}, \text{dirn}, \text{forseq}, \text{body})) \triangleq$
- .2 $\text{let } \text{cv}_{\text{CAS}} = \text{PatternBind2Pattern}(\text{cv}, \{\mapsto\}),$
- .3 $\text{dirn}_{\text{CAS}} = \text{if } \text{dirn} = \text{nil} \text{ then FORWARDS}$
- .4 $\text{else BACKWARDS},$
- .5 $\text{seq}_{\text{CAS}} = \text{Expr2Expr}(\text{forseq}, \{\mapsto\}),$
- .6 $\text{body}_{\text{CAS}} = \text{Stmt2Stmt}(\text{body}) \text{ in}$
- .7 $\text{mk-CAS}'\text{SeqForLoop}(\text{cv}_{\text{CAS}}, \text{dirn}_{\text{CAS}}, \text{seq}_{\text{CAS}}, \text{body}_{\text{CAS}});$

136.0 $\text{SetForLoop2SetForLoop} : \text{OAS}'\text{SetForLoop} \rightarrow \text{CAS}'\text{SetForLoop}$

- .1 $\text{SetForLoop2SetForLoop}(\text{mk-OAS}'\text{SetForLoop}(\text{cv}, \text{forset}, \text{body})) \triangleq$
- .2 $\text{let } \text{cv}_{\text{CAS}} = \text{Pattern2Pattern}(\text{cv}, \{\mapsto\}),$
- .3 $\text{set}_{\text{CAS}} = \text{Expr2Expr}(\text{forset}, \{\mapsto\}),$
- .4 $\text{body}_{\text{CAS}} = \text{Stmt2Stmt}(\text{body}) \text{ in}$
- .5 $\text{mk-CAS}'\text{SetForLoop}(\text{cv}_{\text{CAS}}, \text{set}_{\text{CAS}}, \text{body}_{\text{CAS}});$

137.0 $\text{IndexForLoop2IndexForLoop} : \text{OAS}'\text{IndexForLoop} \rightarrow \text{CAS}'\text{IndexForLoop}$

- .1 $\text{IndexForLoop2IndexForLoop}(\text{mk-OAS}'\text{IndexForLoop}(\text{cv}, \text{lb}, \text{ub}, \text{step}, \text{body})) \triangleq$
- .2 $\text{let } \text{cv}_{\text{CAS}} = \text{Id2Id}(\text{cv}, \{\mapsto\}),$
- .3 $\text{lb}_{\text{CAS}} = \text{Expr2Expr}(\text{lb}, \{\mapsto\}),$
- .4 $\text{ub}_{\text{CAS}} = \text{Expr2Expr}(\text{ub}, \{\mapsto\}),$
- .5 $\text{by}_{\text{CAS}} = \text{if } \text{step} = \text{nil} \text{ then } \text{mk-CAS}'\text{NumLit}(1)$
- .6 $\text{else Expr2Expr}(\text{step}, \{\mapsto\}),$
- .7 $\text{body}_{\text{CAS}} = \text{Stmt2Stmt}(\text{body}) \text{ in}$
- .8 $\text{mk-CAS}'\text{IndexForLoop}(\text{cv}_{\text{CAS}}, \text{lb}_{\text{CAS}}, \text{ub}_{\text{CAS}}, \text{by}_{\text{CAS}}, \text{body}_{\text{CAS}});$

138.0 $\text{WhileLoop2WhileLoop} : \text{OAS}'\text{WhileLoop} \rightarrow \text{CAS}'\text{WhileLoop}$

- .1 $\text{WhileLoop2WhileLoop}(\text{mk-OAS}'\text{WhileLoop}(\text{test}, \text{body})) \triangleq$
- .2 $\text{let } \text{test}_{\text{CAS}} = \text{Expr2Expr}(\text{test}, \{\mapsto\}),$
- .3 $\text{body}_{\text{CAS}} = \text{Stmt2Stmt}(\text{body}) \text{ in}$
- .4 $\text{mk-CAS}'\text{WhileLoop}(\text{test}_{\text{CAS}}, \text{body}_{\text{CAS}})$

NonDeterministic Statement

functions

139.0 $\text{NonDetStmt2NonDetStmt} : \text{OAS}'\text{NonDetStmt} \rightarrow \text{CAS}'\text{NonDetStmt}$

- .1 $\text{NonDetStmt2NonDetStmt}(\text{mk-OAS}'\text{NonDetStmt}(\text{stmts})) \triangleq$
- .2 $\text{let } \text{stmts}_{\text{CAS}} = [\text{Stmt2Stmt}(\text{stmt}(i)) \mid i \in \text{inds } \text{stmts}] \text{ in}$
- .3 $\text{mk-CAS}'\text{NonDetStmt}(\text{stmts}_{\text{CAS}})$

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 *oprtCAS* = *Id2Id* (*oprt*, {→}),
- .3 *argCAS* = *cases len args* :
 - .4 0 → *mk-CAS‘NilLit* (),
 - .5 1 → *Expr2Expr* (*hd args*, {→}),
 - .6 others → let *fields* = [*Expr2Expr* (*args* (*i*), {→}) | *i* ∈ *inds args*] in
mk-CAS‘TupleConstructor (*fields*)
- .7 end,
- .8 *stateCAS* = if *callst* = nil then nil
- .9 else *StateDesignator2StateDesignator* (*callst*) in
- .10 mk-CAS‘Call (*oprtCAS*, *argCAS*, *stateCAS*)
- .11 mk-CAS‘Call (*oprtCAS*, *argCAS*, *stateCAS*)

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 *exprCAS* = *Expr2Expr* (*expr*, {→}) in
- .3 mk-CAS‘ReturnStmt (*exprCAS*)

Exception Handling Statements

functions

142.0 *AlwaysStmt2Always* : OAS‘AlwaysStmt → CAS‘Always

- .1 *AlwaysStmt2Always* (*mk-OAS‘AlwaysStmt* (*alwpost*, *body*)) △
- .2 let *postCAS* = *Stmt2Stmt* (*alwpost*),
- .3 *bodyCAS* = *BlockStmt2BlockOrSequence* (*body*) in
- .4 mk-CAS‘Always (*postCAS*, *bodyCAS*) ;

143.0 *TrapStmt2TrapStmt* : OAS‘TrapStmt → CAS‘TrapStmt

- .1 *TrapStmt2TrapStmt* (*mk-OAS‘TrapStmt* (*pat*, *trappost*, *body*)) △
- .2 let *patCAS* = *PatternBind2Pattern* (*pat*, {→}),
- .3 *postCAS* = *Stmt2Stmt* (*trappost*),
- .4 *bodyCAS* = *BlockStmt2BlockOrSequence* (*body*) in
- .5 mk-CAS‘TrapStmt (*patCAS*, *postCAS*, *bodyCAS*) ;

144.0 *RecTrapStmt2RecTrapStmt* : OAS‘RecTrapStmt → CAS‘RecTrapStmt

- .1 *RecTrapStmt2RecTrapStmt* (*mk-OAS‘RecTrapStmt* (*traps*, *body*)) △
- .2 let *trapsCAS* = {*PatternBind2Pattern* (*match*, {→}) ↦ *Stmt2Stmt* (*trappost*) |
mk-OAS‘Trap (*match*, *trappost*) ∈ *elems traps*},
- .3 *bodyCAS* = *BlockStmt2BlockOrSequence* (*body*) in
- .4 mk-CAS‘RecTrapStmt (*trapsCAS*, *bodyCAS*)

```

.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 exprCAS = Expr2Expr (expr, {→}) in
.3     mk-CAS'Exit (exprCAS)

```

Identity Statement

This section is intentionally left empty.

8.3.6 Patterns and Bindings

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 idCAS = Id2Id (name, {→}) in
.3     mk-CAS'PatternId (idCAS) ;

```

```

148.0   SetEnumPattern2SetEnumPattern : OAS'SetEnumPattern × Env → CAS'SetEnumPattern
.1   SetEnumPattern2SetEnumPattern (mk-OAS'SetEnumPattern (els), ρ) △
.2     let elsCAS = [Pattern2Pattern (els(i), ρ) | i ∈ inds els] in
.3     mk-CAS'SetEnumPattern (elsCAS) ;

```

```

149.0   SetUnionPattern2SetUnionPattern : OAS'SetUnionPattern × Env → CAS'SetUnionPattern
.1   SetUnionPattern2SetUnionPattern (mk-OAS'SetUnionPattern (lp, rp), ρ) △
.2     let lpCAS = Pattern2Pattern (lp, ρ),
.3       rpCAS = Pattern2Pattern (rp, ρ) in
.4     mk-CAS'SetUnionPattern (lpCAS, rpCAS) ;

```

150.0 $\text{SeqEnumPattern2SeqEnumPattern} : \text{OAS}'\text{SeqEnumPattern} \times \text{Env} \rightarrow \text{CAS}'\text{SeqEnumPattern}$
 .1 $\text{SeqEnumPattern2SeqEnumPattern}(\text{mk-OAS}'\text{SeqEnumPattern}(\text{els}), \rho) \triangleq$
 .2 let $\text{els}_{\text{CAS}} = [\text{Pattern2Pattern}(\text{els}(i), \rho) \mid i \in \text{inds els}]$ in
 .3 $\text{mk-CAS}'\text{SeqEnumPattern}(\text{els}_{\text{CAS}})$;

 151.0 $\text{SeqConcPattern2SeqConcPattern} : \text{OAS}'\text{SeqConcPattern} \times \text{Env} \rightarrow \text{CAS}'\text{SeqConcPattern}$
 .1 $\text{SeqConcPattern2SeqConcPattern}(\text{mk-OAS}'\text{SeqConcPattern}(lp, rp), \rho) \triangleq$
 .2 let $lp_{\text{CAS}} = \text{Pattern2Pattern}(lp, \rho)$,
 .3 $rp_{\text{CAS}} = \text{Pattern2Pattern}(rp, \rho)$ in
 .4 $\text{mk-CAS}'\text{SeqConcPattern}(lp_{\text{CAS}}, rp_{\text{CAS}})$;

 152.0 $\text{TuplePattern2TuplePattern} : \text{OAS}'\text{TuplePattern} \times \text{Env} \rightarrow \text{CAS}'\text{TuplePattern}$
 .1 $\text{TuplePattern2TuplePattern}(\text{mk-OAS}'\text{TuplePattern}(\text{fields}), \rho) \triangleq$
 .2 let $\text{fields}_{\text{CAS}} = [\text{Pattern2Pattern}(\text{fields}(i), \rho) \mid i \in \text{inds fields}]$ in
 .3 $\text{mk-CAS}'\text{TuplePattern}(\text{fields}_{\text{CAS}})$;

 153.0 $\text{RecordPattern2RecordPattern} : \text{OAS}'\text{RecordPattern} \times \text{Env} \rightarrow \text{CAS}'\text{RecordPattern}$
 .1 $\text{RecordPattern2RecordPattern}(\text{mk-OAS}'\text{RecordPattern}(id, \text{fields}), \rho) \triangleq$
 .2 let $\text{tag}_{\text{CAS}} = \text{Id2Id}(id, \{\mapsto\})$,
 .3 $\text{fields}_{\text{CAS}} = [\text{Pattern2Pattern}(\text{fields}(i), \rho) \mid i \in \text{inds fields}]$ in
 .4 $\text{mk-CAS}'\text{RecordPattern}(\text{tag}_{\text{CAS}}, \text{fields}_{\text{CAS}})$
 .5 pre $\text{is-OAS}'\text{MkId}(id)$

Bindings

functions

154.0 $\text{PatternBind2Pattern} : \text{OAS}'\text{PatternBind} \times \text{Env} \rightarrow \text{CAS}'\text{Pattern}$
 .1 $\text{PatternBind2Pattern}(pb, \rho) \triangleq$
 .2 cases pb :
 .3 $\text{mk-OAS}'\text{SetBind}(\text{pat}, \text{set}) \rightarrow$ let $\text{pat}_{\text{CAS}} = \text{Pattern2Pattern}(\text{pat}, \rho)$,
 .4 $\text{set}_{\text{CAS}} = \text{Expr2Expr}(\text{set}, \rho)$ in
 .5 $\text{mk-CAS}'\text{SetConstrPattern}(\text{pat}_{\text{CAS}}, \text{set}_{\text{CAS}})$,
 .6 $\text{mk-OAS}'\text{TypeBind}(\text{pat}, \text{type}) \rightarrow$ let $\text{pat}_{\text{CAS}} = \text{Pattern2Pattern}(\text{pat}, \rho)$,
 .7 $\text{type}_{\text{CAS}} = \text{Type2Type}(\text{type})$ in
 .8 $\text{mk-CAS}'\text{TypeConstrPattern}(\text{pat}_{\text{CAS}}, \text{type}_{\text{CAS}})$,
 .9 others
 .10 end ;

155.0 $\text{Bind2Bind} : \text{OAS}'\text{Bind} \times \text{Env} \rightarrow \text{CAS}'\text{Bind}$
 .1 $\text{Bind2Bind}(b, \rho) \triangleq$
 .2 cases b :
 .3 $\text{mk-OAS}'\text{SetBind}(\text{pat}, \text{bindset}) \rightarrow$ let $\text{pat}_{\text{CAS}} = \text{Pattern2Pattern}(\text{pat}, \rho)$,
 .4 $\text{set}_{\text{CAS}} = \text{Expr2Expr}(\text{bindset}, \rho)$ in
 .5 $\text{mk-CAS}'\text{SetBind}(\text{pat}_{\text{CAS}}, \text{set}_{\text{CAS}})$,
 .6 $\text{mk-OAS}'\text{TypeBind}(\text{pat}, \text{type}) \rightarrow$ let $\text{pat}_{\text{CAS}} = \text{Pattern2Pattern}(\text{pat}, \rho)$,
 .7 $\text{type}_{\text{CAS}} = \text{Type2Type}(\text{type})$ in
 .8 $\text{mk-CAS}'\text{TypeBind}(\text{pat}_{\text{CAS}}, \text{type}_{\text{CAS}})$
 .9 end ;

156.0 $BindList2BindSet : OAS^{\text{BindList}} \times Env \rightarrow CAS^{\text{Bind-set}}$

.1 $BindList2BindSet(bs, \rho) \triangleq$
 .2 $\bigcup \{ \{ Bind2Bind (\text{if } is\text{-}OAS^{\text{MultSetBind}}(bs(i))$
 .3 $\text{then let } mk\text{-}OAS^{\text{MultSetBind}}(\text{pats}, expr) = bs(i) \text{ in}$
 .4 $mk\text{-}OAS^{\text{SetBind}}(\text{pats}(j), expr)$
 .5 $\text{else let } mk\text{-}OAS^{\text{MultSetBind}}(\text{pats}, tp) = bs(i) \text{ in}$
 .6 $mk\text{-}OAS^{\text{TypeBind}}(\text{pats}(j), tp), rho) |$
 .7 $j \in \text{inds } bs(i).\text{pats} \} | i \in \text{inds } bs \}$
 .8 pre card $\bigcup \{ \{ Bind2Bind (\text{if } is\text{-}OAS^{\text{MultSetBind}}(bs(i))$
 .9 $\text{then let } mk\text{-}OAS^{\text{MultSetBind}}(\text{pats}, expr) = bs(i) \text{ in}$
 .10 $mk\text{-}OAS^{\text{SetBind}}(\text{pats}(j), expr)$
 .11 $\text{else let } mk\text{-}OAS^{\text{MultSetBind}}(\text{pats}, tp) = bs(i) \text{ in}$
 .12 $mk\text{-}OAS^{\text{TypeBind}}(\text{pats}(j), tp), rho) |$
 .13 $j \in \text{inds } bs(i).\text{pats} \} | i \in \text{inds } bs \} =$
 .14 len conc $[[Bind2Bind (\text{if } is\text{-}OAS^{\text{MultSetBind}}(bs(i))$
 .15 $\text{then let } mk\text{-}OAS^{\text{MultSetBind}}(\text{pats}, expr) = bs(i) \text{ in}$
 .16 $mk\text{-}OAS^{\text{SetBind}}(\text{pats}(j), expr)$
 .17 $\text{else let } mk\text{-}OAS^{\text{MultSetBind}}(\text{pats}, tp) = bs(i) \text{ in}$
 .18 $mk\text{-}OAS^{\text{TypeBind}}(\text{pats}(j), tp), rho) |$
 .19 $j \in \text{inds } bs(i).\text{pats}] | i \in \text{inds } bs]$

annotations

.8 The pre-condition specifies that no duplicate bindings are allowed.

end annotations

8.3.7 Lexical Specification

General

This section is intentionally left empty.

Characters

This section is intentionally left empty.

Symbols

functions

157.0 $Id2Id : [OAS^{\text{Id}}] \times Env \rightarrow [CAS^{\text{Id}}]$

.1 $Id2Id(id, \rho) \triangleq$
 .2 if $id \in \text{dom } \rho$
 .3 then $\rho(id)$
 .4 else cases $id :$
 .5 $mk\text{-}OAS^{\text{ValueId}}(id) \rightarrow mk\text{-}CAS^{\text{ValueId}}(id),$
 .6 $mk\text{-}OAS^{\text{PreId}}(id) \rightarrow mk\text{-}CAS^{\text{PreId}}(id),$
 .7 $mk\text{-}OAS^{\text{PostId}}(id) \rightarrow mk\text{-}CAS^{\text{PostId}}(id),$
 .8 $mk\text{-}OAS^{\text{InvId}}(id) \rightarrow mk\text{-}CAS^{\text{InvId}}(id),$
 .9 $mk\text{-}OAS^{\text{InitId}}(id) \rightarrow mk\text{-}CAS^{\text{InitId}}(id),$
 .10 $mk\text{-}OAS^{\text{MkId}}(id) \rightarrow mk\text{-}CAS^{\text{MkId}}(id),$
 .11 $mk\text{-}OAS^{\text{IsId}}(id) \rightarrow mk\text{-}CAS^{\text{IsId}}(id),$
 .12 others $\rightarrow \text{nil}$
 .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 CAS⁶

end annotations

end *OAS2CAS*

Chapter 9

The Background to the Static Semantics

This section defines the static semantics of VDM-SL, i.e. type checking etc., and its relationship to the Dynamic Semantics, i.e. the complete, denotational semantics of the language as defined in section 5.

9.1 Abstract Syntaxes, Dynamic Semantics and Consistency

The following briefly summarises how the Dynamic Semantics is used to assign meaning to (some) specifications conforming to the concrete syntax. On this basis some consistency-related notions are introduced in order to help clarify the rôle of the Static Semantics.

All specifications conforming to the Concrete Syntax may be parsed to yield abstract syntactic objects (parse trees) conforming to the Outer Abstract Syntax.

The abstract syntactic objects which conform to the Outer Abstract Syntax and, in addition, fulfill certain very simple requirements may be transformed to equivalent syntactic objects conforming to the Core Abstract Syntax. The requirements and transformation have been defined in section 8.

The Dynamic Semantics is defined as a total function on specifications conforming to the Core Abstract Syntax. Applied to a specification, the function yields the set of models of this specification.

Considering an arbitrary specification conforming to the Concrete Syntax, this has a semantics only if its Outer Abstract Syntax representation fulfils the requirements which enable the transformation to a Core Abstract Syntax representation. In this case, the semantics is the set of models which results from applying the Dynamic Semantics to the Core Abstract Syntax representation.

Note that the Dynamic Semantics in case of type errors etc. yields an empty set of models so in this sense the Dynamic Semantics definition may be seen as also defining a notion of consistency. In the following, a specification which has a semantics not being the empty set of models is called a *consistent* specification. Otherwise, the specification is said to be *inconsistent*. If it does not have a semantics (see above) the specification is moreover said to be *meaningless*.

9.2 Rôle of the Static Semantics

Informally speaking, the notion of static semantics comprises type checking and related consistency analyses which are decidable (i.e. may be implemented by a terminating algorithm). In the following, the different kinds of analyses that are included in the Static Semantics are briefly introduced.

The Static Semantics includes the test determining whether a specification is meaningless or not and rejects meaningless specifications. This test is very simple involving only detection of certain cases of multiple definitions of the same identifier.

Regarding inconsistent specifications, there are, according to the Dynamic Semantics, many different kinds of errors which *may* have caused the inconsistency:

1. Use of identifiers not defined or not visible; use of the same identifier as the “tag” of different composite types.
2. Recursion “through” function types in connection with recursive type equations.
3. Use of set types whose element type or a component thereof is a function type (a non-flat type); similarly cases regarding map types where the domain type builds on functions.
4. Type errors, i.e. application of user defined or built-in functions and operations to arguments for which they are not defined. This includes both simple cases such as adding a number and a Boolean value: $1 + \text{true}$ and more complicated cases like $s(n)$ where s is a variable denoting a sequence and n is another variable denoting an index which is greater than the length of the sequence. Note that, according to the Dynamic Semantics, type errors may be ignored in some cases, e.g. when considering operands of non-strict operators such as if-then-else and the Boolean operators and also when considering the body of functions not asserted to be total.
5. Multiple definitions of the same identifier.
6. Violation of specified types for directly defined functions. Considering functions which are *not* asserted to be total, the following must hold for all arguments of the specified types: if the arguments fulfill the pre-condition (or there is no pre-condition) then the body of a directly defined function must either yield a value of the specified result type or the special undefinedness value \perp . For functions asserted to be total, the same holds and in addition they are not allowed to yield \perp for any arguments fulfilling the pre-condition. The same must hold for all possible type instantiations of directly defined polymorphic functions.

Note that functions defined to be total may not be total either because of type errors in their definition or due to what is called non-termination in an operational setting.

7. Violation of specified types for directly defined operations. As for directly defined, partial functions. However, nothing is required regarding the result type if evaluation of the body results in an exit. If the evaluation of the body may complete without exiting and without returning a result value then no result type must be specified for the operation.
8. Unsuccessful attempts to match patterns with values, e.g. in a “let” definition or a “cases” construct where not all cases are covered by the patterns specified.
9. Assignment of a value to a state variable whose type does not include the value.
10. Type invariants which are loose, i.e. under-specified.
11. Unsatisfiability of function and operation specifications in the pre/post style, i.e. the case of a specification which cannot be fulfilled by any function/operation. Satisfiability is required for all possible type instantiations of pre/post specified polymorphic functions.
12. Unsatisfiability of mutually recursive type and function definitions (recursion via an invariant).
13. Unsatisfiability of a state initialisation predicate.

Considering this list, it is clear that inconsistency of specifications is undecidable in its generality. The above mentioned classes 1–3 are clearly decidable but the others definitely do. In fact, undecidability holds even for the sub-case of inconsistency caused by type errors in the application of such simple operations as numerical addition. Consider the following example illustrating this point:

$$\begin{aligned} f : (\mathbb{N} \mid \mathbb{B}) &\rightarrow \mathbb{N} \\ f(x) &\triangleq \\ &\quad \text{if } \text{is-}\mathbb{N}(x) \text{ then } x + 1 \text{ else } 1 \end{aligned}$$

Since the condition tested by the ‘if ... then ... else ...’ expression in general may be an arbitrary predicate, it is, in the general case, undecidable whether the truth of the condition prevents that x may be Boolean.

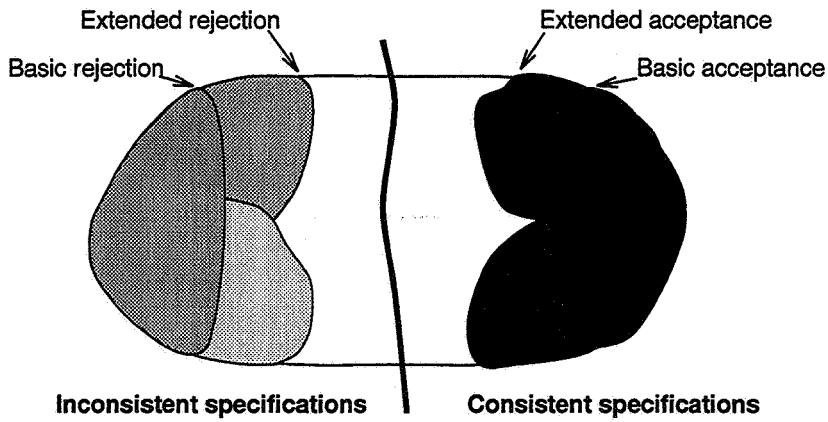


Figure 9.1: Rejected and accepted subsets of specifications.

The Static Semantics is therefore designed to reject a decidable subset of the specifications with inconsistencies in classes 4–9 in addition to rejecting all inconsistencies in the above mentioned classes 1–3.

In traditional type checking for programming languages, the statically undetectable type errors may be caught by dynamic checks carried out at run-time. Since VDM-SL specifications need not be executable, dynamic checks are not possible in general. Therefore, the only general way to gain confidence in the (type) consistency of a specification is to construct a consistency proof or a rigorous argument. To support this, an analysis which automatically accepts some subset of the definitely consistent specifications is also desirable.

The consistency of specifications is undecidable in the same way as inconsistency of specifications. The Static Semantics is therefore designed to accept a decidable subset of the definitely consistent specifications.

The practical applicability of the Static Semantics depends very much on the degree of overlap between the specifications actually written and the ones recognised as either inconsistent or consistent. It seems impossible to identify a “natural” place to stop in the search for more and more advanced analyses recognising increasingly larger sets of specifications. As a way of coping with this problem, the Static Semantics defines a basic rejection and acceptance analysis and also some extensions of these. The set of extensions is open to future additions and which of the extensions to actually implement in VDM-SL tools is left as a choice for the implementors to make. The relationship between the different acceptance and rejection analyses is illustrated in figure 9.1.

Neither the basic rejection analysis nor the extensions defined recognise inconsistencies in the above mentioned classes 10–13. Likewise, the acceptance analyses fail for all specifications containing implicit function definitions or recursive definitions of functions which are asserted to be total. It should be noted, however, that addition of correct analyses within these areas will be in accordance with the open-ended view of the Static Semantics presented above. Also note that the acceptance analyses may be useful even if they fail when considering a whole specification. This is due to the fact that specifications usually contain many trivially consistent subparts which may be recognised automatically. The specifier wishing to manually prove or argue for consistency of the whole specification may therefore be relieved from having to consider all these trivial subproblems.

Regarding inconsistencies in the above mentioned classes 10 and 12, it should be noted that the acceptance analyses are defined to recognise certain cases where invariants are definitely not underspecified (loose). The cases where there is no mutual recursion between function and type definitions are also identified; in these cases the can definitely not be an inconsistency caused by unsatisfiability.

9.3 Overview of the Definition

9.3.1 Notation and Style

The notation used for defining the Static Semantics of VDM-SL is VDM-SL itself. Note that this does not give rise to circularity: the semantics of VDM-SL specifications, including the specification of the Static Semantics, is defined by the Dynamic Semantics whose definition in turn does *not* refer to the Static Semantics.

The acceptance and rejection analyses are defined implicitly in order to emphasise the aspect of *which* specifications to accept/reject rather than the more procedural aspect of *how* to compute an acceptance or rejection result.

9.3.2 Type Representations

In the Outer Abstract Syntax a syntactic domain of types is defined; these are the type expressions which a specifier may write in a specification. For the definition of the Static Semantics this notion of type is, however, not sufficient. Therefore, a notion of "Type Representations" is introduced. All the Outer Abstract Syntax type expressions are representable using the type representations and the transformation from the former to the latter is rather straight-forward. In addition, there are type representations for some special types which are needed in the Static Semantics: the type of the constant "nil", the types of the empty set, sequence and map and a type containing all possible VDM-SL values. Special Type Representations have also been introduced in order to express static semantic properties of statements and operations.

Type definitions in the Outer Abstract Syntax are in the Static Semantics represented by Type Maps which map type names to Type Representations.

Viewed in relation to the Dynamic Semantics, it is important to note that Type Representations denote sets of "real" values, i.e. excluding \perp . When an acceptance analysis concludes that an expression has a type, it may therefore also be concluded that the expression will not evaluate to \perp .

The denotation of those Type Representations which correspond to type expressions in Outer Abstract Syntax can be found by first making the transformation from the Outer Abstract Syntax forms to the equivalent syntactic objects conforming to the Core Abstract Syntax and then applying the Dynamic Semantics. The semantics of the special Type Representations which have no counterpart in the Outer Abstract Syntax could also be defined in terms of the dynamic semantic domains.

In the following, the term "type" is used as a synonym for "Type Representation" whenever this does not cause confusion.

9.3.3 Well-formedness Classifications

An essential part of both the acceptance and rejection analysis concerns type checking of expressions. Regarding acceptance, this is defined in terms of a predicate for *definite well-formedness* of an expression with respect to a type and an environment. The environment contains information about the identifiers which may occur free in the expression. Rejection is defined indirectly in terms of a predicate for *possible well-formedness*. If an expression is not possibly well-formed with respect to a type in a given environment then it may be rejected that the expression can evaluate to a value in this type.

In mathematical terms, definite well-formedness is a *sound* expression typing predicate whereas possible well-formedness is a *complete* expression typing predicate with respect to the Dynamic Semantics.

The definitions of possible and definite well-formedness of expressions are expressed as a single parameterised predicate in order to emphasise the many similarities of the two notions. Likewise, the basic level of checking and the extensions which are defined are all expressed by this single predicate; the parameter indicates which extensions, if any, should be in effect. Regarding the other syntax classes, notably statements, patterns, types, definitions and whole specifications, a similar distinction between possible and definite well-formedness and different levels of checking has been introduced.

So for each syntax class, a single, parameterised well-formedness predicate has been defined. The parameter may be seen as expressing a well-formedness classification. The following choices are possible:

General classification: (A) Possible; (B) Definite (mutually exclusive).

Extensions: (1) Local Context; (2) Total functions; (3) Union Close (zero, one or more may be chosen).

Extension (3) is described below in the section “Generalisations of Characteristic Predicates”, whereas the two other extensions are introduced in the subsection “Well-formedness Predicates” below. As noted in the previous, this set of extensions is open for future additions.

9.3.4 Type Relations

In VDM-SL, the sets of values denoted by different types need not be disjoint. So when considering whether an expression will evaluate to values in a given type or not it is essential to be able to relate types: whether they denote overlapping or disjoint sets of values or whether one set is totally included in the other.

In the Static Semantics, all these relationships are defined in terms of a subtype predicate describing the cases in which all the values of one type are also values of another type. Considering types restricted by arbitrary invariants, it is clear that the subtype predicate is generally undecidable. Even when only dealing with types without invariants, it is an open theoretical problem whether subtyping is decidable or not.

Due to the general undecidability of the problem, two decidable approximations to the subtype relation are defined. The definite subtype relation is sound and may be used for ensuring that a subtype relationship really holds. The possible subtype relation is complete so in case it does *not* hold, it may safely be concluded that the types in question are not in subtype relationship to each other.

9.3.5 Characteristic Predicates

The basic type properties of many classes of composite VDM-SL expressions are in the Static Semantics defined by so-called “expression characteristic predicates” (characteristic predicates for short). The characteristic predicate for an expression class will for all expressions in this class relate the types of the subexpressions to the types of the expressions themselves. Expression characteristic predicates generally come in two variants: one which may be used for checking definite well-formedness and one for possible well-formedness.

The characteristic predicate concerning definite well-formedness of a class of expressions e is defined so it holds for “large” subexpression types and “small” result types. The subexpression types for which it holds are, however, not so large that they contain values for which the value of e may fall outside the result type. Since \perp is not an element of any type, this soundness property prevents that the characteristic predicate may hold for subexpression types containing values for which the expression might evaluate to \perp .

The characteristic predicate concerning possible well-formedness of a class of expressions e is defined so it holds for “small” subexpression types and “large” result types. For each result type, the union of all the subexpression types for which the predicate holds is, however, large enough that it contains all the values for which the value of e falls inside the result type. This completeness property gives a basis for establishing soundness of rejection defined in terms of non-possible well-formedness.

In many cases, the characteristic predicates concerning definite and possible well-formedness of a class of expressions are identical. As an example, consider the class of binary expressions: $e_1 \cup e_2$. Here a single predicate covers both the case of definite and possible well-formedness. The predicate relates the types of the two subexpressions to the result type of the whole expression. The predicate holds only if the two subexpression types are set types over some element types t_1 and t_2 and the result type is a set type over the element type $t_1 \sqcup t_2$.

As a case where the two variants are different, consider the expressions for indexing in sequences. Here, the characteristic predicate for definite well-formedness will be false for all subexpression types, including the cases where the types of the two subexpressions are a sequence type and \mathbb{N} , respectively. This is due to the fact that index specified (the second subexpression) might be greater than the length of the sequence in which case the indexing will yield \perp according to the Dynamic Semantics. The predicate for possible well-formedness holds, however, in the above mentioned case in order to ensure completeness. In other words, it cannot be rejected that the indexing yields a result different from \perp in this case.

The type properties of some VDM-SL expressions, e.g. the ones which introduce new identifiers, cannot be adequately defined in terms of characteristic predicates because the type of subexpressions may depend on the type of the identifiers introduced. Well-formedness of these expressions is therefore expressed directly without referring to characteristic predicates.

Regarding the syntax class of patterns, a similar notion of pattern characteristic predicates has been introduced.

9.3.6 Relaxation of Characteristic Predicates

The relaxation considered is relevant only in connection with the expression characteristic predicates concerning definite well-formedness. These predicates are defined in such a way that the possibility of a subexpression having a union type is not taken into account. Consider, e.g., a predicate for a class of expressions having only one subexpression. If the predicate holds both for a subexpression type t_1 with result type t'_1 and also for a subexpression type t_2 with result type t'_2 , then it is a sound relaxation to also let the predicate hold for the subexpression type $t_1 \mid t_2$ with result type $t'_1 \mid t'_2$. Similar results hold for characteristic predicates of expressions with more than one subexpression. This relaxation is, in the Static Semantics, expressed in terms of a function: "UnionClose" which maps an expression characteristic predicate to its relaxed form. A similar relaxation is defined for the characteristic predicates regarding patterns.

9.3.7 Well-formedness Predicates

As mentioned above, some classes of VDM-SL expressions are simple enough that their type properties can be expressed in the general framework of expression characteristic predicates whereas others must be considered individually. In both cases, however, a family of well-formedness predicates indexed by well-formedness classifications has been defined for each syntax class. Each of these predicates relates expressions to the types and environments for which they are well-formed. The predicates regarding definite well-formedness fulfill a soundness requirement whereas the predicates regarding possible well-formedness fulfill a completeness requirement:

Soundness If the predicate holds then the expression must evaluate to a value in the type for all bindings of identifiers to values in the types of the identifiers (as expressed by the environment).

Completeness If the expression evaluates to a value v different from \perp for some binding of identifiers to values in the types of the identifiers (as expressed by the environment) then the predicate holds for some type containing v .

The well-formedness predicates with the above mentioned properties may be used for verifying that expressions have the specific types required by their contexts. Certain relaxations are, however, necessary. In the case of definite well-formedness, it is acceptable if the predicates hold for types which are smaller than actually required. In the case of possible well-formedness, it is acceptable (or more precisely non-rejectable) if the predicates hold for types which are larger than the required ones. These relaxations are in the Static Semantics expressed as a higher-order function "InType" which maps well-formedness predicates to their relaxed form.

Considering the extended levels of checking, these are realised by relaxing the predicates for definite well-formedness so they hold in more cases (without violating the soundness requirement) and by restricting the predicates for possible well-formedness so they hold in fewer cases (without violating the completeness requirement).

The extension "Union Close" was explained above as a relaxation of characteristic predicates for definite well-formedness thereby indirectly introducing a relaxation of the well-formedness predicates expressed in terms of the characteristic predicates.

The extensions "Local Context" and "Total Functions" are defined as higher order functions mapping well-formedness predicates concerning possible well-formedness to more restricted predicates. Both of these restrictions are expressed in terms of case analyses on the types bound to identifiers in environments.

Consider the possible well-formedness of an expression with respect to a type and an environment. Assume that the environment binds an identifier to a type which can be expressed as the union of a finite

number of other types; these are called the union components of the type. An environment in which all the identifiers are bound to union components of their types in the original environment is called a sub-environment (of the original environment).

Local Context From a Static Semantics point of view, it is unknown precisely which values the identifiers in the environment are bound to during an actual evaluation. Whatever their value may be, it must, however, be contained in one of the union components of their type. Therefore, the original well-formedness predicate may be strengthened (without losing completeness) by requiring that the expression is well-formed in at least one of the sub-environments.

Total Functions For total functions without pre-conditions (explicit or hidden), the body must evaluate to a value in the declared result type for all bindings of formal parameters to values in their declared type. Therefore, the possible well-formedness of the body may be strengthened by requiring that the body be well-formed in all sub-environments which may be formed by considering union components of the types of the formal parameters.

Chapter 10

The Static Semantic Domains

This chapter 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.

10.1 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: $T \mid \text{UNITTYPE}$. 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
      .1           inv mk-UnionTypeR(s)  $\triangleq$  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-ProductTypeR(fs)  $\triangleq$  len fs  $\geq$  2;
9.0   SetTypeR::elemtp : TypeR ;
10.0  SeqTypeR = Seq0TypeR | Seq1TypeR;
11.0  Seq0TypeR::elemtp : TypeR ;
12.0  Seq1TypeR::elemtp : TypeR ;
13.0  MapTypeR = GeneralMapTypeR | InjectiveMapTypeR;
14.0  GeneralMapTypeR::mapdom : TypeR
      .1           maprng : TypeR ;
15.0  InjectiveMapTypeR::mapdom : TypeR
      .1           maprng : TypeR ;
16.0  FunctionTypeR = PartialFnTypeR | 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

```

10.1.1 Special Subclasses of Type Representations

```

23.0   StmtTypeR = TypeR
.1   inv str  $\triangleq$  cases str :
.2       (CONT), mk-RetTypeR(-) , mk-ExitTypeR(-)  $\rightarrow$  true,
.3       mk-UnionTypeR(ts)  $\rightarrow$   $\forall t \in ts \cdot \text{inv-}StmtTypeR(t)$ ,
.4       others  $\rightarrow$  false
.5   end;
24.0   StateTypeR = TypeR
.1   inv t  $\triangleq$  is-CompositeTypeR(t)  $\vee$ 
.2       is-InvTypeR(t)  $\wedge$  is-CompositeTypeR(t.shape)

```

The type of the state (if present) is a composite type, possibly with an invariant, and with a field for each state variable.

10.2 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-set
.4   inv mk-Env(sn, tm, ve, -)  $\triangleq$  (sn  $\neq$  nil  $\Rightarrow$  sn  $\in$  dom tm  $\wedge$  Is[StateTypeR, TypeR](tm(sn)))  $\wedge$ 
.5        $\forall n \in \text{dom } ve \cdot \text{is-DomR}(ve(n)) \Rightarrow n \in \text{dom } tm$ 

```

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 | OldName;
28.0   VisibleEnv = VisName  $\xrightarrow{m}$  StaticSemanticRep
.1   inv ve  $\triangleq$   $\forall n \in \text{dom } ve \cdot \text{is-OldName}(n) \Rightarrow \text{is-VarR}(ve(n))$ 

```

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 | DomR | PolyValR | 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-Env(nil, { $\mapsto$ }, { $\mapsto$ }, {})
```

10.2.1 Accessing Environments

- 35.0 $\text{StateVarNames} : \text{Env} \xrightarrow{t} \text{Name-set}$
 .1 $\text{StateVarNames}(\text{env}) \triangleq$
 .2 $\text{dom StateVisEnv}(\text{env});$
- 36.0 $\text{StateVisEnv} : \text{Env} \xrightarrow{t} \text{VisibleEnv}$
 .1 $\text{StateVisEnv}(\text{env}) \triangleq$
 .2 if $\text{env.statename} = \text{nil}$ then $\{\mapsto\}$
 .3 else cases $\text{env.typemap}(\text{env.statename})$:
 .4 $\text{mk-InvTypeR}(\text{mk-CompositeTypeR}(-, \text{fs}), -), \text{mk-CompositeTypeR}(-, \text{fs}) \rightarrow$
 .5 $\{\text{mk-Name}(\text{id}) \mapsto \text{mk-VarR}(\text{type}) \mid \text{mk-FieldR}(\text{id}, \text{type}) \in \text{elems } \text{fs} \cdot \text{id} \neq \text{nil}\}$
 .6 end;
- 37.0 $\text{IsBoundTo} : \text{VisName} \xrightarrow{t} \text{StaticSemanticRep} \rightarrow (\text{Env} \mid \text{VisibleEnv}) \rightarrow \mathbb{B}$
 .1 $\text{IsBoundTo}(n)(\text{ssr})(\text{env}) \triangleq$
 .2 let $\text{ve} = \text{if Is[VisibleEnv, Env} \mid \text{VisibleEnv}](\text{env}) \text{ then env else env.visibleenv}$ in
 .3 $n \in \text{dom ve} \wedge$
 .4 $\text{ssr} = \text{ve}(n);$
- 38.0 $\text{IsBoundToValTypeR} : \Pi \xrightarrow{t} \text{VisName} \rightarrow \text{TypeR} \rightarrow (\text{Env} \mid \text{VisibleEnv}) \rightarrow \mathbb{B}$
 .1 $\text{IsBoundToValTypeR}(\pi)(n)(t)(\text{env}) \triangleq$
 .2 $\text{IsBoundTo}(n)(\text{mk-ValR}(t, \pi))(\text{env}) \vee$
 .3 $\text{IsBoundTo}(n)(\text{mk-VarR}(t))(\text{env}) \vee$
 .4 $\text{is-POS}(\pi) \wedge \text{IsBoundTo}(n)(\text{mk-ValR}(t, \text{NegWfClass}(\pi)))(\text{env});$
- 39.0 $\text{IsBoundToDomTypeR} : \text{Name} \xrightarrow{t} \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsBoundToDomTypeR}(n)(t)(\text{env}) \triangleq$
 .2 $\text{IsVisibleTypeName}(n)(\text{env}) \wedge t = \text{env.typemap}(n);$
- 40.0 $\text{IsVisibleTypeVar} : \text{TypeVar} \xrightarrow{t} \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsVisibleTypeVar}(\text{tv})(\text{env}) \triangleq$
 .2 $\text{tv} \in \text{env.typevars};$
- 41.0 $\text{IsVisibleTypeName} : \text{Name} \xrightarrow{t} \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsVisibleTypeName}(n)(\text{env}) \triangleq$
 .2 $\text{IsBoundTo}(n)(\text{mk-DomR}(\text{LOOKINTYPEMAP}))(\text{env})$

10.2.2 Updating Environments

- 42.0 $\text{UpdateVisEnv} : \text{VisibleEnv} \xrightarrow{t} \text{VisibleEnv} \rightarrow \text{VisibleEnv}$
 .1 $\text{UpdateVisEnv}(\text{newdefs})(\text{oldenv}) \triangleq$
 .2 $\text{oldenv} \uplus \text{newdefs};$
- 43.0 $\text{UpdateEnv} : \text{VisibleEnv} \mid \text{TypeVar-set} \xrightarrow{t} \text{Env} \rightarrow \text{Env}$
 .1 $\text{UpdateEnv}(\text{new})(\text{oldenv}) \triangleq$
 .2 if $\text{Is[VisibleEnv, VisibleEnv} \mid \text{TypeVar}](\text{new})$
 .3 then $\mu(\text{oldenv}, \text{visibleenv} \mapsto \text{UpdateVisEnv}(\text{new})(\text{oldenv.visibleenv}))$
 .4 else $\mu(\text{oldenv}, \text{typevars} \mapsto \text{oldenv.typevars} \cup \text{new});$

44.0 $\text{RemoveFromEnv} : \text{Name-set} \xrightarrow{t} \text{Env} \rightarrow \text{Env}$
 .1 $\text{RemoveFromEnv}(\text{names})(\text{oldenv}) \triangleq$
 .2 $\mu(\text{oldenv}, \text{visibleenv} \mapsto \text{names} \Leftarrow \text{oldenv.visibleenv})$

10.3 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 $\Pi = \text{POS} \mid \text{DEF};$
 46.0 $\text{POS} :: \text{checkset} : \text{Check-set};$
 47.0 $\text{DEF} :: \text{checkset} : \text{Check-set};$
 48.0 $\text{Check} = \text{LOCALCONTEXT} \mid \text{TOTALFUNC} \mid \text{UNIONCLOSE}$

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 $\text{NegWfClass} : \Pi \xrightarrow{t} \Pi$
 .1 $\text{NegWfClass}(\pi) \triangleq$
 .2 cases $\pi :$
 .3 $\text{mk-POS}(cs) \rightarrow \text{mk-DEF}(cs),$
 .4 $\text{mk-DEF}(cs) \rightarrow \text{mk-POS}(cs)$
 .5 end

10.4 Type Relations

The definitions of various type relationships in this section are based on a notion of *TypeRelation* which is simply a finite set of type representation pairs:

50.0 $\text{TypeRelation} = (\text{TypeR} \times \text{TypeR})\text{-set}$

10.4.1 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 $\text{RelationKind} = \text{FULL} \mid \text{ESSENTIAL}$
- 52.0 $\text{IsFullSubtype} : \Pi \xrightarrow{t} \text{TypeR} \times \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{IsFullSubtype}(\pi)(t_1, t_2)(\text{env}) \triangleq$
 - .2 $\text{Is-x-Subtype}(\pi)(\text{FULL})(t_1, t_2)(\text{env});$
- 53.0 $\text{IsEssentialSubtype} : \Pi \xrightarrow{t} \text{TypeR} \times \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{IsEssentialSubtype}(\pi)(t_1, t_2)(\text{env}) \triangleq$
 - .2 $\text{Is-x-Subtype}(\pi)(\text{ESSENTIAL})(t_1, t_2)(\text{env})$
- 54.0 $\text{Is-x-Subtype} : \Pi \xrightarrow{t} \text{RelationKind} \rightarrow \text{TypeR} \times \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{Is-x-Subtype}(\pi)(\text{kind})(t_1, t_2)(\text{env}) \triangleq$
 - .2 $\text{CheckProof}(\pi)(\text{kind})(\text{DistributeTypeR}(t_1)(\text{env}), t_2)(\{\})\{\}(\text{env})$

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 . 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 $\text{CheckProof} :$
- .1 $\Pi \xrightarrow{t} \text{RelationKind} \rightarrow \text{TypeR} \times \text{TypeR} \rightarrow \text{TypeRelation} \rightarrow \text{TypeRelation} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 - .2 $\text{CheckProof}(\pi)(\text{kind})(t_1, t_2)(\text{pos})(\text{neg})(\text{env}) \triangleq$
 - .3 $\text{CheckProofBasic}(\pi)(t_1, t_2)(\text{env}) \vee$
 - .4 $\text{CheckProofStructural}(\pi)(\text{kind})(t_1, t_2)(\text{pos})(\text{neg})(\text{env}) \vee$
 - .5 $\text{CheckProofName}(\pi)(\text{kind})(t_1, t_2)(\text{pos})(\text{neg})(\text{env}) \vee$
 - .6 $\text{CheckProofInv}(\pi)(\text{kind})(t_1, t_2)(\text{pos})(\text{neg})(\text{env}) \vee$
 - .7 $\text{CheckProofUnion}(\pi)(\text{kind})(t_1, t_2)(\text{pos})(\text{neg})(\text{env}) \vee$
 - .8 $\text{CheckProofDistUnion}(\pi)(t_1, t_2)(\text{env}) \vee$
 - .9 $\text{CheckProofTypeVar}(\pi)(t_1, t_2) \vee$
 - .10 $\text{Is-x-EmptyType}(\pi)(\text{kind})(t_1)(\text{env})$

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 :  $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     CheckProofBasic ( $\pi$ ) ( $t_1, t_2$ ) ( $env$ )  $\triangleq$ 
.2       cases  $mk$ - ( $t_1, t_2$ ) :
.3          $mk$ - ((RAT), -)  $\rightarrow t_2 = REAL$ ,
.4          $mk$ - ((INT), -)  $\rightarrow t_2 \in \{RAT, REAL\}$ ,
.5          $mk$ - ((NAT), -)  $\rightarrow t_2 \in \{INT, RAT, REAL\}$ ,
.6          $mk$ - ((NATONE), -)  $\rightarrow t_2 \in \{NAT, INT, RAT, REAL\}$ ,
.7          $mk$ - (-, (ANY))  $\rightarrow true$ ,
.8          $mk$ - ((EMPTYSET), -)  $\rightarrow is-SetTypeR(t_2)$ ,
.9          $mk$ - ((EMPTYSEQ), -)  $\rightarrow is-Seq0TypeR(t_2)$ ,
.10         $mk$ - ((EMPTYSMAP), -)  $\rightarrow is-GeneralMapTypeR(t_2) \vee is-InjectiveMapTypeR(t_2)$ ,
.11         $mk$ - (-, (EMPTYSET))  $\rightarrow is-POS(\pi) \wedge$ 
.12           $is-SetTypeR(t_1) \wedge IsEmptyType(\pi)(t_1.elemtp)(env)$ ,
.13         $mk$ - (-, (EMPTYSEQ))  $\rightarrow is-POS(\pi) \wedge$ 
.14           $is-Seq0TypeR(t_1) \wedge IsEmptyType(\pi)(t_1.elemtp)(env)$ ,
.15         $mk$ - (-, (EMPTYSMAP))  $\rightarrow$ 
.16           $is-POS(\pi) \wedge (is-GeneralMapTypeR(t_1) \vee is-InjectiveMapTypeR(t_1)) \wedge$ 
.17           $(IsEmptyType(\pi)(t_1.mapdom)(env) \vee IsEmptyType(\pi)(t_1.maprng)(env))$ ,
.18        others  $\rightarrow t_1 = t_2$ 
.19      end

```

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     $\Pi \xrightarrow{t} RelationKind \rightarrow TypeR \times TypeR \rightarrow TypeRelation \rightarrow TypeRelation \rightarrow Env \rightarrow \mathbb{B}$ 
.2  CheckProofStructural ( $\pi$ )( $kind$ )( $t_1, t_2$ )( $pos$ )( $neg$ )( $env$ )  $\triangleq$ 
.3  let check =  $\lambda t_1 : TypeR, t_2 : TypeR .$ 
.4    CheckProof( $\pi$ )( $kind$ )( $t_1, t_2$ )( $pos$ )( $neg$ )( $env$ ) in
.5  cases mk-( $t_1, t_2$ ) :
.6    mk-(mk-SetTypeR( $et_1$ ), mk-SetTypeR( $et_2$ ))  $\rightarrow$  check( $et_1, et_2$ ),
.7    mk-(mk-Seq0TypeR( $et_1$ ),  $-$ )  $\rightarrow$ 
.8      check(mk-UnionTypeR({EMPTYSEQ, mk-Seq1TypeR( $et_1$ )}),  $t_2$ ),
.9    mk-(-, mk-Seq0TypeR( $et_2$ ))  $\rightarrow$ 
.10   check( $t_1, mk-UnionTypeR({EMPTYSEQ, mk-Seq1TypeR( $et_2$ )})$ )),
.11   mk-(mk-Seq1TypeR( $et_1$ ), mk-Seq0TypeR( $et_2$ ))  $\rightarrow$  check( $et_1, et_2$ ),
.12   mk-(mk-Seq1TypeR( $et_1$ ), mk-Seq1TypeR( $et_2$ ))  $\rightarrow$  check( $et_1, et_2$ ),
.13   mk-(mk-ProductTypeR( $ts_1$ ), mk-ProductTypeR( $ts_2$ ))  $\rightarrow$ 
.14   len  $ts_1 =$  len  $ts_2 \wedge \forall i \in \text{inds } ts_1 \cdot \text{check}(ts_1(i), ts_2(i))$ ,
.15   mk-(mk-CompositeTypeR( $n_1, fs_1$ ), mk-CompositeTypeR( $n_2, fs_2$ ))  $\rightarrow$ 
.16    $n_1 = n_2 \wedge \text{len } fs_1 = \text{len } fs_2 \wedge$ 
.17    $\forall i \in \text{inds } fs_1 \cdot fs_1(i).id = fs_2(i).id \wedge \text{check}(fs_1(i).type, fs_2(i).type)$ ,
.18   mk-(mk-PartialFnTypeR( $at_1, rt_1$ ), mk-PartialFnTypeR( $at_2, rt_2$ ))  $\rightarrow$ 
.19   check( $rt_1, rt_2$ )  $\wedge$  check( $at_1, at_2$ )  $\wedge$  (check( $at_2, at_1$ )  $\vee$  is-POS( $\pi$ )),
.20   mk-(mk-TotalFnTypeR( $at_1, rt_1$ ), mk-PartialFnTypeR( $at_2, rt_2$ ))  $\rightarrow$ 
.21   check( $rt_1, rt_2$ )  $\wedge$  check( $at_1, at_2$ )  $\wedge$  (check( $at_2, at_1$ )  $\vee$  is-POS( $\pi$ )),
.22   mk-(mk-TotalFnTypeR( $at_1, rt_1$ ), mk-TotalFnTypeR( $at_2, rt_2$ ))  $\rightarrow$ 
.23   check( $rt_1, rt_2$ )  $\wedge$  check( $at_1, at_2$ )  $\wedge$  check( $at_2, at_1$ ),
.24   mk-(mk-GeneralMapTypeR( $at_1, rt_1$ ), mk-GeneralMapTypeR( $at_2, rt_2$ ))  $\rightarrow$ 
.25   check( $at_1, at_2$ )  $\wedge$  check( $rt_1, rt_2$ ),
.26   mk-(mk-GeneralMapTypeR( $at_1, rt_1$ ), mk-InjectiveMapTypeR( $at_2, rt_2$ ))  $\rightarrow$ 
.27   check( $at_1, at_2$ )  $\wedge$  check( $rt_1, rt_2$ )  $\wedge$ 
.28   IsOneValueType( $\pi$ )( $at_1$ )( $env$ ),
.29   mk-(mk-InjectiveMapTypeR( $at_1, rt_1$ ), mk-GeneralMapTypeR( $at_2, rt_2$ ))  $\rightarrow$ 
.30   check( $at_1, at_2$ )  $\wedge$  check( $rt_1, rt_2$ ),
.31   mk-(mk-InjectiveMapTypeR( $at_1, rt_1$ ), mk-InjectiveMapTypeR( $at_2, rt_2$ ))  $\rightarrow$ 
.32   check( $at_1, at_2$ )  $\wedge$  check( $rt_1, rt_2$ ),
.33   mk-(mk-QuoteTypeR( $q_1$ ), mk-QuoteTypeR( $q_2$ ))  $\rightarrow q_1 = q_2$ ,
.34   mk-(mk-RetTypeR( $et_1$ ), mk-RetTypeR( $et_2$ ))  $\rightarrow$  check( $et_1, et_2$ ),
.35   mk-(mk-ExitTypeR( $et_1$ ), mk-ExitTypeR( $et_2$ ))  $\rightarrow$  check( $et_1, et_2$ ),
.36   mk-(mk-OpTypeR( $st_1, at_1, rt_1$ ), mk-OpTypeR( $st_2, at_2, rt_2$ ))  $\rightarrow$ 
.37   check( $rt_1, rt_2$ )  $\wedge$  check( $at_1, at_2$ )  $\wedge$  check( $at_2, at_1$ )  $\wedge$ 
.38   check( $st_1, st_2$ )  $\wedge$  check( $st_2, st_1$ ),
.39  others  $\rightarrow$  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 .

58.0 *CheckProofName* :

- .1 $\Pi \xrightarrow{t} RelationKind \rightarrow TypeR \times TypeR \rightarrow TypeRelation \rightarrow TypeRelation \rightarrow Env \rightarrow \mathbb{B}$
- .2 $CheckProofName(\pi)(kind)(t_1, t_2)(pos)(neg)(env) \triangleq$
 $is-Name(t_1) \wedge$
 $(mk-(t_1, t_2)) \in pos \vee$
 $\exists t : TypeR \cdot IsBoundToDomTypeR(t_1)(t)(env) \wedge$
 $CheckProof(\pi)(kind)(t, t_2)(pos \cup \{mk-(t_1, t_2)\})(neg)(env))$
- .7 \vee
- .8 $is-Name(t_2) \wedge$
- .9 $\exists t : TypeR \cdot IsBoundToDomTypeR(t_2)(t)(env) \wedge$
 $mk-(t_1, t_2) \notin neg \wedge$
 $CheckProof(\pi)(kind)(t_1, t)(pos)(neg \cup \{mk-(t_1, t_2)\})(env)$
- .11

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 $\Pi \xrightarrow{t} RelationKind \rightarrow TypeR \times TypeR \rightarrow TypeRelation \rightarrow TypeRelation \rightarrow Env \rightarrow \mathbb{B}$
- .2 $CheckProofInv(\pi)(kind)(t_1, t_2)(pos)(neg)(env) \triangleq$
 $cases mk-(t_1, t_2) :$
 $mk-(-, mk-InvTypeR(et_2, -)) \rightarrow$
 $CheckProof(\pi)(kind)(t_1, et_2)(pos)(neg)(env) \wedge is-POS(\pi),$
 $mk-(mk-InvTypeR(et_1, -), -) \rightarrow$
 $CheckProof(\pi)(kind)(et_1, t_2)(pos)(neg)(env) \vee$
 $is-POS(\pi) \wedge kind = FULL,$
 $others \rightarrow false$
- .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 $mk-InvTypeR(et_1, -)$ 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

$$T = \text{char} \mid \mathbb{N}$$

$$\text{inv } x \triangleq \text{true}$$

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 $\Pi \xrightarrow{t} RelationKind \rightarrow TypeR \times TypeR \rightarrow TypeRelation \rightarrow TypeRelation \rightarrow Env \rightarrow \mathbb{B}$
 .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 $\exists t \in t_2.summands \cdot CheckProof(\pi)(kind)(t_1, t)(pos)(neg)(env)$

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* : $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $CheckProofDistUnion(\pi)(t_1, t_2)(env) \triangleq$
 .2 $is\text{-POS}(\pi) \wedge$
 .3 $(is\text{-ProductTypeR}(t_1) \vee is\text{-PartialFnTypeR}(t_1) \vee is\text{-TotalFnTypeR}(t_1) \vee$
 .4 $is\text{-GeneralMapTypeR}(t_1) \vee is\text{-InjectiveMapTypeR}(t_1))$
 .5 \wedge
 .6 $IsEquivToUnionTypeR(\pi)(t_1)(env) \wedge$
 .7 $is\text{-UnionTypeR}(t_2) \wedge$
 .8 $\exists s_1, s_2 \in t_2.summands \cdot$
 .9 $s_1 \neq s_2 \wedge IsOverlapping(\pi)(s_1, t_1)(env) \wedge IsOverlapping(\pi)(s_2, t_1)(env)$

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* : $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow \mathbb{B}$
 .1 $CheckProofTypeVar(\pi)(t_1, t_2) \triangleq$
 .2 $is\text{-POS}(\pi) \wedge (is\text{-TypeVarId}(t_1) \vee is\text{-TypeVarId}(t_2))$

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 :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   IsEquivToUnionTypeR( $\pi$ )( $t$ )( $env$ )  $\triangleq$ 
.2    $\neg IsEmptyType(NegWfClass(\pi))(t)(env) \wedge$ 
.3   cases  $t$  :
.4      $mk\text{-Name}(\cdot) \rightarrow IsEquivToUnionTypeR(\pi)(env.typemap(t))(env)$ ,
.5      $mk\text{-UnionTypeR}(\cdot), mk\text{-Seq0TypeR}(\cdot) \rightarrow true$ ,
.6      $mk\text{-ProductTypeR}(ts) \rightarrow \exists i \in \text{inds } ts \cdot IsEquivToUnionTypeR(\pi)(ts(i))(env)$ ,
.7      $mk\text{-PartialFnTypeR}(at, rt), mk\text{-TotalFnTypeR}(at, rt) \rightarrow$ 
.8        $is\text{-POS}(\pi) \wedge IsFiniteTypeR(\pi)(at)(env) \wedge IsEquivToUnionTypeR(\pi)(rt)(env)$ ,
.9      $mk\text{-GeneralMapTypeR}(at, rt) \rightarrow$ 
.10     $is\text{-POS}(\pi) \wedge IsFiniteTypeR(\pi)(at)(env) \wedge IsEquivToUnionTypeR(\pi)(rt)(env)$ ,
.11     $mk\text{-InjectiveMapTypeR}(at, rt) \rightarrow$ 
.12       $is\text{-POS}(\pi) \wedge$ 
.13       $(IsFiniteTypeR(\pi)(at)(env) \wedge IsEquivToUnionTypeR(\pi)(rt)(env)) \vee$ 
.14       $IsFiniteTypeR(\pi)(rt)(env) \wedge IsEquivToUnionTypeR(\pi)(at)(env)$ ),
.15     $mk\text{-InvTypeR}(t_1, \cdot) \rightarrow IsEquivToUnionTypeR(\pi)(t_1)(env)$ ,
.16     $mk\text{-TypeVarId}(\cdot) \rightarrow is\text{-POS}(\pi)$ ,
.17    others  $\rightarrow false$ 
.18  end

```

This function is used for checking whether a product, function or map type t can be rewritten as a union of other types solely on account of distributivity over union. Note, however, that in this context, a *Seq0* type is considered a union of *EMPTYSEQ* and a *Seq1* type.

10.4.2 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   Is-x-OverlappingSubtype( $\pi$ )(ESSENTIAL)( $t_1, t_2$ )( $env$ );
65.0  Is-x-OverlappingSubtype :  $\Pi \xrightarrow{t} RelationKind \rightarrow TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   Is-x-OverlappingSubtype( $\pi$ )(kind)( $t_1, t_2$ )( $env$ )  $\triangleq$ 
.2   Is-x-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    $\neg IsDisjoint(NegWfClass(\pi))(t_1, t_2)(env)$ ;
67.0  IsDisjoint :  $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   IsDisjoint( $\pi$ )( $t_1, t_2$ )( $env$ )  $\triangleq$ 
.2    $\exists drel : TypeRelation \cdot IsDisjointnessRel(\pi)(drel)(env) \wedge mk\text{-}(t_1, t_2) \in drel$ 

```

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    $\forall mk\text{-}(t_1, t_2) \in drel \cdot CheckPairDisjoint(\pi)(t_1, t_2)(drel)(env)$ 
```

When checking a disjointness proof, each subgoal must be verified. This is done by the auxiliary function *CheckPairDisjoint*.

```
69.0  CheckPairDisjoint :  $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow TypeRelation \rightarrow Env \rightarrow \mathbb{B}$ 
.1   CheckPairDisjoint( $\pi$ )( $t_1, t_2$ )( $drel$ )( $env$ )  $\triangleq$ 
.2   cases  $mk\text{-}(t_1, t_2)$  :
.3      $mk\text{-}(mk\text{-}SetTypeR(et_1), mk\text{-}SetTypeR(et_2)) \rightarrow \text{false},$ 
.4      $mk\text{-}(mk\text{-}Seq0TypeR(et_1), mk\text{-}Seq0TypeR(et_2)) \rightarrow \text{false},$ 
.5      $mk\text{-}(mk\text{-}Seq1TypeR(et_1), mk\text{-}Seq0TypeR(et_2)) \rightarrow mk\text{-}(et_1, et_2) \in drel,$ 
.6      $mk\text{-}(mk\text{-}Seq0TypeR(et_1), mk\text{-}Seq1TypeR(et_2)) \rightarrow mk\text{-}(et_1, et_2) \in drel,$ 
.7      $mk\text{-}(mk\text{-}Seq1TypeR(et_1), mk\text{-}Seq1TypeR(et_2)) \rightarrow mk\text{-}(et_1, et_2) \in drel,$ 
.8      $mk\text{-}(mk\text{-}ProductTypeR(ts_1), mk\text{-}ProductTypeR(ts_2)) \rightarrow$ 
.9        $\text{len } ts_1 \neq \text{len } ts_2 \vee \exists i \in \text{inds } ts_1 \cdot mk\text{-}(ts_1(i), ts_2(i)) \in drel,$ 
.10     $mk\text{-}(mk\text{-}CompositeTypeR(n_1, fs_1), mk\text{-}CompositeTypeR(n_2, fs_2)) \rightarrow$ 
.11       $n_1 \neq n_2 \vee \text{len } fs_1 \neq \text{len } fs_2 \vee$ 
.12       $\exists i \in \text{inds } fs_1 \cdot$ 
.13         $fs_1(i).id \neq fs_2(i).id \vee mk\text{-}(fs_1(i).type, fs_2(i).type) \in drel,$ 
.14         $mk\text{-}(mk\text{-}GeneralMapTypeR(at_1, rt_1), mk\text{-}GeneralMapTypeR(at_2, rt_2)) \rightarrow \text{false},$ 
.15         $mk\text{-}(mk\text{-}GeneralMapTypeR(at_1, rt_1), mk\text{-}InjectiveMapTypeR(at_2, rt_2)) \rightarrow \text{false},$ 
.16         $mk\text{-}(mk\text{-}InjectiveMapTypeR(at_1, rt_1), mk\text{-}GeneralMapTypeR(at_2, rt_2)) \rightarrow \text{false},$ 
.17         $mk\text{-}(mk\text{-}InjectiveMapTypeR(at_1, rt_1), mk\text{-}InjectiveMapTypeR(at_2, rt_2)) \rightarrow \text{false},$ 
.18         $mk\text{-}(mk\text{-}QuoteTypeR(q_1), mk\text{-}QuoteTypeR(q_2)) \rightarrow q_1 \neq q_2,$ 
.19         $mk\text{-}(mk\text{-}RetTypeR(et_1), mk\text{-}RetTypeR(et_2)) \rightarrow mk\text{-}(et_1, et_2) \in drel,$ 
.20         $mk\text{-}(mk\text{-}ExitTypeR(et_1), mk\text{-}ExitTypeR(et_2)) \rightarrow mk\text{-}(et_1, et_2) \in drel,$ 
.21         $mk\text{-}(mk\text{-}OpTypeR(st_1, at_1, rt_1), mk\text{-}OpTypeR(st_2, at_2, rt_2)) \rightarrow$ 
.22           $(mk\text{-}(rt_1, rt_2) \in drel \vee mk\text{-}(st_1, st_2) \in drel) \wedge$ 
.23           $\neg EmptyOpOverlap(NegWfClass(\pi))(t_1, t_2)(env),$ 
.24        others  $\rightarrow$  if  $Is[FunctionTypeR, TypeR](t_1) \wedge Is[FunctionTypeR, TypeR](t_2)$ 
.25          then CheckPairFunctionDisjoint( $\pi$ )( $t_1, t_2$ )( $drel$ )( $env$ )
.26          else CheckPairNonMatchDisjoint( $\pi$ )( $t_1, t_2$ )( $drel$ )( $env$ )
.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*.

```

70.0  CheckPairNonMatchDisjoint :  $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow TypeRelation \rightarrow Env \rightarrow \mathbb{B}$ 
.1   CheckPairNonMatchDisjoint ( $\pi$ ) ( $t_1, t_2$ ) ( $drel$ ) ( $env$ )  $\triangleq$ 
.2   cases  $mk-(t_1, t_2)$  :
.3      $mk-(mk-Name(i), t), mk-(t, mk-Name(i)) \rightarrow$ 
.4        $\exists t' : TypeR \cdot IsBoundToDomTypeR(mk-Name(i))(t')(env) \wedge mk-(t, t') \in drel,$ 
.5      $mk-(mk-UnionTypeR(ts), t), mk-(t, mk-UnionTypeR(ts)) \rightarrow$ 
.6        $\forall t' \in ts \cdot mk-(t, t') \in drel,$ 
.7      $mk-(mk-InvTypeR(t', -), t), mk-(t, mk-InvTypeR(t', -)) \rightarrow$ 
.8        $mk-(t, t') \in drel \vee is-POS(\pi),$ 
.9      $mk-(-, (ANY)), mk-((ANY), -) \rightarrow \text{false},$ 
.10     $mk-(-, (VOID)), mk-((VOID), -) \rightarrow \text{true},$ 
.11     $mk-((EMPTYSET), t), mk-(t, (EMPTYSET)) \rightarrow$ 
.12       $\neg is-SetTypeR(t),$ 
.13     $mk-((EMPTYSKIP), t), mk-(t, (EMPTYSKIP)) \rightarrow$ 
.14       $\neg (is-GeneralMapTypeR(t) \vee is-InjectiveMapTypeR(t)),$ 
.15     $mk-((EMPTYSEQ), t), mk-(t, (EMPTYSEQ)) \rightarrow$ 
.16       $\neg is-Seq0TypeR(t),$ 
.17     $mk-(-, mk-TypeVarId(-)), mk-(mk-TypeVarId(-), -) \rightarrow is-POS(\pi),$ 
.18    others  $\rightarrow t_1 \neq t_2 \wedge$ 
.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 preceding cases are encountered, a syntactical difference between t_1 and t_2 implies their disjointness.

```

71.0  CheckPairFunctionDisjoint :
.1    $\Pi \xrightarrow{t} FunctionTypeR \times FunctionTypeR \rightarrow TypeRelation \rightarrow Env \rightarrow \mathbb{B}$ 
.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      $(mk-(rt_1, rt_2)) \in drel \vee$ 
.8      $is-TotalFnTypeR(t_1) \wedge is-TotalFnTypeR(t_2) \wedge mk-(at_1, at_2) \in drel \vee$ 
.9      $is-POS(\pi) \wedge at_1 \neq at_2$ 

```

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.

```

72.0  EmptyFnOverlap :  $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   EmptyFnOverlap ( $\pi$ ) ( $t_1, t_2$ ) ( $env$ )  $\triangleq$ 
.2      $IsTypeOfEmptyFn(\pi)(t_1)(env) \wedge$ 
.3      $IsTypeOfEmptyFn(\pi)(t_2)(env)$ 

```

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 :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
    .1  IsTypeOfEmptyFn ( $\pi$ ) (type) (env)  $\triangleq$ 
    .2  cases type :
    .3      mk-PartialFnTypeR (at, -), mk-TotalFnTypeR (at, -)  $\rightarrow$ 
    .4          Is-x-EmptyType ( $\pi$ ) (ESSENTIAL) (at) (env),
    .5          mk-TypeVarId (-)  $\rightarrow$  is-POS ( $\pi$ ),
    .6          others  $\rightarrow$  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 :  $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
    .1  EmptyOpOverlap ( $\pi$ ) (t1, t2) (env)  $\triangleq$ 
    .2      IsTypeOfEmptyOp ( $\pi$ ) (t1) (env)  $\wedge$ 
    .3      IsTypeOfEmptyOp ( $\pi$ ) (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 :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
    .1  IsTypeOfEmptyOp ( $\pi$ ) (type) (env)  $\triangleq$ 
    .2  cases type :
    .3      mk-OpTypeR (stt, at, rt)  $\rightarrow$ 
    .4          let at1 = mk-ProductTypeR ([stt, at]),
    .5          rt1 = mk-ProductTypeR ([stt, rt]) in
    .6          IsTypeOfEmptyFn ( $\pi$ ) (mk-PartialFnTypeR (at1, rt1)) (env),
    .7          others  $\rightarrow$  false
    .8      end

```

This predicate tests whether a type is an operation type containing just the empty operation.

10.4.3 Auxiliary Type Relations and Functions

```

76.0  IsMapTypeR :  $TypeR \xrightarrow{t} TypeR \times TypeR \rightarrow \mathbb{B}$ 
    .1  IsMapTypeR (mt) (dt, rt)  $\triangleq$ 
    .2      mt = mk-GeneralMapTypeR (dt, rt)  $\vee$ 
    .3      mt = mk-InjectiveMapTypeR (dt, rt)  $\vee$  mt = EMPTYMAP;

```



```

77.0  IsFunctionTypeR :  $TypeR \xrightarrow{t} TypeR \times TypeR \rightarrow \mathbb{B}$ 
    .1  IsFunctionTypeR (ft) (dt, rt)  $\triangleq$ 
    .2      ft = mk-PartialFnTypeR (dt, rt)  $\vee$ 
    .3      ft = mk-TotalFnTypeR (dt, rt);

```



```

78.0  IsSeqTypeR :  $TypeR \xrightarrow{t} TypeR \rightarrow \mathbb{B}$ 
    .1  IsSeqTypeR (seqt) (et)  $\triangleq$ 
    .2      seqt = mk-Seq0TypeR (et)  $\vee$ 
    .3      seqt = mk-Seq1TypeR (et)  $\vee$  seqt = EMPTYSEQ;

```

79.0 $\text{LengthofSeq} : \Pi \xrightarrow{t} \text{TypeR} \times \mathbb{N} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{LengthofSeq}(\pi)(\text{seqtype}, \text{seqlength})(\text{env}) \triangleq$
 .2 $\text{is-POS}(\pi)$

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 $\text{CardofSet} : \Pi \xrightarrow{t} \text{TypeR} \times \mathbb{N} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{CardofSet}(\pi)(\text{settype}, \text{setcard})(\text{env}) \triangleq$
 .2 $\text{is-POS}(\pi)$

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 $\text{IsNonEmptySet} : \Pi \xrightarrow{t} \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsNonEmptySet}(\pi)(\text{type})(\text{env}) \triangleq$
 .2 $\text{is-POS}(\pi) \wedge \neg \text{type} = \text{EMPTYSET}$

This function checks that *settype* only represents nonempty set values.

82.0 $\text{IsEquivalent} : \Pi \xrightarrow{t} \text{TypeR} \times \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsEquivalent}(\pi)(t_1, t_2)(\text{env}) \triangleq$
 .2 $\text{IsFullSubtype}(\pi)(t_1, t_2)(\text{env}) \wedge \text{IsFullSubtype}(\pi)(t_2, t_1)(\text{env})$

Here type equivalence is defined as the mutual, full subtype relationship.

83.0 $\text{IsEssEquivTypeR} : \Pi \rightarrow \text{TypeR} \times \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsEssEquivTypeR}(\pi)(t_1, t_2)(\text{env}) \triangleq$
 .2 $\text{IsEssentialSubtype}(\pi)(t_1, t_2)(\text{env}) \wedge$
 .3 $\text{IsFullSubtype}(\pi)(t_2, t_1)(\text{env});$

84.0 $\text{IsNonEmptyEssEquivTypeR} : \Pi \rightarrow \text{TypeR} \times \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsNonEmptyEssEquivTypeR}(\pi)(t_1, t_2)(\text{env}) \triangleq$
 .2 $\text{IsSubtypeR}(\pi)(t_1, t_2)(\text{env}) \wedge$
 .3 $\text{IsFullSubtype}(\pi)(t_2, t_1)(\text{env})$

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.

85.0 $\text{IsIntersectionType} : \Pi \xrightarrow{t} \text{TypeR} \times \text{TypeR} \rightarrow \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsIntersectionType}(\pi)(t_1, t_2)(t)(\text{env}) \triangleq$
 .2 cases π :
 .3 $\text{mk-POS}(\cdot) \rightarrow \text{IsFullSubtype}(\pi)(t, t_1)(\text{env}) \wedge \text{IsFullSubtype}(\pi)(t, t_2)(\text{env}),$
 .4 $\text{mk-DEF}(\cdot) \rightarrow t = t_1 \vee t = t_2$
 .5 end

$\text{IsIntersectionType}(\pi)(t_1, t_2)(t)(\text{env})$ 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  $\pi$  :
.3          $mk\text{-POS}(\cdot) \rightarrow t_1 = t_2,$ 
.4          $mk\text{-DEF}(\cdot) \rightarrow IsEquivalent(\pi)(t_1, t_2)(env) \wedge IsOneValueType(\pi)(t_1)(env)$ 
.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       Is-x-EmptyType ( $\pi$ ) ( $FULL$ ) ( $t$ ) ( $env$ );

```

88.0 *Is-x-EmptyType* : $\Pi \xrightarrow{t} RelationKind \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

```

.1     Is-x-EmptyType ( $\pi$ ) ( $kind$ ) ( $t$ ) ( $env$ )  $\triangleq$ 
.2       IsEmptyAux ( $\pi$ ) ( $kind$ ) ( $t$ ) ( $env$ ) ( $\{\}$ );

```

89.0 *IsEmptyAux* : $\Pi \xrightarrow{t} RelationKind \rightarrow TypeR \rightarrow Env \rightarrow TypeR\text{-set} \rightarrow \mathbb{B}$

```

.1     IsEmptyAux ( $\pi$ ) ( $kind$ ) ( $t$ ) ( $env$ ) ( $empty\text{-types}$ )  $\triangleq$ 
.2        $t \in empty\text{-types} \vee$ 
.3       let  $empty\text{-types}_1 = empty\text{-types} \cup \{t\}$  in
.4       cases  $t$  :
.5          $mk\text{-Name}(\cdot) \rightarrow IsEmptyAux(\pi)(kind)(env.typeimap(t))(env)(empty\text{-types}_1),$ 
.6          $mk\text{-Seq1TypeR}(t_1) \rightarrow IsEmptyAux(\pi)(kind)(t_1)(env)(empty\text{-types}_1),$ 
.7          $mk\text{-UnionTypeR}(ts) \rightarrow \forall t_1 \in ts \cdot IsEmptyAux(\pi)(kind)(t_1)(env)(empty\text{-types}_1),$ 
.8          $mk\text{-ProductTypeR}(ts) \rightarrow$ 
.9            $\exists t_1 \in elems\ ts \cdot IsEmptyAux(\pi)(kind)(t_1)(env)(empty\text{-types}_1),$ 
.10           $mk\text{-CompositeTypeR}(\cdot, fs) \rightarrow$ 
.11             $\exists mk\text{-FieldR}(\cdot, et) \in elems\ fs \cdot IsEmptyAux(\pi)(kind)(et)(env)(empty\text{-types}_1),$ 
.12           $mk\text{-InvTypeR}(et, \cdot) \rightarrow$ 
.13             $IsEmptyAux(\pi)(kind)(et)(env)(empty\text{-types}_1) \vee is\text{-POS}(\pi) \wedge kind = FULL,$ 
.14           $mk\text{-TotalFnTypeR}(at, rt) \rightarrow$ 
.15             $Is-x\text{-EmptyType}(\pi)(kind)(rt)(env) \wedge$ 
.16             $\neg Is-x\text{-EmptyType}(NegWfClass(\pi))(kind)(at)(env),$ 
.17           $mk\text{-PartialFnTypeR}(\cdot, \cdot) \rightarrow$ 
.18            false,
.19           $mk\text{-OpTypeR}(stt, at, rt) \rightarrow$ 
.20            false,
.21           $mk\text{-RetTypeR}(et), mk\text{-ExitTypeR}(et) \rightarrow IsEmptyAux(\pi)(kind)(et)(env)(empty\text{-types}_1),$ 
.22           $mk\text{-TypeVarId}(\cdot) \rightarrow is\text{-POS}(\pi),$ 
.23          others  $\rightarrow$  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 emptyness, 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 :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     IsOneValueType ( $\pi$ )( $type$ )( $env$ )  $\triangleq$ 
.2     IsOneValueTypeAux( $\pi$ )( $type$ )( $env$ )( $\{\}$ );

91.0   IsOneValueTypeAux :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow TypeR\text{-set} \rightarrow \mathbb{B}$ 
.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-Name(-)  $\rightarrow$  IsOneValueTypeAux( $\pi$ )( $env.typeMap(type)$ )( $env$ )( $visitedtypes_1$ ),
.6       mk-InvTypeR( $et$ , -)  $\rightarrow$   $\neg$  IsEmptyType(NegWfClass( $\pi$ ))( $et$ )( $env$ )  $\wedge$  is-POS( $\pi$ ),
.7       mk-QuoteTypeR(-)  $\rightarrow$  true,
.8       mk-CompositeTypeR(-,  $fields$ )  $\rightarrow$ 
.9          $\forall f \in elems fields \cdot IsOneValueTypeAux(\pi)(f.type)(env)(visitedtypes_1),$ 
.10      mk-UnionTypeR( $ts$ )  $\rightarrow$ 
.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-ProductTypeR( $factors$ )  $\rightarrow$ 
.16             $\forall t \in elems factors \cdot IsOneValueTypeAux(\pi)(t)(env)(visitedtypes_1),$ 
.17            mk-SetTypeR( $et$ ), mk-Seq0Type( $et$ )  $\rightarrow$  IsEmptyType( $\pi$ )( $et$ )( $env$ ),
.18            mk-GeneralMapTypeR( $at, rt$ ), mk-InjectiveMapTypeR( $at, rt$ )  $\rightarrow$ 
.19              IsEmptyType( $\pi$ )( $at$ )( $env$ )  $\vee$  IsEmptyType( $\pi$ )( $rt$ )( $env$ ),
.20            mk-PartialFnTypeR( $at, rt$ )  $\rightarrow$ 
.21              IsEmptyType( $\pi$ )( $at$ )( $env$ )  $\vee$  IsEmptyType( $\pi$ )( $rt$ )( $env$ ),
.22            mk-TotalFnTypeR( $at, rt$ )  $\rightarrow$ 
.23              IsEmptyType( $\pi$ )( $at$ )( $env$ )  $\vee$ 
.24                is-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-RetTypeR( $et$ ), mk-ExitTypeR( $et$ )  $\rightarrow$  IsOneValueTypeAux( $\pi$ )( $et$ )( $env$ )( $visitedtypes_1$ ),
.28            (EMPTYSET), (EMPTYSEQ), (EMPTYSMAP), (UNITTYPE), (CONT), (VOID)  $\rightarrow$  true,
.29            mk-TypeVarId(-)  $\rightarrow$  is-POS( $\pi$ ),
.30            others  $\rightarrow$  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.

```

92.0  IsFiniteTypeR :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   IsFiniteTypeR( $\pi$ )( $t$ )( $env$ )  $\triangleq$ 
.2   IsEmptyType( $\pi$ )( $t$ )( $env$ )  $\vee$  IsOneValueType( $\pi$ )( $t$ )( $env$ )  $\vee$ 
.3   cases  $t$  :
.4     mk-Name(-)  $\rightarrow$ 
.5       let  $nt = env.typeMap(t)$  in
.6       cases  $\pi$  :
.7         mk-DEF(-)  $\rightarrow$  DoesNotBuildOn( $\pi$ )( $nt, t$ )( $env$ )  $\wedge$  IsFiniteTypeR( $\pi$ )( $nt$ )( $env$ ),
.8         mk-POS(-)  $\rightarrow$  BuildsOn( $\pi$ )( $nt, t$ )( $env$ )  $\vee$  IsFiniteTypeR( $\pi$ )( $nt$ )( $env$ )
.9       end,
.10      mk-InvTypeR( $et, -$ )  $\rightarrow$  is-POS( $\pi$ )  $\vee$  IsFiniteTypeR( $\pi$ )( $et$ )( $env$ ),
.11      mk-CompositeTypeR(-, fields)  $\rightarrow$ 
.12         $\forall f \in \text{elems } fields \cdot IsFiniteTypeR(\pi)(f.type)(env)$ ,
.13      mk-UnionTypeR( $ts$ )  $\rightarrow$ 
.14         $\forall t \in ts \cdot IsFiniteTypeR(\pi)(t)(env)$ ,
.15      mk-ProductTypeR(factors)  $\rightarrow$ 
.16         $\forall t \in \text{elems } factors \cdot IsFiniteTypeR(\pi)(t)(env)$ ,
.17      mk-SetTypeR( $et$ )  $\rightarrow$  IsFiniteTypeR( $\pi$ )( $et$ )( $env$ ),
.18      mk-Seq0TypeR( $et$ )  $\rightarrow$  IsEmptyType( $\pi$ )( $et$ )( $env$ ),
.19      mk-GeneralMapTypeR( $at, rt$ ), mk-InjectiveMapTypeR( $at, rt$ )  $\rightarrow$ 
.20        IsFiniteTypeR( $\pi$ )( $at$ )( $env$ )  $\wedge$  IsFiniteTypeR( $\pi$ )( $rt$ )( $env$ ),
.21      mk-PartialFnTypeR( $at, rt$ ), mk-TotalFnTypeR( $at, rt$ )  $\rightarrow$ 
.22        IsFiniteTypeR( $\pi$ )( $at$ )( $env$ )  $\wedge$  IsFiniteTypeR( $\pi$ )( $rt$ )( $env$ ),
.23      mk-OpTypeR( $stt, at, rt$ )  $\rightarrow$ 
.24        IsFiniteTypeR( $\pi$ )( $stt$ )( $env$ )  $\wedge$ 
.25        IsFiniteTypeR( $\pi$ )( $at$ )( $env$ )  $\wedge$  IsFiniteTypeR( $\pi$ )( $rt$ )( $env$ ),
.26      mk-RetTypeR( $et$ ), mk-ExitTypeR( $et$ )  $\rightarrow$  IsFiniteTypeR( $\pi$ )( $et$ )( $env$ ),
.27      mk-QuoteTypeR(-)  $\rightarrow$  true,
.28      (BOOLEAN)  $\rightarrow$  true,
.29      others  $\rightarrow$  false
.30    end;

```

93.0 $IsUnionComp : TypeR \times TypeR \xrightarrow{t} Env \rightarrow \mathbb{B}$

```

.1  IsUnionComp( $t_1, t_2$ )( $env$ )  $\triangleq$ 
.2  let  $check = \lambda t_1 : TypeR, t_2 : TypeR .$ 
.3    IsUnionComp( $t_1, t_2$ )( $env$ ) in
.4  cases  $mk-(t_1, t_2)$  :
.5    mk-(mk-InvTypeR( $et_1, i$ ), mk-InvTypeR( $et_2, i$ ))  $\rightarrow$  check( $et_1; et_2$ ),
.6    mk-(mk-CompositeTypeR( $n, fs_1$ ), mk-CompositeTypeR( $n, fs_2$ ))  $\rightarrow$ 
.7      len  $fs_1 = len fs_2 \wedge$ 
.8       $\forall i \in \text{inds } fs_1 \cdot fs_1(i).id = fs_2(i).id \wedge check(fs_1(i).type, fs_2(i).type)$ ,
.9      mk-(-, mk-UnionTypeR( $ts_2$ ))  $\rightarrow \exists et_2 \in ts_2 \cdot check(t_1, et_2)$ ,
.10     mk-(mk-ProductTypeR( $ts_1$ ), mk-ProductTypeR( $ts_2$ ))  $\rightarrow$ 
.11       len  $ts_1 = len ts_2 \wedge \forall i \in \text{inds } ts_1 \cdot check(ts_1(i), ts_2(i))$ ,
.12       mk-(-, mk-Seq0TypeR( $et_2$ ))  $\rightarrow$ 
.13         check( $t_1, mk-UnionTypeR(\{\text{EMPTYSEQ}, mk-Seq1TypeR(et_2)\})$ )),
.14       mk-(mk-RetTypeR( $et_1$ ), mk-RetTypeR( $et_2$ ))  $\rightarrow$  check( $et_1, et_2$ ),
.15       mk-(mk-ExitTypeR( $et_1$ ), mk-ExitTypeR( $et_2$ ))  $\rightarrow$  check( $et_1, et_2$ ),
.16       others  $\rightarrow t_1 = t_2$ 
.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 no unions to split, t_2 is its own union component.

94.0 $\text{IsUnionComponent} : \text{TypeR} \times \text{TypeR} \xrightarrow{t} \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsUnionComponent}(t_1, t_2)(\text{env}) \triangleq$
 .2 $\text{IsUnionComp}(\text{TypeSubst}(\text{env}.typemap)(t_1), t_2)(\text{env})$

Like the above predicate except that all type names are expanded once before considering the union components.

95.0 $\text{DistributeTypeR} : \text{TypeR} \xrightarrow{t} \text{Env} \rightarrow \text{TypeR}$
 .1 $\text{DistributeTypeR}(t)(\text{env}) \triangleq$
 .2 if $\exists t_1, t_2 : \text{TypeR}$.
 .3 $t_1 \neq t_2 \wedge \text{IsUnionComponent}(t_1, t)(\text{env}) \wedge \text{IsUnionComponent}(t_2, t)(\text{env})$
 .4 then let *summands* : TypeR -set be st
 .5 $\forall s : \text{TypeR} : s \in \text{summands} \Leftrightarrow \text{IsUnionComponent}(s, t)(\text{env})$ in
 .6 $\text{mk-UnionTypeR}(\text{summands})$
 .7 else t

This function “normalises” a type t by forming a union type of all the union components of t .

96.0 $\text{Shape} : \text{TypeR} \xrightarrow{t} \text{TypeR}$
 .1 $\text{Shape}(t) \triangleq$
 .2 cases t :
 .3 $\text{mk-InvTypeR}(\text{et}, -) \rightarrow \text{et}$,
 .4 others $\rightarrow t$
 .5 end

10.5 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 $\text{LambdaR} :: \text{id} : \text{Id}$
 .1 $\text{type} : \text{TypeR}$
 .2 $\text{body} : \text{Expr} \mid \text{LambdaR}$
 .3 $\text{fnpre} : [\text{Expr}]$
 .4 $\text{total} : \mathbb{B}$

Chapter 11

The Static Semantics

In this chapter 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.

11.1 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 $wf\text{-}Spec$ 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 $wf\text{-}Spec$ 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-Spec :  $\Pi \xrightarrow{t} DefinitionBlock^*$  →  $\mathbb{B}$ 
.1     wf-Spec( $\pi$ )( $dbs$ )  $\triangleq$ 
.2       let  $typedefs = \text{conc } [dbs(i).typedefs \mid i \in \text{inds } dbs \cdot \text{is-TypeDefinitions}(dbs(i))]$ ,
.3          $statedefs = [dbs(i) \mid i \in \text{inds } dbs \cdot \text{is-StateDef}(dbs(i))]$ ,
.4          $valdefs = \text{conc } [dbs(i).valdefs \mid i \in \text{inds } dbs \cdot \text{is-ValueDefinitions}(dbs(i))]$ ,
.5          $fndefs = \text{conc } [dbs(i).funcdefs \mid i \in \text{inds } dbs \cdot \text{is-FunctionDefinitions}(dbs(i))]$ ,
.6          $opdefs = \text{conc } [dbs(i).opdefs \mid i \in \text{inds } dbs \cdot \text{is-OperationDefinitions}(dbs(i))]$ ,
.7          $fnppdefs = ExtractFnPrePostDefs(fndefs)$ ,
.8          $used-ids = UsedIds(typedefs, statedefs, valdefs, fndefs, opdefs)$ ,
.9          $oppdefs = ExtractOpPrePostDefs(opdefs, statedefs, used-ids)$ ,
.10         $invdefs = ExtractInvDefs(typedefs \curvearrowright statedefs)$ ,
.11         $initdefs = ExtractInitDefs(statedefs)$ ,
.12         $valfnopdefs = valdefs \curvearrowright fndefs \curvearrowright opdefs \curvearrowright$ 
.13           $fnppdefs \curvearrowright oppdefs \curvearrowright invdefs \curvearrowright initdefs$  in
.14       $\exists env : Env, venv : VisibleEnv .$ 
.15       $wf-TypeStateDefsInInitialEnv(\pi)(typedefs, statedefs)(env) \wedge$ 
.16       $wf-ValFnOpDefs(\pi)(valfnopdefs)(venv)(env) \wedge$ 
.17       $NoNameClash(typedefs \curvearrowright valfnopdefs) \wedge$ 
.18       $NoUnsatisfiableTypes(\pi)(valfnopdefs \curvearrowright typedefs)(env) \wedge$ 
.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 $ValFnDef = ValueDef \mid FunctionDef$;
- 105.0 $TypeDefs = TypeDef^*$

11.1.1 Auxiliary Well-formedness Requirements

106.0 $NoNameClash : ValFnOpTpDefs \xrightarrow{t} \mathbb{B}$

- .1 $NoNameClash(ds) \triangleq$
- .2 $\forall i, j \in \text{inds } ds .$
- .3 $i \neq j \Rightarrow$
- .4 $\text{LefthandSide}(ds(i)) \neq \text{LefthandSide}(ds(j)) \wedge$
- .5 $(\text{DefNamesInDef}(ds(i)) \cap \text{DefNamesInDef}(ds(j))) \neq \{\}$
- .6 $\Rightarrow \text{is-ValueDef}(ds(i)) \wedge \text{is-ValueDef}(ds(j)))$

Note that, according to the Dynamic Semantics, identical names are allowed in the patterns of two different value definitions (106.5.-6).

107.0 $LefthandSide : ValFnOpTpDef \xrightarrow{t} \text{Pattern} \mid Id$

- .1 $LefthandSide(d) \triangleq$
- .2 $\text{if is-ValueDef}(d) \text{ then } d.\text{pat} \text{ else } d.id;$

108.0 $NoUnsatisfiableTypes : \Pi \xrightarrow{t} ValFnOpTpDefs \rightarrow Env \rightarrow \mathbb{B}$

- .1 $NoUnsatisfiableTypes(\pi)(ds)(env) \triangleq$
- .2 $\text{is-POS}(\pi) \vee$
- .3 $\neg \exists fd : ValFnDef, td : TypeDef .$
- .4 $\{mk(fd, td), mk(td, fd)\} \subseteq ValFnOpTpTransDefDep(ds)$

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 $NoLooseInvariants : \Pi \xrightarrow{t} FunctionDef^* \times ValFnOpDefs \rightarrow Env \rightarrow \mathbb{B}$

- .1 $NoLooseInvariants(\pi)(invfns, ds)(env) \triangleq$
- .2 $\text{is-POS}(\pi) \vee$
- .3 $\forall invfn \in \text{elems } invfns .$
- .4 $NoLoosenessInDep ValFnDefs(invfn)(ds)(env)$

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 $NoLoosenessInDep ValFnDefs : ValFnDef \xrightarrow{t} ValFnOpDefs \rightarrow Env \rightarrow \mathbb{B}$

- .1 $NoLoosenessInDep ValFnDefs(d_1)(ds)(env) \triangleq$
- .2 $NoLoosenessInValFnDef(d_1) \wedge$
- .3 $\forall d_2 \in \text{elems } ds .$
- .4 $mk-(d_1, d_2) \in ValFnOpTpTransDefDep(ds) \wedge$
- .5 $(\text{is-ValueDef}(d_2) \vee \text{IsFunctionDef}(d_2)) \Rightarrow$
- .6 $NoLoosenessInValFnDef(d_2);$

111.0 $NoLoosenessInValFnDef : ValFnDef \xrightarrow{t} \mathbb{B}$

- .1 $NoLoosenessInValFnDef(d) \triangleq$
- .2 $(\forall s \in \text{SynComps}[Expr](d) \cdot NoLoosenessInExpr(s)) \wedge$
- .3 $\forall s \in \text{SynComps}[Pattern](d) \cdot NoLoosenessInPat(s) \wedge$
- .4 $\forall s \in \text{SynComps}[ExplFnDef](d) .$
- .5 $\neg \text{HasNonTrivialPreCond}(mk-POS(\{\}))(s) \wedge$
- .6 $\text{SynComps}[ImplFnDef](d) = \{\};$

- 112.0 $NoLoosenessInExpr : Expr \xrightarrow{t} \mathbb{B}$
- .1 $NoLoosenessInExpr(e) \triangleq$
 - .2 $\neg is-LetBeSTExpr(e);$
- 113.0 $NoLoosenessInPat : Pattern \xrightarrow{t} \mathbb{B}$
- .1 $NoLoosenessInPat(p) \triangleq$
 - .2 $\neg is-SetUnionPattern(p) \wedge \neg is-SetEnumPattern(p) \wedge \neg is-SeqConcPattern(p)$

11.2 Definitions

11.2.1 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 $wf-TypeStateDefsInInitialEnv : \Pi \xrightarrow{t} TypeDefs \times StateDef^* \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf-TypeStateDefsInInitialEnv(\pi)(tds, sds)(env) \triangleq$
 - .2 $\text{len } sds \leq 1 \wedge$
 - .3 $\text{dom } ExtractTypeMap(tds) \cap \text{dom } StateTypeMap(sds) = \{\} \wedge$
 - .4 $wf-TypeMap(\pi)(env) \wedge$
 - .5 $env.typemap = ExtractTypeMap(tds) \sqcup StateTypeMap(sds) \wedge$
 - .6 $env.visibleenv = \{n \mapsto mk-DomR(LOOKINTYPEMAP) \mid n : Name \cdot n \in \text{dom } env.typemap\} \wedge$
 - .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 $\text{if } sds = [] \text{ then nil else } mk-Name((\text{hd } sds).stid);$
- 116.0 $StateTypeMap : StateDef^* \xrightarrow{t} TypeMap$
- .1 $StateTypeMap(sds) \triangleq$
 - .2 $\text{if } sds = [] \text{ then } \{\mapsto\}$
 - .3 $\text{else let } mk-StateDef(stid, fields, stinv, -) = \text{hd } sds \text{ in}$
 - .4 $ExtractTypeMap([mk-TaggedTypeDef(stid, fields, stinv)]);$
- 117.0 $SatisfiableStateInitialisation : \Pi \xrightarrow{t} StateDef^* \rightarrow \mathbb{B}$
- .1 $SatisfiableStateInitialisation(\pi)(sds) \triangleq$
 - .2 $is-POS(\pi) \vee sds = []$

Types and Type maps

In this section, 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 $wf\text{-}TypeMap : \Pi \xrightarrow{t} Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}TypeMap(\pi)(env) \triangleq$
 - .2 $\forall t \in \text{rng } env.\text{typemap} .$
 - .3 $wf\text{-}TypeR(\pi)(t)(env);$
- 119.0 $wf\text{-}Type : \Pi \xrightarrow{t} Type \mid OpType \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}Type(\pi)(type)(typerep)(env) \triangleq$
 - .2 $typerep = ExtractTypeR(type)(env) \wedge wf\text{-}TypeR(\pi)(typerep)(env);$
- 120.0 $wf\text{-}AssertedType : \Pi \xrightarrow{t} [Type] \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}AssertedType(\pi)(ot)(t)(env) \triangleq$
 - .2 $\text{if } ot = \text{nil}$
 - .3 $\text{then } t = \text{ANY}$
 - .4 $\text{else } wf\text{-}Type(\pi)(ot)(t)(env);$

```

121.0   wf-TypeR :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-TypeR( $\pi$ )(type)(env)  $\triangleq$ 
.2       cases type :
.3         mk-InvTypeR(t, -)  $\rightarrow$ 
.4           wf-TypeR( $\pi$ )(t)(env),
.5         mk-CompositeTypeR(id, fs)  $\rightarrow$ 
.6           ( $\exists! t : CompositeTypeR$  IsTypeRwithTagInEnv(t, id)(env)  $\wedge$ 
.7             ( $\forall m k$ -FieldR(-, t)  $\in$  elems fs  $\cdot$  wf-TypeR( $\pi$ )(t)(env))  $\wedge$ 
.8                $\forall i_1, i_2 \in$  inds fs .
.9                  $i_1 \neq i_2 \wedge fs(i_1).id \neq nil \Rightarrow fs(i_1).id \neq fs(i_2).id,$ 
.10            mk-UnionTypeR(ts)  $\rightarrow$ 
.11               $\forall t \in ts \cdot wf-TypeR(\pi)(t)(env),$ 
.12            mk-ProductTypeR(ts)  $\rightarrow$ 
.13               $\forall t \in \text{elems } ts \cdot wf-TypeR(\pi)(t)(env),$ 
.14            mk-SetTypeR(t)  $\rightarrow$ 
.15              IsFlatTypeR( $\pi$ )(t)(env)  $\wedge$  wf-TypeR( $\pi$ )(t)(env),
.16            mk-Seq0TypeR(t)  $\rightarrow$ 
.17              wf-TypeR( $\pi$ )(t)(env),
.18            mk-Seq1TypeR(t)  $\rightarrow$ 
.19              wf-TypeR( $\pi$ )(t)(env),
.20            mk-GeneralMapTypeR(dt, rt)  $\rightarrow$ 
.21              IsFlatTypeR( $\pi$ )(dt)(env)  $\wedge$ 
.22                wf-TypeR( $\pi$ )(dt)(env)  $\wedge$  wf-TypeR( $\pi$ )(rt)(env),
.23            mk-InjectiveMapTypeR(dt, rt)  $\rightarrow$ 
.24              IsFlatTypeR( $\pi$ )(dt)(env)  $\wedge$ 
.25                wf-TypeR( $\pi$ )(dt)(env)  $\wedge$  wf-TypeR( $\pi$ )(rt)(env),
.26            mk-PartialFnTypeR(dt, rt)  $\rightarrow$ 
.27              DoesNotBuildOn( $\pi$ )(type, type)(env)  $\wedge$ 
.28                wf-TypeR( $\pi$ )(dt)(env)  $\wedge$  wf-TypeR( $\pi$ )(rt)(env),
.29            mk-TotalFnTypeR(dt, rt)  $\rightarrow$ 
.30              wf-TypeR( $\pi$ )(mk-PartialFnTypeR(dt, rt))(env),
.31            mk-OpTypeR(-, dt, rt)  $\rightarrow$ 
.32              IsFlatTypeR( $\pi$ )(dt)(env)  $\wedge$  IsFlatTypeR( $\pi$ )(rt)(env)  $\wedge$ 
.33                wf-TypeR( $\pi$ )(dt)(env)  $\wedge$  wf-TypeR( $\pi$ )(rt)(env),
.34            mk-RetTypeR(t)  $\rightarrow$  wf-TypeR( $\pi$ )(t)(env),
.35            mk-ExitTypeR(t)  $\rightarrow$  wf-TypeR( $\pi$ )(t)(env),
.36            mk-Name(-)  $\rightarrow$ 
.37              IsVisibleTypeName(type)(env),
.38            mk-TypeVarId(-)  $\rightarrow$ 
.39              IsVisibleTypeVar(type)(env),
.40            others  $\rightarrow$  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 :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     IsFlatTypeR( $\pi$ )(t)(env)  $\triangleq$ 
.2        $\forall ft : FunctionTypeR \cdot ft \neq t \wedge DoesNotBuildOn(\pi)(t, ft)(env);$ 

```

- 123.0 $BuildsOn : \Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $BuildsOn(\pi)(t_1, t_2)(env) \triangleq$
 .2 $mk-(t_1, t_2) \in TransTypeRDep(\pi)(t_1)(env);$
- 124.0 $DoesNotBuildOn : \Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $DoesNotBuildOn(\pi)(t_1, t_2)(env) \triangleq$
 .2 $\neg BuildsOn(NegWfClass(\pi))(t_1, t_2)(env);$
- 125.0 $IsTypeRwithTagInEnv : TypeR \times Id \xrightarrow{t} Env \rightarrow \mathbb{B}$
 .1 $IsTypeRwithTagInEnv(t, id)(env) \triangleq$
 .2 $IsTagOfType(id, t)(\text{rng } env.\text{typemap});$
- 126.0 $IsTagOfType : Id \times TypeR \xrightarrow{t} TypeR\text{-set} \rightarrow \mathbb{B}$
 .1 $IsTagOfType(id, t)(ts) \triangleq$
 .2 $is\text{-CompositeTypeR}(t) \wedge$
 .3 $\exists t_1 \in ts .$
 .4 $IsTypeRSubExpr(t, t_1) \wedge t.id = id;$
- 127.0 $IsTypeRSubExpr : TypeR \times TypeR \xrightarrow{t} \mathbb{B}$
 .1 $IsTypeRSubExpr(se, e) \triangleq$
 .2 $mk-(se, e) \in TransReflSynSubComp[TypeR](e)$

Extraction of Type Representations

- 128.0 $ExtractTypeMap : TypeDef^* \xrightarrow{t} TypeMap$
 .1 $ExtractTypeMap(tds) \triangleq$
 .2 $\text{if } tds = []$
 .3 $\text{then } \{\mapsto\}$
 .4 $\text{else let } tdef = \text{hd } tds,$
 .5 $\quad name = mk\text{-Name}(tdef.id),$
 .6 $\quad tinv = tdef.typeinv,$
 .7 $\quad tp = \text{if } is\text{-TaggedTypeDef}(tdef)$
 .8 $\quad \quad \text{then } mk\text{-CompositeType}(tdef.id, tdef.fields) \text{ else } tdef.shape,$
 .9 $\quad tpr = ExtractTypeR(tp)(EmptyEnv) \text{ in }$
 .10 $\quad \{name \mapsto \text{if } tinv = \text{nil} \text{ then } tpr \text{ else } mk\text{-InvTypeR}(tpr, tinv)\} \dagger$
 .11 $\quad ExtractTypeMap(\text{tl } tds);$

```

129.0    $\text{ExtractTypeR} : \text{Type} \mid \text{OpType} \xrightarrow{t} \text{Env} \rightarrow \text{TypeR}$ 
.1      $\text{ExtractTypeR}(tp)(env) \triangleq$ 
.2       cases tp :
.3          $\text{mk-BracketedType}(t) \rightarrow \text{ExtractTypeR}(t)(env),$ 
.4          $\text{mk-QuoteType}(lit) \rightarrow \text{mk-QuoteTypeR}(lit),$ 
.5          $\text{mk-CompositeType}(id, fs) \rightarrow \text{mk-CompositeTypeR}(id, \text{ExtractFieldRs}(fs)(env)),$ 
.6          $\text{mk-UnionType}(ts) \rightarrow \text{mk-UnionTypeR}(\text{elems ExtractTypeRs}(ts)(env)),$ 
.7          $\text{mk-ProductType}(ts) \rightarrow \text{mk-ProductTypeR}(\text{ExtractTypeRs}(ts)(env)),$ 
.8          $\text{mk-OptionalType}(t) \rightarrow \text{mk-UnionTypeR}(\{\text{ExtractTypeR}(t)(env), \text{UNITTYPE}\}),$ 
.9          $\text{mk-SetType}(t) \rightarrow \text{mk-SetTypeR}(\text{ExtractTypeR}(t)(env)),$ 
.10         $\text{mk-Seq0Type}(t) \rightarrow \text{mk-Seq0TypeR}(\text{ExtractTypeR}(t)(env)),$ 
.11         $\text{mk-Seq1Type}(t) \rightarrow \text{mk-Seq1TypeR}(\text{ExtractTypeR}(t)(env)),$ 
.12         $\text{mk-GeneralMapType}(dt, rt) \rightarrow$ 
.13           $\text{mk-GeneralMapTypeR}(\text{ExtractTypeR}(dt)(env), \text{ExtractTypeR}(rt)(env)),$ 
.14         $\text{mk-InjectiveMapType}(dt, rt) \rightarrow$ 
.15           $\text{mk-InjectiveMapTypeR}(\text{ExtractTypeR}(dt)(env), \text{ExtractTypeR}(rt)(env)),$ 
.16         $\text{mk-PartialFnType}(dt, rt) \rightarrow$ 
.17           $\text{mk-PartialFnTypeR}(\text{ExtractTypeR}(dt)(env), \text{ExtractTypeR}(rt)(env)),$ 
.18         $\text{mk-TotalFnType}(dt, rt) \rightarrow$ 
.19           $\text{mk-TotalFnTypeR}(\text{ExtractTypeR}(dt)(env), \text{ExtractTypeR}(rt)(env)),$ 
.20         $\text{mk-OpType}(dt, rt) \rightarrow$ 
.21           $\text{mk-OpTypeR}(\text{env.statename}, \text{ExtractTypeR}(dt)(env), \text{ExtractTypeR}(rt)(env)),$ 
.22        others  $\rightarrow tp$ 
.23      end;
130.0    $\text{ExtractTypeRs} : \text{Type}^* \xrightarrow{t} \text{Env} \rightarrow \text{TypeR}^*$ 
.1      $\text{ExtractTypeRs}(ts)(env) \triangleq$ 
.2        $[\text{ExtractTypeR}(ts(i))(env) \mid i \in \text{inds } ts];$ 
131.0    $\text{ExtractFieldRs} : \text{Field}^* \xrightarrow{t} \text{Env} \rightarrow \text{FieldR}^*$ 
.1      $\text{ExtractFieldRs}(fs)(env) \triangleq$ 
.2        $[\text{mk-FieldR}(fs(i).sel, \text{ExtractTypeR}(fs(i).type)(env)) \mid i \in \text{inds } fs]$ 

```

11.2.2 Value, Function, and Operation Definitions

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.

132.0 $wf\text{-}ValFnOpDefs : \Pi \xrightarrow{t} ValFnOpDefs \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}ValFnOpDefs(\pi)(ds)(denv)(env) \triangleq$
 .2 $\text{let } defgrps : ValFnOpDef\text{-set}^* = OrderedDefGroups(ds, env) \text{ in}$
 .3 $wf\text{-OrderedDefGroups}(\pi)(defgrps)(denv)(env)$

Ordered Definition Groups and Definition Sequences

133.0 $NonTpDef = ValFnOpDef \mid DefBind;$
 134.0 $NonTpDefs = NonTpDef^*$

135.0 $wf\text{-OrderedDefGroups} : \Pi \xrightarrow{t} ValFnOpDef\text{-set}^* \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-OrderedDefGroups}(\pi)(dgs)(denv)(env) \triangleq$
 .2 $\text{cases } dgs :$
 .3 $([]) \rightarrow denv = \{\mapsto\},$
 .4 $\text{preceding} \curvearrowright [last] \rightarrow$
 .5 $\exists denv_1, denv_2 : VisibleEnv .$
 .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 $wf\text{-DefGroup} : \Pi \xrightarrow{t} ValFnOpDef\text{-set} \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-DefGroup}(\pi)(dg)(denv)(env) \triangleq$
 .2 $\exists dl : ValFnOpDefs, deps, essential, order : DefDepRel .$
 .3 $\exists defnames, useful : Name\text{-set}, saenv : VisibleEnv .$
 .4 $IsOrderingOf(dg, dl)(order) \wedge$
 .5 $deps = ValFnOpTpDefDep(dl) \wedge$
 .6 $IsSafeAssertedVisEnv(\pi)(dl)(saenv)(env) \wedge$
 .7 $defnames = DefNamesInDefs(dl) \wedge$
 .8 $useful = \{n \mid n \in \text{dom } saenv .$
 .9 $\exists t : TypeR . IsBoundToValTypeR(\pi)(n)(t)(saenv) \wedge t \neq \text{ANY}\} \wedge$
 .10 $essential =$
 .11 $\{mk-(d_1, d_2) \mid mk-(d_1, d_2) \in deps \wedge \neg DefNamesInDef(d_2) \subseteq useful\} \wedge$
 .12 $order = FindDepOrdering(essential) \wedge$
 .13 $wf\text{-OrderedDefs}(\pi)(dl, true)(denv)(UpdateEnv(saenv)(RemoveFromEnv(defnames)(env)))$

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.

137.0 $wf\text{-}OrderedDefs : \Pi \xrightarrow{t} NonTpDefs \times \mathbb{B} \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}OrderedDefs(\pi)(ds, envmerge)(denv)(env) \triangleq$
- .2 cases $ds :$
- .3 $([] \rightarrow denv = \{\rightarrow\},$
- .4 $preceding \curvearrowright [last] \rightarrow$
- .5 $\exists denv_1, denv_2 : VisibleEnv .$
- .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 $IsVisEnvMerge : \Pi \rightarrow VisibleEnv \times VisibleEnv \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$

- .1 $IsVisEnvMerge(\pi)(venv_1, venv_2)(venv)(env) \triangleq$
- .2 $\exists valenv, valenv_1, valenv_2, rest_1, rest_2 : VisibleEnv .$
- .3 $venv_1 = valenv_1 \sqcup rest_1 \wedge venv_2 = valenv_2 \sqcup rest_2 \wedge$
- .4 $\forall ssr \in rng valenv_1 \cup rng valenv_2 . is\text{-}ValR(ssr) \wedge$
- .5 $IsVenvMerge(\pi)(valenv_1, valenv_2)(valenv)(env) \wedge$
- .6 $venv = valenv \sqcup rest_1 \sqcup rest_2$

Definitions

139.0 $wf\text{-}NonTpDef : \Pi \xrightarrow{t} NonTpDef \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$

- .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 $is\text{-}DefBind(d) \wedge wf\text{-}DefBind(\pi)(d)(denv);$

140.0 $wf\text{-}ValueDef : \Pi \xrightarrow{t} ValueDef \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}ValueDef(\pi)(mk\text{-}ValueDef(pat, tp, val))(denv)(env) \triangleq$
- .2 let $pb = \text{if } tp = \text{nil} \text{ then } pat \text{ else } mk\text{-}TypeBind(pat, tp) \text{ in }$
- .3 $\exists t : TypeR .$
- .4 $wf\text{-}Expr(\pi)(val)(t)(env) \wedge$
- .5 $wf\text{-}PatternBind(\pi)(pb)(mk\text{-}TVE(t, denv))(env);$

- 141.0 $wf\text{-}ExplFnDef : \Pi \xrightarrow{t} ExplFnDef \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}ExplFnDef(\pi)(d)(denv)(env) \triangleq$
 - .2 $d.id = d.idrepeated \wedge$
 - .3 $\exists n : Name, ssr : StaticSemanticRep, tvs : TypeVar^*, at : FunctionTypeR .$
 - .4 $IsAssertedFromDef(\pi)(d)(\{n \mapsto ssr\})(env) \wedge$
 - .5 $IsFnRep(tvs, at)(ssr) \wedge$
 - .6 $\exists lam : LambdaR, vt : FunctionTypeR, ssr_2 : StaticSemanticRep .$
 - .7 $IsCurriedFn(lam)(d.parms, at, d.fnpre, d.body)(env) \wedge$
 - .8 $wf\text{-}LambdaR(\pi)(lam)(vt)(UpdateEnv(elems tvs))(env) \wedge$
 - .9 $IsSubtype(\pi)(vt, at)(env) \wedge$
 - .10 $IsFnRep(tvs, vt)(ssr_2) \wedge$
 - .11 $denv = \{n \mapsto ssr_2\};$
- 142.0 $IsFnRep : TypeVar^* \times FunctionTypeR \xrightarrow{t} StaticSemanticRep \rightarrow \mathbb{B}$
- .1 $IsFnRep(tvs, t)(ssr) \triangleq$
 - .2 $\exists \pi : \Pi .$
 - .3 $tvs = [] \wedge ssr = mk\text{-}ValR(t, \pi) \vee$
 - .4 $tvs \neq [] \wedge ssr = mk\text{-}PolyValR(tvs, mk\text{-}ValR(t, \pi));$
- 143.0 $wf\text{-}ImplFnDef : \Pi \xrightarrow{t} ImplFnDef \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}ImplFnDef(\pi)(ifd)(denv)(env) \triangleq$
 - .2 $\text{let } mk\text{-}ImplFnDef(id, tpparms, partps, residtype, fnpre, fnpost) = ifd \text{ in}$
 - .3 $\exists n : Name, ft : TotalFnTypeR .$
 - .4 $\neg IsEmptyType(NegWfClass(\pi))(ft)(env) \wedge$
 - .5 $denv = \{n \mapsto mk\text{-}ValR(ft, \pi)\} \wedge IsAssertedFromDef(\pi)(ifd)(denv)(env) \wedge$
 - .6 $\exists x : Name, fnpre', fnpost' : Expr .$
 - .7 $x.name \neq residtype.id \wedge$
 - .8 $IsLetExpandedParTypes(fnpre')(partps, x, fnpre)(env) \wedge$
 - .9 $IsLetExpandedParTypes(fnpost')(partps, x, fnpost)(env) \wedge$
 - .10 $\text{let } venv_{pre} = \{x \mapsto mk\text{-}ValR(ft.fndom, \pi)\},$
 - .11 $env_{pre} = UpdateEnv(venv_{pre})(env),$
 - .12 $venv_{post} = \{mk\text{-}Name(residtype.id) \mapsto mk\text{-}ValR(ft.fnrng, \pi)\},$
 - .13 $env_{post} = UpdateEnv(venv_{post})(env_{pre}) \text{ in }$
 - .14 $PreNeverTrue(\pi)(fnpre')(env_{pre}) \vee$
 - .15 $\text{cases } \pi :$
 - .16 $mk\text{-}DEF(-) \rightarrow \text{false},$
 - .17 $mk\text{-}POS(-) \rightarrow PreAlwaysTrue(NegWfClass(\pi))(fnpre')(env_{pre}) \Rightarrow$
 - .18 $\text{let } locvars = \text{dom } venv_{pre} \cup \text{dom } venv_{post},$
 - .19 $wfe = (InScope[Expr](\pi)(locvars)) \circ$
 - .20 $(InType[Expr](\pi)(wf\text{-}Expr(\pi))) \text{ in }$
 - .21 $wfe(fnpost')(BOOLEAN)(env_{post})$
 - .22 end;
- 144.0 $wf\text{-}DefBind : \Pi \xrightarrow{t} DefBind \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}DefBind(\pi)(mk\text{-}DefBind(pb, expr))(denv)(env) \triangleq$
 - .2 $\exists t : TypeR .$
 - .3 $wf\text{-}Expr(\pi)(expr)(t)(env) \wedge$
 - .4 $wf\text{-}PatternBind(\pi)(pb)(mk\text{-}TVE(t, denv))(env)$

Well-formedness of Operation Definitions

```

145.0  wf-ExplOprtDef :  $\Pi \xrightarrow{t} ExplOprtDef \rightarrow VisibleEnv \rightarrow Env \rightarrow \mathbb{B}$ 
.1    wf-ExplOprtDef ( $\pi$ )( $eod$ )( $denv$ )( $env$ )  $\triangleq$ 
.2      let  $mk\text{-ExplOprtDef}(id, optype, idrepeated, params, body, oppre) = eod$ ,
.3           $env' = UpdateEnv(StateVisEnv(env))(env)$ ,
.4           $p = Pats2Pat(params)$  in
.5           $id = idrepeated \wedge$ 
.6           $\exists n : Name, ot : OpTypeR .$ 
.7               $denv = \{n \mapsto mk\text{-ValR}(ot, \pi)\} \wedge IsAssertedFromDef(\pi)(eod)(denv)(env') \wedge$ 
.8               $\exists x : Name, oppre' : Expr, body' : Stmt .$ 
.9                   $IsLetExpansion(oppree')(p, nil, x, oppre)(env') \wedge$ 
.10                  $IsLetExpansion(body')(p, nil, x, body)(env') \wedge$ 
.11                 let  $venv = \{x \mapsto mk\text{-ValR}(ot.opdom, \pi)\}$ ,
.12                      $newenv = UpdateEnv(venv)(env')$  in
.13                      $PreNeverTrue(\pi)(oppree')(newenv) \vee$ 
.14                     cases  $\pi$  :
.15                          $mk\text{-DEF}(cs) \rightarrow$ 
.16                              $wf\text{-Stmt}(mk\text{-DEF}(cs))(body)(ot.oprng)(newenv) \vee$ 
.17                              $\neg \exists body_t : StmtTypeR .$ 
.18                             let  $\pi_{POS} = mk\text{-POS}(cs)$ ,
.19                                  $wfs = InScope[Stmt](\pi_{POS})(\text{dom } venv)(wf\text{-Stmt}(\pi_{POS}))$  in
.20                                  $wfs(body')(body_t)(newenv)$ ,
.21                          $mk\text{-POS}(cs) \rightarrow$ 
.22                              $PreAlwaysTrue(NegWfClass(\pi))(oppree')(newenv) \Rightarrow$ 
.23                              $\neg \exists body_t : StmtTypeR .$ 
.24                              $wf\text{-Stmt}(mk\text{-DEF}(cs))(body')(body_t)(newenv) \wedge$ 
.25                              $IsDisjoint(mk\text{-DEF}(cs))(body_t, ot.oprng)(env')$ 
.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).

```

146.0  wf-ImplOprtDef :  $\Pi \xrightarrow{t} \text{ImplOprtDef} \rightarrow \text{VisibleEnv} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   wf-ImplOprtDef( $\pi$ )( $\text{iod}$ )( $\text{denv}$ )( $\text{env}$ )  $\triangleq$ 
.2     let  $\text{mk-ImplOprtDef}(id, \text{params}, \text{residtype}, \text{opext}, \text{oppre}, \text{oppot}, \text{excps}) = \text{iod}$ ,
.3        $\text{env}' = \text{UpdateEnv}(\text{StateVisEnv}(\text{env}))(\text{env})$ ,
.4        $\text{oppre}' = \text{PreOExceptions2Pre}(\text{oppre}, \text{excps})$ ,
.5        $\text{oppot}' = \text{PostOExceptions2Post}(\text{oppot}, \text{excps})$  in
.6      $\exists n : \text{Name}, ot : \text{OpTypeR} .$ 
.7        $\text{denv} = \{n \mapsto \text{mk-ValR}(ot, \pi)\} \wedge \text{IsAssertedFromDef}(\pi)(\text{iod})(\text{denv})(\text{env}') \wedge$ 
.8        $\text{wf-OExternals}(\pi)(\text{opext})(\text{env}') \wedge$ 
.9        $\exists x : \text{Name}, \text{oppre}'', \text{oppot}'' : \text{Expr} .$ 
.10       $(\text{residtype} \neq \text{nil} \Rightarrow x.\text{name} \neq \text{residtype}.id) \wedge$ 
.11       $\text{IsLetExpandedParTypes}(\text{oppre}'')(\text{params}, x, \text{oppre}')(env') \wedge$ 
.12       $\text{IsLetExpandedParTypes}(\text{oppot}'')(\text{params}, x, \text{oppot}')(env') \wedge$ 
.13      let  $\text{venv}_1 = \{x \mapsto \text{mk-ValR}(ot.\text{opdom}, \pi)\}$ ,
.14         $\text{venv}_2 = \text{if } \text{residtype} = \text{nil}$ 
.15        then  $\{\mapsto\}$ 
.16        else  $\{\text{mk-Name}(\text{residtype}.id) \mapsto \text{mk-ValR}(ot.\text{oprng}, \pi)\}$ ,
.17         $\text{env}_{\text{pre}} = \text{UpdateEnv}(\text{venv}_1)(\text{env}')$ ,
.18         $\text{venv}_3 = \{\text{mk-OldName}(nn.\text{name}) \mapsto \text{ssr} \mid$ 
.19           $nn \in \text{StateVarNames}(\text{env}'), \text{ssr} : \text{StaticSemanticRep} .$ 
.20           $\text{IsBoundTo}(nn)(\text{ssr})(\text{env}')\}$ ,
.21         $\text{env}_{\text{post}} = \text{UpdateEnv}(\text{venv}_3 \sqcup \text{venv}_2)(\text{env}_{\text{pre}})$  in
.22         $\text{PreNeverTrue}(\pi)(\text{oppre}'')(\text{env}_{\text{post}}) \vee$ 
.23        cases  $\pi :$ 
.24           $\text{mk-DEF}(\cdot) \rightarrow \text{false}$ ,
.25           $\text{mk-POS}(\cdot) \rightarrow$ 
.26             $\text{PreAlwaysTrue}(\text{NegWfClass}(\pi))(\text{oppre}')(env_{\text{pre}}) \Rightarrow$ 
.27            let  $\text{locvars} = \text{dom } \text{venv}_1 \cup \text{dom } \text{venv}_2 \cup \text{dom } \text{venv}_3$ ,
.28             $wfs = (\text{InScope}[\text{Expr}](\pi)(\text{locvars})) \circ (\text{InType}[\text{Expr}](\text{wf-Expr}(\pi)))$  in
.29             $wfs(\text{oppot}'')(\text{BOOLEAN})(\text{env}_{\text{post}})$ 
.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 :  $\Pi \xrightarrow{t} \text{OExternals} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   wf-OExternals( $\pi$ )( $\text{exts}$ )( $\text{env}$ )  $\triangleq$ 
.2      $\forall \text{varinf} \in \text{elems exts} \cdot \text{wf-VarInf}(\pi)(\text{varinf})(\text{env});$ 
148.0  wf-VarInf :  $\Pi \xrightarrow{t} \text{VarInf} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   wf-VarInf( $\pi$ )( $\text{mk-VarInf}(\text{mode}, \text{vars}, \text{type})$ )( $\text{env}$ )  $\triangleq$ 
.2      $\text{mode} = \text{READ} \Rightarrow$ 
.3      $\forall v \in \text{elems vars} \cdot v \in \text{dom env.visibleenv}$ 

```

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 induced dependencies: A group A depends on another B iff at least one definition in A depends on an entity defined in B .

- 149.0 $OrderedDefGroups(ds : ValFnOpDefs, env : Env) odg : ValFnOpDef\text{-set}^*$
 - .1 post let $deprel = ValFnOpTpTransDefDep(ds)$,
 $simrel = SimultaneousDefRel(\text{elems } ds, deprel)$,
 $groupset = Partition[ValFnOpDef](\text{elems } ds, simrel)$,
 $order = InducedPartitionOrder[ValFnOpDef](groupset, deprel)$ in
 $\text{len } odg = \text{card } groupset \wedge \text{elems } odg = groupset \wedge$
 $IsPartiallyOrdered[ValFnOpTpDef\text{-set}](odg)(order)$;
- 150.0 $SimultaneousDefRel : ValFnOpDef\text{-set} \times DefDepRel \xrightarrow{t} DefDepRel$
 - .1 $SimultaneousDefRel(dset, drel) \triangleq$
 $\{mk-(d, d) \mid d \in dset\} \cup$
 $\{mk-(d_1, d_2) \mid d_1, d_2 \in dset \cdot \{mk-(d_1, d_2), mk-(d_2, d_1)\} \subseteq drel\};$
- 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 $\forall s_1, s_2 \in ss \cdot$
 $mk-(s_1, s_2) \in srel \Leftrightarrow$
 $\exists e_1 \in s_1, e_2 \in s_2 \cdot mk-(e_1, e_2) \in rel$;
- 152.0 $Partition[@T](s : @T\text{-set}, rel : (@T \times @T)\text{-set}) ss : @T\text{-set-set}$
 - .1 post $\bigcup ss = s \wedge \{\} \not\subseteq ss \wedge$
 $\forall s_1, s_2 \in ss \cdot$
 $s_1 \cap s_2 = \{\} \vee s_1 = s_2 \wedge$
 $\forall e_1, e_2 \in s \cdot$
 $mk-(e_1, e_2) \in rel \Leftrightarrow$
 $\exists s \in ss \cdot \{e_1, e_2\} \subseteq s$

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 $\exists subrel : DefDepRel \cdot$
 $subrel \subseteq rel \wedge order = TransClosure[ValFnOpTpDef](subrel) \wedge$
 $IsAsymmetric[ValFnOpTpDef](order) \wedge$
 $\forall pair \in rel \cdot$
 $pair \not\subseteq subrel \Rightarrow$
 $\neg IsAsymmetric[ValFnOpTpDef](TransClosure[ValFnOpTpDef](rel \cup \{pair\}))$;
- 154.0 $IsAsymmetric[@T] : (@T \times @T)\text{-set} \xrightarrow{t} \mathbb{B}$
 - .1 $IsAsymmetric(rel) \triangleq$
 $\forall x, y : @T \cdot$
 $mk-(x, y) \in rel \Rightarrow mk-(y, x) \notin rel$;

- 155.0 $IsOrderingOf : ValFnOpDef\text{-set} \times ValFnOpDefs \xrightarrow{t} DefDepRel \rightarrow \mathbb{B}$
- .1 $IsOrderingOf(dg, dl)(order) \triangleq$
 - .2 $\text{elems } dl = dg \wedge \text{len } dl = \text{card } dg \wedge$
 - .3 $IsPartiallyOrdered[ValFnOpTpDef](dl)(order);$
- 156.0 $IsPartiallyOrdered[@T] : @T^* \xrightarrow{t} (@T \times @T)\text{-set} \rightarrow \mathbb{B}$
- .1 $IsPartiallyOrdered(s)(order) \triangleq$
 - .2 $\forall i_1, i_2 \in \text{inds } s .$
 - .3 $mk-(s(i_2), s(i_1)) \in order \Rightarrow i_1 \leq i_2$

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 $IsSafeAssertedVisEnv : \Pi \xrightarrow{t} ValFnOpDefs \rightarrow \text{VisibleEnv} \rightarrow Env \rightarrow \mathbb{B}$
- .1 $IsSafeAssertedVisEnv(\pi)(ds)(saenv)(env) \triangleq$
 - .2 $\text{if } (\forall d \in \text{elems } ds .$
 - .3 $\quad is\text{-ValueDef}(d) \vee is\text{-ExplFnDef}(d) \wedge d.tpparms = [] \vee$
 - .4 $\quad \forall d \in \text{elems } ds .$
 - .5 $\quad is\text{-ExplFnDef}(d) \wedge d.tpparms \neq [] \vee$
 - .6 $\quad is\text{-POS}(\pi)$
 - .7 $\quad \text{then } IsSafeAssertedFromDefs}(\pi)(ds)(saenv)(env)$
 - .8 $\quad \text{else } IsSafeAssertedFromDefs}(NegWfClass(\pi))(ds)(saenv)(env);$
- 158.0 $IsSafeAssertedFromDefs : \Pi \xrightarrow{t} ValFnOpDefs \rightarrow \text{VisibleEnv} \rightarrow Env \rightarrow \mathbb{B}$
- .1 $IsSafeAssertedFromDefs(\pi)(ds)(saenv)(env) \triangleq$
 - .2 $\text{if } ds = [] \text{ then } saenv = \{\mapsto\}$
 - .3 $\text{else } \exists saenv_1, saenv_2 : \text{VisibleEnv} .$
 - .4 $\quad IsSafeAssertedFromDef}(\pi)(hd \ ds)(saenv_1)(env) \wedge$
 - .5 $\quad IsSafeAssertedFromDefs}(\pi)(tl \ ds)(saenv_2)(env) \wedge$
 - .6 $\quad IsVisEnvMerge}(\pi)(saenv_1, saenv_2)(saenv)(env);$
- 159.0 $IsSafeAssertedFromDef : \Pi \xrightarrow{t} ValFnOpDef \rightarrow \text{VisibleEnv} \rightarrow Env \rightarrow \mathbb{B}$
- .1 $IsSafeAssertedFromDef(\pi)(d)(saenv)(env) \triangleq$
 - .2 $\exists aenv : \text{VisibleEnv} .$
 - .3 $\quad IsAssertedFromDef}(\pi)(d)(aenv)(env) \wedge$
 - .4 $\quad saenv = \{n \mapsto SafeEnvEntry}(\pi)(aenv(n))(is\text{-ValueDef}(d)) \mid n \in \text{dom } aenv\};$

160.0 $\text{SafeEnvEntry} : \Pi \xrightarrow{t} \text{StaticSemanticRep} \rightarrow \mathbb{B} \rightarrow \text{StaticSemanticRep}$
 .1 $\text{SafeEnvEntry}(\pi)(\text{ssr})(\text{origins-from-valuedef}) \triangleq$
 .2 if $\text{is-POS}(\pi)$ then ssr
 .3 else cases ssr :
 .4 $\text{mk-ValR}(t, -) \rightarrow$
 .5 if $\text{Is[FunctionTypeR, TypeR]}(t) \wedge \neg \text{origins-from-valuedef}$
 .6 then $\text{mk-ValR}(\text{MakePartialFnTypeR}(t), \pi)$
 .7 else $\text{mk-ValR}(t, \text{NegWfClass}(\pi))$,
 .8 $\text{mk-PolyValR}(\text{tppars}, \text{valr}) \rightarrow$
 .9 $\text{mk-PolyValR}(\text{tppars}, \text{SafeEnvEntry}(\pi)(\text{valr})(\text{origins-from-valuedef}))$,
 .10 others $\rightarrow \text{ssr}$
 .11 end;

 161.0 $\text{MakePartialFnTypeR} : \text{FunctionTypeR} \xrightarrow{t} \text{PartialFnTypeR}$
 .1 $\text{MakePartialFnTypeR}(\text{ft}) \triangleq$
 .2 $\text{mk-PartialFnTypeR}(\text{ft.fndom}, \text{ft.fnrng});$

 162.0 $\text{IsAssertedVisEnv} : \Pi \xrightarrow{t} \text{ValFnOpDefs} \rightarrow \text{VisibleEnv} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsAssertedVisEnv}(\pi)(\text{ds})(\text{denv})(\text{env}) \triangleq$
 .2 $\text{ds} = [] \wedge \text{denv} = \{\mapsto\} \vee$
 .3 $\exists \text{denv}_1, \text{denv}_2 : \text{VisibleEnv}.$
 .4 $\text{IsAssertedVisEnv}(\pi)(\text{tl ds})(\text{denv}_1)(\text{env}) \wedge$
 .5 $\text{IsAssertedFromDef}(\pi)(\text{hd ds})(\text{denv}_2)(\text{env}) \wedge$
 .6 $\text{IsVisEnvMerge}(\pi)(\text{denv}_1, \text{denv}_2)(\text{denv})(\text{env});$

163.0 $\text{IsAssertedFromDef} : \Pi \xrightarrow{t} \text{ValFnOpDef} \rightarrow \text{VisibleEnv} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsAssertedFromDef}(\pi)(d)(\text{denv})(\text{env}) \triangleq$
 .2 cases d :
 .3 $\text{mk-ValueDef}(\text{pat}, \text{otype}, -) \rightarrow$
 .4 $\exists t, et : \text{TypeR} .$
 .5 $\text{wf-AssertedType}(\pi)(\text{otype})(t)(\text{env}) \wedge$
 .6 $\text{IsNonEmptyEssEquivTypeR}(\pi)(et, t)(\text{env}) \wedge$
 .7 $\text{HasMatchingSubtype}(\pi)(\text{pat})(et)(\text{denv})(\text{env}),$
 .8 $\text{mk-ExplFnDef}(i, \text{otvs}, \text{type}, -, -, -, -) \rightarrow$
 .9 $\exists t, et : \text{FunctionTypeR}, vr : \text{ValR} .$
 .10 $\text{wf-Type}(\pi)(\text{type})(t)(\text{env}) \wedge$
 .11 $\text{IsNonEmptyEssEquivTypeR}(\pi)(et, t)(\text{env}) \wedge$
 .12 $\text{denv} = \text{MakeFnEntry}(\pi)(i, \text{HasNonTrivialPreCond}(\pi)(d), \text{otvs}, et),$
 .13 $\text{mk-ImplFnDef}(i, \text{otvs}, ps, \text{mk-IdType}(-, \text{rtype}), -, -) \rightarrow$
 .14 $\exists at, rt, ft, et : \text{TypeR}, vr : \text{ValR} .$
 .15 $\text{wf-ParameterTypes}(\pi)(ps)(at)(\text{env}) \wedge$
 .16 $\text{wf-Type}(\pi)(\text{rtype})(rt)(\text{env}) \wedge$
 .17 $ft = \text{mk-TotalFnTypeR}(at, rt) \wedge$
 .18 $\text{IsNonEmptyEssEquivTypeR}(\pi)(et, ft)(\text{env}) \wedge$
 .19 $\text{denv} = \text{MakeFnEntry}(\pi)(i, \text{HasNonTrivialPreCond}(\pi)(d), \text{otvs}, et),$
 .20 $\text{mk-ExplOprtDef}(i, \text{otype}, -, -, -, -) \rightarrow$
 .21 $\exists ot, et : \text{OpTypeR} .$
 .22 $\text{wf-Type}(\pi)(\text{otype})(ot)(\text{env}) \wedge$
 .23 $\text{IsNonEmptyEssEquivTypeR}(\pi)(et, ot)(\text{env}) \wedge$
 .24 $\text{denv} = \{\text{mk-Name}(i) \mapsto \text{mk-ValR}(et, \pi)\},$
 .25 $\text{mk-ImplOprtDef}(i, ps, ridtp, -, -, -, -) \rightarrow$
 .26 $\exists at, rt, ot, et : \text{TypeR} .$
 .27 $\text{wf-ParameterTypes}(\pi)(ps)(at)(\text{env}) \wedge$
 .28 $(\text{wf-Type}(\pi)(ridtp.type)(rt)(\text{env}) \vee ridtp = \text{nil} \wedge rt = \text{UNITTYPE}) \wedge$
 .29 $ot = \text{mk-OpTypeR}(\text{env}.statename, at, rt) \wedge$
 .30 $\text{IsNonEmptyEssEquivTypeR}(\pi)(et, ot)(\text{env}) \wedge$
 .31 $\text{denv} = \{\text{mk-Name}(i) \mapsto \text{mk-ValR}(et, \pi)\}$
 .32 end;

164.0 $\text{MakeFnEntry} : \Pi \xrightarrow{t} \text{Id} \times \mathbb{B} \times \text{TypeVar}^* \times \text{FunctionTypeR} \rightarrow \text{VisibleEnv}$
 .1 $\text{MakeFnEntry}(\pi)(i, \text{haspre}, \text{otvs}, t) \triangleq$
 .2 let $vr = \text{mk-ValR}(\text{if haspre then MakePartialFnTypeR}(t) \text{ else } t, \pi),$
 .3 $ssr = \text{if otvs} = [] \text{ then } vr \text{ else } \text{mk-PolyValR}(\text{otvs}, vr) \text{ in}$
 .4 $\{\text{mk-Name}(i) \mapsto ssr\};$

165.0 $\text{wf-ParameterTypes} : \Pi \xrightarrow{t} \text{ParameterTypes} \rightarrow \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{wf-ParameterTypes}(\pi)(ps)(t)(\text{env}) \triangleq$
 .2 let $atype = \text{ParameterTypes2Type}(ps)$ in
 .3 $\text{wf-Type}(\pi)(atype)(t)(\text{env});$

166.0 $\text{HasMatchingSubtype} : \Pi \xrightarrow{t} \text{Pattern} \rightarrow \text{TypeR} \rightarrow \text{VisibleEnv} \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{HasMatchingSubtype}(\pi)(\text{pat})(\text{tp})(\text{ve})(\text{env}) \triangleq$
 .2 $\exists stp : \text{TypeR} .$
 .3 $\text{IsSubtypeR}(\pi)(stp, tp)(\text{env}) \wedge$
 .4 $\text{wf-Pattern}(\pi)(\text{pat})(\text{mk-TVE}(stp, ve))(\text{env})$

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.

Pre-Conditions

- 167.0 $\text{HasNonTrivialPreCond} : \Pi \xrightarrow{t} \text{FunctionDef} \rightarrow \mathbb{B}$
- .1 $\text{HasNonTrivialPreCond}(\pi)(d) \triangleq$
 - .2 $\neg \text{is-DEF}(\pi) \wedge$
 - .3 cases $d :$
 - .4 $\text{mk-ExplFnDef}(-, -, -, -, -, \text{body}, \text{fnpre}), \text{mk-ImplFnDef}(-, -, -, -, \text{fnpre}, -) \rightarrow$
 - .5 $\text{fnpre} \neq \text{nil}$
 - .6 end;
- 168.0 $\text{PreNeverTrue} : \Pi \xrightarrow{t} [\text{Expr}] \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{PreNeverTrue}(\pi)(\text{opre})(\text{env}) \triangleq$
 - .2 $\text{opre} \neq \text{nil} \wedge$
 - .3 cases $\pi :$
 - .4 $\text{mk-POS}(-) \rightarrow \text{true},$
 - .5 $\text{mk-DEF}(\text{cs}) \rightarrow \neg \text{InType}[\text{Expr}](\text{mk-POS}(\text{cs}))(\text{wf-Expr}(\text{mk-POS}(\text{cs}))) (\text{opre})(\text{BOOLEAN})(\text{env})$
 - .6 end;
- 169.0 $\text{PreAlwaysTrue} : \Pi \xrightarrow{t} [\text{Expr}] \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{PreAlwaysTrue}(\pi)(\text{opre})(\text{env}) \triangleq$
 - .2 $\text{opre} = \text{nil} \vee \text{is-POS}(\pi)$

11.3 Expressions

In this section 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 ($X\text{Expr}$) has the form:

$$\begin{aligned} \text{wf-}X\text{Expr} &: \Pi \xrightarrow{t} \text{Expr} \rightarrow \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B} \\ \text{wf-}X\text{Expr}(\pi)(e)(t)(\text{env}) &\triangleq \\ &\text{body} \end{aligned}$$

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.

11.3.1 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:

$$170.0 \quad \text{ExprCP} = \text{TypeR}^* \rightarrow \text{TypeR} \rightarrow \text{Env} \rightarrow \mathbb{B}$$

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 :  $\Pi \xrightarrow{t} ExprCP \rightarrow Expr^* \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-SCompExpr( $\pi$ )( $CP$ )( $exprs$ )( $restype$ )( $env$ )  $\triangleq$ 
.2        $\exists exprtypes : TypeR^*$  .
.3         len  $exprs$  = len  $exprtypes$ 
.4         ^
.5         ( $\forall i \in \text{inds } exprs \cdot wf-Expr(\pi)(exprs(i))(exprtypes(i))(env)$ )
.6         ^
.7         cases  $\pi$  :
.8           mk-DEF({(UNIONCLOSE)})  $\cup \neg$   $\rightarrow$ 
.9             UnionCloseExprCP( $CP$ )( $exprtypes$ )( $restype$ )( $env$ ),
.10            mk-DEF( $\neg$ )  $\rightarrow$  CP( $exprtypes$ )( $restype$ )( $env$ ),
.11            mk-POS( $\neg$ )  $\rightarrow$  ComplCP( $\pi$ )( $CP$ )( $exprtypes$ )( $restype$ )( $env$ )
.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 :  $\Pi \xrightarrow{t} ExprCP \rightarrow ExprCP$ 
.1     ComplCP( $\pi$ )( $CP$ )( $ets$ )( $rt$ )( $env$ )  $\triangleq$ 
.2       let lifteditoCP :  $TypeR^* \xrightarrow{t} \mathbb{B}$ 
.3         lifteditoCP( $cts$ )  $\triangleq$ 
.4         len  $ets$  = len  $cts$   $\wedge$ 
.5          $\forall i \in \text{inds } ets \cdot IsSubtypeR(\pi)(ets(i), cts(i))(env) \wedge$ 
.6          $\exists cppt : TypeR \cdot CP(cts)(cppt)(env)$ 
.7         in
.8          $\exists cts : TypeR^*$  .
.9           lifteditoCP( $cts$ )  $\wedge$  CP( $cts$ )( $rt$ )( $env$ )  $\wedge$ 
.10           $\forall octs : TypeR^*$  .
.11            lifteditoCP( $octs$ )  $\Rightarrow$ 
.12             $\forall i \in \text{inds } cts \cdot$ 
.13              IsSubtypeR( $\pi$ )( $octs(i), cts(i)$ )( $env$ )  $\Rightarrow$  IsSubtypeR( $\pi$ )( $cts(i), octs(i)$ )( $env$ )

```

ComplCP(π)(CP) completes the expression characteristic predicate *CP* so it satisfies a completeness property in connection with possible well-formedness: If $CP(pt_1, \dots, pt_n)(rt)(env)$ is true and the types t_i are subtypes of pt_i then $(ComplCP(CP))(t_1, \dots, t_n)(rt)(env)$ 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.

11.3.2 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 section 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*.

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 equivalent to t . Furthermore, for definite well-formedness (i.e. *is-DEF*(π)) 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*(π)(t, t')(env) and for possible well-formedness (i.e. *is-POS*(π)) 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*(π)(t', t)(env).

- 175.0 $SubsumeExpr : \Pi \xrightarrow{t} WFE \rightarrow WFE$
- .1 $SubsumeExpr(\pi)(wfe)(expr)(rtype)(env) \triangleq$
 - .2 $\exists et : TypeR .$
 - .3 $wfe(expr)(et)(env) \wedge Subsume(\pi)(et, rtype)(env);$
- 176.0 $Subsume : \Pi \rightarrow TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $Subsume(\pi)(source-t, expected-t)(env) \triangleq$
 - .2 cases $\pi :$
 - .3 $mk\text{-}DEF(-) \rightarrow IsSubtypeR(\pi)(source-t, expected-t)(env),$
 - .4 $mk\text{-}POS(-) \rightarrow IsSubtypeR(\pi)(expected-t, source-t)(env)$
 - .5 end

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 transformed predicate *InType*(π)(*wfe*) will for *is-DEF*(π) behave exactly as the untransformed predicate *wfe*. However, for possible well-formedness, i.e. *is-POS*(π), *InType*(π)(*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-POS*(π), then it may be rejected that the expression e evaluates to a value in the type ct .

177.0 $InType[@C] : \Pi \xrightarrow{t} (@C \rightarrow WF) \rightarrow @C \rightarrow WF$
 .1 $InType(\pi)(wfc)(c)(expectedt)(env) \triangleq$
 .2 $\exists rt : TypeR .$
 .3 $wfc(c)(rt)(env) \wedge$
 .4 $\text{cases } \pi :$
 .5 $mk\text{-}DEF(-) \rightarrow rt = expectedt,$
 .6 $mk\text{-}POS(-) \rightarrow IsSubtypeR(\pi)(rt, expectedt)(env)$
 .7 end

Union Close

This section 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 $UnionCloseExprCP : ExprCP \xrightarrow{t} ExprCP$
 .1 $UnionCloseExprCP(cp)(ats)(rt)(env) \triangleq$
 .2 $\forall aats : TypeR^*. \quad$
 .3 $\text{len } aats = \text{len } ats \wedge$
 .4 $\forall i \in \text{inds } aats .$
 .5 $IsUnionComponent(aats(i), ats(i))(env)$
 .6 \Rightarrow
 .7 $\exists art : TypeR .$
 .8 $IsUnionComponent(art, rt)(env) \wedge cp(aats)(art)(env)$

Sub-Environment Based Checks

The higher order functions described in this section 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 $InScope[@C] : \Pi \xrightarrow{t} Name\text{-}set \rightarrow (@C \rightarrow WF) \rightarrow @C \rightarrow WF$
 .1 $InScope(\pi)(locnames)(wfc)(c)(type)(env) \triangleq$
 .2 $\text{cases } \pi :$
 .3 $mk\text{-}POS(\{\text{LOCALCONTEXT}\}) \cup - \rightarrow$
 .4 $\exists subenv : Env .$
 .5 $IsSubEnvLoc(\pi)(locnames)(subenv, env) \wedge wfc(c)(type)(subenv),$
 .6 $\text{others} \rightarrow wfc(c)(type)(env)$
 .7 end

The next function concerns check of *total functions* (without preconditions). In general, for a function defined to be 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 $InTotalFnScope : \Pi \xrightarrow{t} Name\text{-set} \rightarrow WFE \rightarrow WFE$

- .1 $InTotalFnScope(\pi)(locnames)(wfe)(expr)(type)(env) \triangleq$
- .2 cases π :
- .3 $mk\text{-POS}(\{\text{TOTALFUNC}\}) \cup - \rightarrow$
- .4 $\forall subenv : Env .$
- .5 $IsSubEnvLoc(\pi)(locnames)(subenv, env) \wedge wfe(expr)(type)(subenv),$
- .6 others $\rightarrow InScope[Expr](\pi)(locnames)(wfe)(expr)(type)(env)$
- .7 end

Sub-Environments

181.0 $IsSubEnvLoc : \Pi \xrightarrow{t} Name\text{-set} \rightarrow Env \times Env \rightarrow \mathbb{B}$

- .1 $IsSubEnvLoc(\pi)(locnames)(subenv, env) \triangleq$
- .2 $subenv = \mu (env, visibleenv \mapsto subenv.visibleenv) \wedge$
- .3 let $subVenv = subenv.visibleenv,$
- .4 $Venv = env.visibleenv$ in
- .5 $\text{dom } subVenv = \text{dom } Venv \wedge$
- .6 $\forall name \in \text{dom } Venv \setminus locnames \cdot subVenv(name) = Venv(name) \wedge$
- .7 $\forall name \in \text{dom } Venv \cap locnames .$
- .8 cases $mk\text{-(}subVenv(name), Venv(name)\text{)} :$
- .9 $mk\text{-(}mk\text{-}ValR(subt, -), mk\text{-}ValR(t, -)\text{)} \rightarrow IsUnionComponent(subt, t)(env),$
- .10 $mk\text{-(}mk\text{-}VarR(subt), mk\text{-}VarR(t)\text{)} \rightarrow IsUnionComponent(subt, t)(env),$
- .11 $mk\text{-(}ssrep1, ssrep2\text{)} \rightarrow ssrep1 = ssrep2$
- .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 occurring in $locnames$ to the same type (line 181.6). Finally, $subenv$'s visible environment will map identifiers occurring in $locnames$ to types which are union components of the corresponding type in env 's visible environment.

11.3.3 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 $mk\text{-}BracketedExpr(-) \rightarrow wf\text{-}BracketedExpr(\pi)(e)(t)(env),$
- .4 $mk\text{-}LetExpr(-, -) \rightarrow wf\text{-}LetExpr(\pi)(e)(t)(env),$
- .5 $mk\text{-}LetBeSE Expr(-, -, -) \rightarrow wf\text{-}LetBeSE Expr(\pi)(e)(t)(env),$
- .6 $mk\text{-}DefExpr(-, -) \rightarrow wf\text{-}DefExpr(\pi)(e)(t)(env),$
- .7 $mk\text{-}IfExpr(-, -, -, -) \rightarrow wf\text{-}IfExpr(\pi)(e)(t)(env),$
- .8 $mk\text{-}CasesExpr(-, -, -) \rightarrow wf\text{-}CasesExpr(\pi)(e)(t)(env),$
- .9 $mk\text{-}PrefixExpr(-, -) \rightarrow wf\text{-}PrefixExpr(\pi)(e)(t)(env),$
- .10 $mk\text{-}MapInverseExpr(-) \rightarrow wf\text{-}MapInverseExpr(\pi)(e)(t)(env),$
- .11 $mk\text{-}BinaryExpr(-, -, -) \rightarrow wf\text{-}BinaryExpr(\pi)(e)(t)(env),$
- .12 $mk\text{-}AllExpr(-, -) \rightarrow wf\text{-}AllExpr(\pi)(e)(t)(env),$
- .13 $mk\text{-}ExistsExpr(-, -) \rightarrow wf\text{-}ExistsExpr(\pi)(e)(t)(env),$
- .14 $mk\text{-}ExistsUniqueExpr(-, -) \rightarrow wf\text{-}ExistsUniqueExpr(\pi)(e)(t)(env),$
- .15 $mk\text{-}IotaExpr(-, -) \rightarrow wf\text{-}IotaExpr(\pi)(e)(t)(env),$
- .16 $mk\text{-}SetEnumeration(-) \rightarrow wf\text{-}SetEnumeration(\pi)(e)(t)(env),$
- .17 $mk\text{-}SetComprehension(-, -, -) \rightarrow wf\text{-}SetComprehension(\pi)(e)(t)(env),$
- .18 $mk\text{-}SetRange(-, -) \rightarrow wf\text{-}SetRange(\pi)(e)(t)(env),$
- .19 $mk\text{-}SeqEnumeration(-) \rightarrow wf\text{-}SeqEnumeration(\pi)(e)(t)(env),$
- .20 $mk\text{-}SeqComprehension(-, -, -) \rightarrow wf\text{-}SeqComprehension(\pi)(e)(t)(env),$
- .21 $mk\text{-}SubSequence(-, -, -) \rightarrow wf\text{-}SubSequence(\pi)(e)(t)(env),$
- .22 $mk\text{-}MapEnumeration(-) \rightarrow wf\text{-}MapEnumeration(\pi)(e)(t)(env),$
- .23 $mk\text{-}MapComprehension(-, -, -) \rightarrow wf\text{-}MapComprehension(\pi)(e)(t)(env),$
- .24 $mk\text{-}TupleConstructor(-) \rightarrow wf\text{-}TupleConstructor(\pi)(e)(t)(env),$
- .25 $mk\text{-}RecordConstructor(-, -) \rightarrow wf\text{-}RecordConstructor(\pi)(e)(t)(env),$
- .26 $mk\text{-}RecordModifier(-, -) \rightarrow wf\text{-}RecordModifier(\pi)(e)(t)(env),$
- .27 $mk\text{-}Apply(-, -) \rightarrow wf\text{-}Apply(\pi)(e)(t)(env),$
- .28 $mk\text{-}FieldSelect(-, -) \rightarrow wf\text{-}FieldSelect(\pi)(e)(t)(env),$
- .29 $mk\text{-}FctTypeInst(-, -) \rightarrow wf\text{-}FctTypeInst(\pi)(e)(t)(env),$
- .30 $mk\text{-}Lambda(-, -) \rightarrow wf\text{-}Lambda(\pi)(e)(t)(env),$
- .31 $mk\text{-}IsDefTypeExpr(-, -) \rightarrow wf\text{-}IsDefTypeExpr(\pi)(e)(t)(env),$
- .32 $mk\text{-}IsBasicTypeExpr(-, -) \rightarrow wf\text{-}IsBasicTypeExpr(\pi)(e)(t)(env),$
- .33 $mk\text{-}Name(-) \rightarrow wf\text{-}Name(\pi)(e)(t)(env),$
- .34 $mk\text{-}OldName(-) \rightarrow wf\text{-}OldName(\pi)(e)(t)(env),$
- .35 $\text{others} \rightarrow wf\text{-}Literal(\pi)(e)(t)(env)$
- .36 end

$wf\text{-}ExprBasic$ 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 $wf\text{-}Expr$ is defined. $wf\text{-}Expr$ is the well-formedness predicate to be used for checking all sub-component 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 $SubsumeExpr(\pi)(wf\text{-}ExprBasic(\pi))$

11.3.4 Bracketed Expression

184.0 $wf\text{-}BracketedExpr : \Pi \xrightarrow{t} BracketedExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}BracketedExpr(\pi)(mk\text{-}BracketedExpr(e))(type)(env) \triangleq$
 .2 $wf\text{-}Expr(\pi)(e)(type)(env)$

11.3.5 Local Binding Expressions

Let Expression

185.0 $wf\text{-}LetExpr : \Pi \xrightarrow{t} LetExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}LetExpr(\pi)(mk\text{-}LetExpr(localdefs, body))(type)(env) \triangleq$
 .2 $\exists venv : VisibleEnv .$
 .3 $\quad let wfe = InScope[Expr](\pi)(dom venv)(wf\text{-}Expr(\pi)),$
 .4 $\quad lenv = UpdateEnv(venv)(env) \text{ in}$
 .5 $\quad wf\text{-}ValFnOpDefs(\pi)(localdefs)(venv)(env) \wedge$
 .6 $\quad wfe(body)(type)(lenv)$

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*).

Let Be ST Expression

186.0 $wf\text{-}LetBeSTExpr : \Pi \xrightarrow{t} LetBeSTExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}LetBeSTExpr(\pi)(mk\text{-}LetBeSTExpr(bind, ocond, body))(type)(env) \triangleq$
 .2 $(\exists venv : ValEnv, bindattrs : BindAttrs .$
 .3 $\quad wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \wedge$
 .4 $\quad let newenv = UpdateEnv(venv)(env),$
 .5 $\quad wfe = (InScope[Expr](\pi)(dom venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
 .6 $\quad (ocond \neq \text{nil} \Rightarrow$
 .7 $\quad \exists truetype : TypeR .$
 .8 $\quad wfe(ocond)(truetype)(newenv) \wedge IsTrueType(\pi)(truetype)(env)) \wedge$
 .9 $\quad InScope[Expr](\pi)(dom venv)(wf\text{-}Expr(\pi))(body)(type)(newenv)) \wedge$
 .10 $\neg \exists venv : ValEnv, bindattrs : BindAttrs .$
 .11 $\quad wf\text{-}Bind(NegWfClass(\pi))(bind)(venv, bindattrs)(env) \wedge$
 .12 $\quad \text{EMPTY} \in bindattrs$

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 one cannot accept that `EMPTY` is in `bindattrs`, then one cannot reject that the set of generated value environments is non-empty.

187.0 $IsTrueType : \Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

```
.1   $IsTrueType(\pi)(t)(env) \triangleq$ 
.2  cases  $\pi$  :
.3   $mk\text{-DEF}(-) \rightarrow \text{false},$ 
.4   $mk\text{-POS}(-) \rightarrow t = \text{BOOLEAN}$ 
.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.

Def Expression

188.0 $wf\text{-DefExpr} : \Pi \xrightarrow{t} DefExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

```
.1   $wf\text{-DefExpr}(\pi)(mk\text{-DefExpr}(defs, body))(type)(env) \triangleq$ 
.2   $\exists venv : \text{VisibleEnv} .$ 
.3  let  $wfe = \text{InScope}[Expr](\pi)(\text{dom } venv)(wf\text{-Expr}(\pi)),$ 
.4   $lenv = \text{UpdateEnv}(venv)(env) \text{ in }$ 
.5   $wf\text{-OrderedDefs}(\pi)(defs, \text{false})(venv)(env) \wedge$ 
.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 `DefExprs`, 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).

11.3.6 Conditional Expressions

If Expression

189.0 $wf\text{-IfExpr} : \Pi \xrightarrow{t} IfExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

```
.1   $wf\text{-IfExpr}(\pi)(mk\text{-IfExpr}(test, cons, elseifaltns, altn))(type)(env) \triangleq$ 
.2  let  $altn' : Expr =$ 
.3  cases  $elseifaltns :$ 
.4   $([]) \rightarrow altn,$ 
.5   $[mk\text{-ElsifExpr}(test_1, cons_1)] \curvearrowright raltns \rightarrow$ 
.6   $mk\text{-IfExpr}(test_1, cons_1, raltns, altn)$ 
.7  end in
.8   $\text{InType}[Expr](\pi)(wf\text{-Expr}(\pi))(test)(\text{BOOLEAN})(env) \wedge$ 
.9  cases  $\pi :$ 
.10  $mk\text{-DEF}(-) \rightarrow$ 
.11  $wf\text{-Expr}(\pi)(cons)(type)(env) \wedge wf\text{-Expr}(\pi)(altn')(type)(env),$ 
.12  $mk\text{-POS}(-) \rightarrow$ 
.13  $wf\text{-Expr}(\pi)(cons)(type)(env) \vee wf\text{-Expr}(\pi)(altn')(type)(env)$ 
.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 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 definitely 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 definitely 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 be definitely well-formed for some result *type* not less than the union type $t_1 \sqcup t_2$.

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 possible 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

Cases Expression

```

190.0  wf-CasesExpr :  $\Pi \xrightarrow{t} CasesExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1    wf-CasesExpr ( $\pi$ )( $e$ )(type)(env)  $\triangleq$ 
.2    let  $mk\text{-}CasesExpr(sel, altns, (\text{nil})) = RemoveCaseOther(e)$  in
.3     $\exists selt : TypeR .$ 
.4       $wf\text{-}Expr(\pi)(sel)(selt)(env) \wedge$ 
.5      cases  $\pi :$ 
.6         $mk\text{-}DEF(-) \rightarrow$ 
.7         $(\forall alt \in \text{elems } altns .$ 
.8           $\forall pat \in \text{elems } alt.\text{match} .$ 
.9             $\forall matcht : TypeR, venv : ValEnv .$ 
.10            $(\text{let } \pi' = mk\text{-}POS(\pi.\text{checkset}) \text{ in }$ 
.11              $wf\text{-}Expr(\pi')(sel)(matcht)(env) \wedge$ 
.12              $wf\text{-}Pattern(\pi')(pat)(mk\text{-}TVE(matcht, venv))(env))$ 
.13            $\Rightarrow$ 
.14           let  $wfe = InScope[Expr](\pi)(\text{dom } venv)(wf\text{-}Expr(\pi))$  in
.15            $wfe(alt.\text{body})(type)(UpdateEnv(venv)(env)))$ 
.16            $\wedge$ 
.17            $CanMatch(\pi)(mk\text{-}CasesExpr(sel, altns, \text{nil}))(env),$ 
.18            $mk\text{-}POS(-) \rightarrow$ 
.19            $\exists alt \in \text{elems } altns .$ 
.20            $\exists pat \in \text{elems } alt.\text{match} .$ 
.21            $\exists venv : ValEnv .$ 
.22            $wf\text{-}Pattern(\pi)(pat)(mk\text{-}TVE(selt, venv))(env) \wedge$ 
.23           let  $wfe = InScope[Expr](\pi)(\text{dom } venv)(wf\text{-}Expr(\pi))$  in
.24            $wfe(alt.\text{body})(type)(UpdateEnv(venv)(env))$ 
.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 separately.

For definite well-formedness it is required that all alternative expressions which may be evaluated in the Dynamic Semantics are definitely 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 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 *definitely* 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 (*sel*) (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 :  $\Pi \xrightarrow{t} \text{CasesExpr} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1     CanMatch ( $\pi$ )(mk-CasesExpr(sel, altns, (nil)))(env)  $\triangleq$ 
.2       let Is-matchtypes : TypeR-set  $\rightarrow \mathbb{B}$ 
.3         Is-matchtypes (mtypes)  $\triangleq$ 
.4            $\forall mtype \in mtypes .$ 
.5              $\exists alt \in \text{elems } altns .$ 
.6                $\exists pat \in \text{elems } alt.match, venv : \text{ValEnv} .$ 
.7                 wf-Pattern( $\pi$ )(pat)(mk-TVE(mtype, venv))(env)
.8               in
.9                  $\exists mtypes : \text{TypeR-set}, seltype : \text{TypeR} .$ 
.10                Is-matchtypes(mtypes)  $\wedge$ 
.11                wf-Expr( $\pi$ )(sel)(seltype)(env)  $\wedge$ 
.12                IsSubtypeR( $\pi$ )(seltype, mk-UnionTypeR(mtypes))(env)
.13   pre is-DEF( $\pi$ )

```

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 definitely 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 definitely will match some pattern of some alternative.

```

192.0   RemoveCaseOther : CasesExpr  $\xrightarrow{t} \text{CasesExpr}$ 
.1     RemoveCaseOther (mk-CasesExpr(sel, altns, other))  $\triangleq$ 
.2       cases other :
.3         (nil)  $\rightarrow$  mk-CasesExpr(sel, altns, other),
.4         others  $\rightarrow$  let lastalt = mk-CaseAltn([mk-PatternId(nil)], other) in
.5           mk-CasesExpr(sel, altns  $\setminus [lastalt]$ , nil)
.6       end

```

11.3.7 Unary Expressions

All unary expressions are checked by using characteristic predicates.

```

193.0 wf-PrefixExpr :  $\Pi \xrightarrow{t} PrefixExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1 wf-PrefixExpr ( $\pi$ ) (mk-PrefixExpr (pref, expr)) (type) (env)  $\triangleq$ 
.2 cases pref :
.3 (NUMPLUS)  $\rightarrow wf\text{-NUMPLUS}(\pi)(expr)(type)(env)$ ,
.4 (NUMMINUS)  $\rightarrow wf\text{-NUMMINUS}(\pi)(expr)(type)(env)$ ,
.5 (NUMABS)  $\rightarrow wf\text{-NUMABS}(\pi)(expr)(type)(env)$ ,
.6 (FLOOR)  $\rightarrow wf\text{-FLOOR}(\pi)(expr)(type)(env)$ ,
.7 (NOT)  $\rightarrow wf\text{-NOT}(\pi)(expr)(type)(env)$ ,
.8 (SETCARD)  $\rightarrow wf\text{-SETCARD}(\pi)(expr)(type)(env)$ ,
.9 (SETPOWER)  $\rightarrow wf\text{-SETPOWER}(\pi)(expr)(type)(env)$ ,
.10 (SETDISTRUNION)  $\rightarrow wf\text{-SETDISTRUNION}(\pi)(expr)(type)(env)$ ,
.11 (SETDISTRINTERSECT)  $\rightarrow wf\text{-SETDISTRINTERSECT}(\pi)(expr)(type)(env)$ ,
.12 (SEQHEAD)  $\rightarrow wf\text{-SEQHEAD}(\pi)(expr)(type)(env)$ ,
.13 (SEQTAIL)  $\rightarrow wf\text{-SEQTAIL}(\pi)(expr)(type)(env)$ ,
.14 (SEQLEN)  $\rightarrow wf\text{-SEQLEN}(\pi)(expr)(type)(env)$ ,
.15 (SEQELEMS)  $\rightarrow wf\text{-SEQELEMS}(\pi)(expr)(type)(env)$ ,
.16 (SEQINDICES)  $\rightarrow wf\text{-SEQINDICES}(\pi)(expr)(type)(env)$ ,
.17 (SEQDISTRCONC)  $\rightarrow wf\text{-SEQDISTRCONC}(\pi)(expr)(type)(env)$ ,
.18 (MAPDOM)  $\rightarrow wf\text{-MAPDOM}(\pi)(expr)(type)(env)$ ,
.19 (MAPRNG)  $\rightarrow wf\text{-MAPRNG}(\pi)(expr)(type)(env)$ ,
.20 (MAPDISTRMERGE)  $\rightarrow wf\text{-MAPDISTRMERGE}(\pi)(expr)(type)(env)$ 
.21 end

```

Numeric Operations

```

194.0 wf-NUMPLUS :  $\Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1 wf-NUMPLUS ( $\pi$ ) (e) (type) (env)  $\triangleq$ 
.2 let CP :  $TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3 CP ([et]) (rt) (env)  $\triangleq$ 
.4 IsSubtypeR ( $\pi$ ) (et, REAL) (env)  $\wedge rt = e_t$ 
.5 in
.6 wf-SCompExpr ( $\pi$ ) (CP) ([e]) (type) (env);

195.0 wf-NUMMINUS :  $\Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1 wf-NUMMINUS ( $\pi$ ) (e) (type) (env)  $\triangleq$ 
.2 let CP :  $TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3 CP ([et]) (rt) (env)  $\triangleq$ 
.4 IsSubtypeR ( $\pi$ ) (et, REAL) (env)  $\wedge$ 
.5 rt = mk-UnionTypeR ({et, INT})
.6 in
.7 wf-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.

196.0 $wf\text{-}NUMABS : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}NUMABS(\pi)(e)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([e_t])(rt)(env) \triangleq$
 .4 $IsSubtypeR(\pi)(e_t, REAL)(env) \wedge$
 .5 cases $e_t :$
 .6 $(INT) \rightarrow rt = NAT,$
 .7 others $\rightarrow rt = e_t$
 .8 end
 .9 in
 .10 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env);$

197.0 $wf\text{-}FLOOR : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}FLOOR(\pi)(e)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([e_t])(rt)(env) \triangleq$
 .4 $IsIntersectionType(\pi)(INT, e_t)(rt)(env)$
 .5 in
 .6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

Logical Operation

198.0 $wf\text{-}NOT : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}NOT(\pi)(e)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([e_t])(rt)(env) \triangleq$
 .4 $e_t = BOOLEAN \wedge rt = BOOLEAN$
 .5 in
 .6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

Set Operations

199.0 $wf\text{-SETCARD} : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SETCARD}(\pi)(e)(type)(env) \triangleq$
- .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $is\text{-SetTypeR}(e_t) \wedge rt = NAT$
- .5 in
- .6 $wf\text{-SCompExpr}(\pi)(CP)([e])(type)(env);$

200.0 $wf\text{-SETPOWER} : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SETPOWER}(\pi)(e)(type)(env) \triangleq$
- .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .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 $wf\text{-SETDISTRUNION} : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SETDISTRUNION}(\pi)(e)(type)(env) \triangleq$
- .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $\exists t : TypeR .$
- .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 $wf\text{-SETDISTRINTERSECT} : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SETDISTRINTERSECT}(\pi)(e)(type)(env) \triangleq$
- .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .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 $wf\text{-SCompExpr}(\pi)(CP)([e])(type)(env)$

Sequence Operations

203.0 $wf\text{-}SEQHEAD : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}SEQHEAD(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .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 $wf\text{-}SEQTAIL : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}SEQTAIL(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .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 $wf\text{-}SEQLEN : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}SEQLEN(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $\text{is}\text{-}Seq0 TypeR(e_t) \wedge rt = \text{NAT} \vee$
- .5 $\text{is}\text{-}Seq1 TypeR(e_t) \wedge rt = \text{NATONE}$
- .6 in
- .7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env);$

206.0 $wf\text{-}SEQELEMS : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}SEQELEMS(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $\exists t : TypeR \cdot$
- .5 $\text{IsSeqTypeR}(e_t)(t) \wedge rt = mk\text{-}SetTypeR(t)$
- .6 in
- .7 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env);$

207.0 $wf\text{-}SEQINDICES : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}SEQINDICES(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $(\exists elem : TypeR \cdot \text{IsSeqTypeR}(e_t)(elem)) \wedge rt = mk\text{-}SetTypeR(\text{NATONE})$
- .5 in
- .6 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env);$

208.0 $wf\text{-}SEQDISTRCONC : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}SEQDISTRCONC(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $\exists seqt, t : TypeR \cdot$
- .5 $\text{IsSeqTypeR}(e_t)(seqt) \wedge \text{IsSeqTypeR}(seqt)(t) \wedge$
- .6 $\text{if is}\text{-}Seq1 TypeR(e_t) \wedge \text{is}\text{-}Seq1 TypeR(seqt)$
- .7 $\text{then rt} = mk\text{-}Seq1 TypeR(t) \text{ else rt} = mk\text{-}Seq0 TypeR(t)$
- .8 in
- .9 $wf\text{-}SCompExpr(\pi)(CP)([e])(type)(env)$

Map Operations

209.0 $wf\text{-}MAPDOM : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}MAPDOM(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $\exists t_1, t_2 : TypeR .$
- .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 $wf\text{-}MAPRNG : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}MAPRNG(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $\exists t_1, t_2 : TypeR .$
- .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 $wf\text{-}MAPDISTRMERGE : \Pi \xrightarrow{t} Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}MAPDISTRMERGE(\pi)(e)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([e_t])(rt)(env) \triangleq$
- .4 $\exists mapt, t_1, t_2 : TypeR .$
- .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 $CompatibleMapSet : \Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $CompatibleMapSet(\pi)(type)(env) \triangleq$
- .2 $\exists mapt, t_1, t_2 : TypeR .$
- .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.

Map Inverse Expression

```
213.0   wf-MapInverseExpr : Π → MapInverseExpr → TypeR → Env → B
.1      wf-MapInverseExpr (π)(mk-MapInverseExpr(mape))(type)(env) △
.2      let CP : TypeR* → TypeR → Env → B
.3          CP ([mapt])(rt)(env) △
.4          ∃ t1, t2 : TypeR .
.5              mapt = mk-InjectiveMapTypeR(t1, t2) ∧
.6              rt = mk-InjectiveMapTypeR(t2, t1) ∧
.7              IsFlatTypeR(π)(t2)(env)
.8          in
.9      wf-SCompExpr(π)(CP)([mape])(type)(env)
```

11.3.8 Binary Expressions

The binary expressions – other than some of the logical operations – are checked by using characteristic predicates.

```

214.0   wf-BinaryExpr :  $\Pi \xrightarrow{t} BinaryExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-BinaryExpr ( $\pi$ )(mk-BinaryExpr( $le$ ,  $opr$ ,  $re$ ))( $type$ )( $env$ )  $\triangleq$ 
.2       cases  $opr$  :
.3         (EQ)  $\rightarrow$  wf-EQ( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.4         (NE)  $\rightarrow$  wf-NE( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.5         (OR)  $\rightarrow$  wf-OR( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.6         (AND)  $\rightarrow$  wf-AND( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.7         (IMPLY)  $\rightarrow$  wf-IMPLY( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.8         (EQUIV)  $\rightarrow$  wf-EQUIV( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.9         (NUMLT)  $\rightarrow$  wf-NUMLT( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.10        (NUMLE)  $\rightarrow$  wf-NUMLE( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.11        (NUMGT)  $\rightarrow$  wf-NUMGT( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.12        (NUMGE)  $\rightarrow$  wf-NUMGE( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.13        (INSET)  $\rightarrow$  wf-INSET( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.14        (NOTINSET)  $\rightarrow$  wf-NOTINSET( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.15        (SUBSET)  $\rightarrow$  wf-SUBSET( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.16        (PROPERSUBSET)  $\rightarrow$  wf-PROPERSUBSET( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.17        (NUMPLUS)  $\rightarrow$  wf-NUMBINPLUS( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.18        (NUMMINUS)  $\rightarrow$  wf-NUMBINMINUS( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.19        (NUMMULT)  $\rightarrow$  wf-NUMMULT( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.20        (NUMDIV)  $\rightarrow$  wf-NUMDIV( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.21        (NUMREM)  $\rightarrow$  wf-NUMREM( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.22        (NUMMOD)  $\rightarrow$  wf-NUMMOD( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.23        (INTDIV)  $\rightarrow$  wf-INTDIV( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.24        (SETUNION)  $\rightarrow$  wf-SETUNION( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.25        (SETDIFFERENCE)  $\rightarrow$  wf-SETDIFFERENCE( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.26        (SETINTERSECT)  $\rightarrow$  wf-SETINTERSECT( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.27        (SEQCONC)  $\rightarrow$  wf-SEQCONC( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.28        (MAPMERGE)  $\rightarrow$  wf-MAPMERGE( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.29        (MAPORSEQMOD)  $\rightarrow$  wf-MAPORSEQMOD( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.30        (MAPDOMRESTRTO)  $\rightarrow$  wf-MAPDOMRESTRTO( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.31        (MAPDOMRESTRBY)  $\rightarrow$  wf-MAPDOMRESTRBY( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.32        (MAPRNGRESTRTO)  $\rightarrow$  wf-MAPRNGRESTRTO( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.33        (MAPRNGRESTRBY)  $\rightarrow$  wf-MAPRNGRESTRBY( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.34        (COMPOSE)  $\rightarrow$  wf-COMPOSE( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ ),
.35        (ITERATE)  $\rightarrow$  wf-ITERATE( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ )
.36      end

```

Numeric Operations

```

215.0   wf-NUMBINPLUS :  $\Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-NUMBINPLUS ( $\pi$ )( $le$ ,  $re$ )( $type$ )( $env$ )  $\triangleq$ 
.2       let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3          $CP ([let_t, re_t])(rt)(env) \triangleq$ 
.4          $(\forall t \in \{let_t, re_t\} \cdot IsSubtypeR(\pi)(t, REAL)(env)) \wedge$ 
.5          $rt = mk-UnionTypeR(\{let_t, re_t\})$ 
.6         in
.7         wf-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 $wf\text{-NUMBINMINUS} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-NUMBINMINUS}(\pi)(le, re)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 .4 $(\forall t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, REAL)(env)) \wedge$
 .5 $rt = mk\text{-UnionTypeR}(\{le_t, re_t, INT\})$
 .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 $wf\text{-NUMMULT} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-NUMMULT}(\pi)(le, re)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 .4 $(\forall t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, REAL)(env)) \wedge$
 .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 $wf\text{-NUMDIV} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-NUMDIV}(\pi)(le, re)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 .4 $(\forall t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, REAL)(env)) \wedge$
 .5 $rt = mk\text{-UnionTypeR}(\{le_t, re_t, RAT\}) \wedge$
 .6 $(is\text{-DEF}(\pi) \Rightarrow IsSubtypeR(\pi)(re_t, NATONE)(env))$
 .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).

219.0 $wf\text{-INTDIV} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-INTDIV}(\pi)(le, re)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 .4 $rt = INT \wedge$
 .5 $le_t = INT \wedge$
 .6 $re_t = \text{if } is\text{-POS}(\pi) \text{ then INT else NATONE}$
 .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 $wf\text{-}NUMREM : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}NUMREM(\pi)(le, re)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 .4 $(\forall t \in \{le_t, re_t\} \cdot IsSubtypeR(\pi)(t, INT)(env)) \wedge$
 .5 $rt = mk\text{-}UnionTypeR(\{le_t, re_t, NAT\}) \wedge$
 .6 $(is\text{-}DEF(\pi) \Rightarrow IsSubtypeR(\pi)(re_t, NATONE)(env))$
 .7 in
 .8 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

221.0 $wf\text{-}NUMMOD : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}NUMMOD(\pi)(le, re)(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 .4 $le_t = NAT \wedge re_t = NATONE \wedge$
 .5 $rt = NAT$
 .6 in
 .7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Numeric Comparison Operators

```

222.0   wf-NUMLT :  $\Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1      wf-NUMLT ( $\pi$ )( $le, re$ )( $type$ )( $env$ )  $\triangleq$ 
.2      let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3       $CP ([le_t, re_t])(rt)(env) \triangleq$ 
.4       $(\forall t \in \{le_t, re_t\} \cdot t = REAL) \wedge$ 
.5       $rt = BOOLEAN$ 
.6      in
.7      wf-SCompExpr( $\pi$ )( $CP$ )([ $le, re$ ])( $type$ )( $env$ );

223.0   wf-NUMLE :  $\Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1      wf-NUMLE ( $\pi$ )( $le, re$ )( $type$ )( $env$ )  $\triangleq$ 
.2      let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3       $CP ([le_t, re_t])(rt)(env) \triangleq$ 
.4       $(\forall t \in \{le_t, re_t\} \cdot t = REAL) \wedge$ 
.5       $rt = BOOLEAN$ 
.6      in
.7      wf-SCompExpr( $\pi$ )( $CP$ )([ $le, re$ ])( $type$ )( $env$ );

224.0   wf-NUMGT :  $\Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1      wf-NUMGT ( $\pi$ )( $le, re$ )( $type$ )( $env$ )  $\triangleq$ 
.2      let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3       $CP ([le_t, re_t])(rt)(env) \triangleq$ 
.4       $(\forall t \in \{le_t, re_t\} \cdot t = REAL) \wedge$ 
.5       $rt = BOOLEAN$ 
.6      in
.7      wf-SCompExpr( $\pi$ )( $CP$ )([ $le, re$ ])( $type$ )( $env$ );

225.0   wf-NUMGE :  $\Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1      wf-NUMGE ( $\pi$ )( $le, re$ )( $type$ )( $env$ )  $\triangleq$ 
.2      let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3       $CP ([le_t, re_t])(rt)(env) \triangleq$ 
.4       $(\forall t \in \{le_t, re_t\} \cdot t = REAL) \wedge$ 
.5       $rt = BOOLEAN$ 
.6      in
.7      wf-SCompExpr( $\pi$ )( $CP$ )([ $le, re$ ])( $type$ )( $env$ )

```

Equality operations

- 226.0 $wf-EQ : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf-EQ(\pi)(le, re)(type)(env) \triangleq$
 - .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 - .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 - .4 $rt = \text{BOOLEAN}$
 - .5 in
 - .6 $wf-SCompExpr(\pi)(CP)([le, re])(type)(env);$
- 227.0 $wf-NE : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf-NE(\pi)(le, re)(type)(env) \triangleq$
 - .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 - .3 $CP([le_t, re_t])(rt)(env) \triangleq$
 - .4 $rt = \text{BOOLEAN}$
 - .5 in
 - .6 $wf-SCompExpr(\pi)(CP)([le, re])(type)(env)$

Logical Operations

- 228.0 $wf-OR : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf-OR(\pi)(le, re)(type)(env) \triangleq$
 - .2 $wf-NonStrictLOGOPRT(\pi)(le, re)(type)(env);$
- 229.0 $wf-AND : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf-AND(\pi)(le, re)(type)(env) \triangleq$
 - .2 $wf-NonStrictLOGOPRT(\pi)(le, re)(type)(env);$
- 230.0 $wf-IMPLY : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf-IMPLY(\pi)(le, re)(type)(env) \triangleq$
 - .2 $wf-NonStrictLOGOPRT(\pi)(le, re)(type)(env)$

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).

231.0 $wf\text{-}NonStrictLOGOPRT : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}NonStrictLOGOPRT(\pi)(le, re)(type)(env) \triangleq$
 .2 $(\text{let } \pi_D = mk\text{-}DEF(\pi.\text{checkset}) \text{ in}$
 .3 $\text{cases } \pi :$
 .4 $mk\text{-}DEF(-) \rightarrow$
 .5 $\forall e \in \{le, re\} .$
 .6 $InType[Expr](\pi)(wf\text{-}Expr(\pi))(e)(\text{BOOLEAN})(env),$
 .7 $mk\text{-}POS(-) \rightarrow$
 .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 $\forall t : TypeR .$
 .11 $wf\text{-}Expr(\pi_D)(e_2)(t)(env) \Rightarrow$
 .12 $IsOverlapping(\pi)(t, \text{BOOLEAN})(env)$
 .13 $\text{end})$
 .14 \wedge
 .15 $type = \text{BOOLEAN};$

232.0 $wf\text{-}EQUIV : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}EQUIV(\pi)(le, re)(type)(env) \triangleq$
 .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .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 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

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 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([let_t, re_t])(rt)(env) \triangleq$
- .4 $\text{is-SetTypeR}(re_t) \wedge rt = \text{BOOLEAN}$
- .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 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([let_t, re_t])(rt)(env) \triangleq$
- .4 $\text{is-SetTypeR}(re_t) \wedge rt = \text{BOOLEAN}$
- .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 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([let_t, re_t])(rt)(env) \triangleq$
- .4 $\text{is-SetTypeR}(le_t) \wedge \text{is-SetTypeR}(re_t) \wedge$
- .5 $rt = \text{BOOLEAN}$
- .6 in
- .7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env);$

236.0 $wf\text{-}PROPERSUBSET : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}PROPERSUBSET(\pi)(le, re)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([let_t, re_t])(rt)(env) \triangleq$
- .4 $\text{is-SetTypeR}(le_t) \wedge \text{is-SetTypeR}(re_t) \wedge$
- .5 $rt = \text{BOOLEAN}$
- .6 in
- .7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Set Operations

237.0 $wf\text{-SETUNION} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SETUNION}(\pi)(le, re)(type)(env) \triangleq$
- .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([le_t, re_t])(rt)(env) \triangleq$
- .4 $\exists et_1, et_2 : TypeR .$
- .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 $wf\text{-SETDIFFERENCE} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SETDIFFERENCE}(\pi)(le, re)(type)(env) \triangleq$
- .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 $rt = le_t$
- .6 in
- .7 $wf\text{-SCompExpr}(\pi)(CP)([le, re])(type)(env);$

239.0 $wf\text{-SETINTERSECT} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SETINTERSECT}(\pi)(le, re)(type)(env) \triangleq$
- .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 $IsIntersectionType(\pi)(le_t, re_t)(rt)(env)$
- .6 in
- .7 $wf\text{-SCompExpr}(\pi)(CP)([le, re])(type)(env)$

Sequence Operation

240.0 $wf\text{-SEQCONC} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-SEQCONC}(\pi)(le, re)(type)(env) \triangleq$
- .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([le_t, re_t])(rt)(env) \triangleq$
- .4 $\exists et_1, et_2 : TypeR .$
- .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 $is\text{-Seq1TypeR}(le_t) \vee is\text{-Seq1TypeR}(re_t)$
- .8 then $rt = mk\text{-Seq1TypeR}(newelemtp)$
- .9 else $rt = mk\text{-Seq0TypeR}(newelemtp)$
- .10 in
- .11 $wf\text{-SCompExpr}(\pi)(CP)([le, re])(type)(env)$

Sequence and Map Modification Operation

```

241.0  wf-MAPORSEQMOD :  $\Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1    wf-MAPORSEQMOD ( $\pi$ )( $le, re$ )( $type$ )( $env$ )  $\triangleq$ 
.2    let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3       $CP ([let, ret])(rt)(env) \triangleq$ 
.4       $\exists dt_2, rt_2 : TypeR .$ 
.5       $IsMapTypeR(re_t)(dt_2, rt_2) \wedge$ 
.6       $((\exists dt_1, rt_1 : TypeR .$ 
.7       $IsMapTypeR(le_t)(dt_1, rt_1) \wedge$ 
.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    \vee
.12    cases  $le_t :$ 
.13       $mk-Seq0TypeR(et) \rightarrow rt = mk-Seq0TypeR(mk-UnionTypeR(\{et, rt_2\})),$ 
.14       $mk-Seq1TypeR(et) \rightarrow rt = mk-Seq1TypeR(mk-UnionTypeR(\{et, rt_2\}))$ 
.15    end
.16    \wedge
.17     $IsSubtypeR(\pi)(dt_2, NATONE)(env) \wedge$ 
.18     $MapDomInInds(\pi)(dt_2, le_t)(env))$ 
.19  in
.20  wf-SCompExpr( $\pi$ )( $CP$ )([ $le, re$ ])( $type$ )( $env$ )

```

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 the 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 line .242.2).

```

242.0  MapDomInInds :  $\Pi \xrightarrow{t} TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1    MapDomInInds ( $\pi$ )( $mapdomtype, sequencetype$ )( $env$ )  $\triangleq$ 
.2     $is-POS(\pi)$ 

```

Map Operations

243.0 $wf\text{-MAPMERGE} : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-MAPMERGE}(\pi)(le, re)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([le_t, re_t])(rt)(env) \triangleq$
- .4 $\exists at_1, rt_1, at_2, rt_2 : TypeR .$
- .5 $IsMapTypeR(le_t)(at_1, rt_1) \wedge IsMapTypeR(re_t)(at_2, rt_2) \wedge$
- .6 $Compatible2Maps(\pi)(le_t, re_t)(env)$
- .7 \wedge
- .8 $\text{let newat} = mk\text{-UnionTypeR}(\{at_1, at_2\}),$
- .9 $\text{newrt} = mk\text{-UnionTypeR}(\{rt_1, rt_2\}) \text{ in}$
- .10 $rt = mk\text{-GeneralMapTypeR}(newat, newrt)$
- .11 in
- .12 $wf\text{-SCompExpr}(\pi)(CP)([le, re])(type)(env);$

244.0 $Compatible2Maps : \Pi \rightarrow TypeR \times TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $Compatible2Maps(\pi)(mt_1, mt_2)(env) \triangleq$
- .2 $\exists dt_1, rt_1, dt_2, rt_2 : TypeR .$
- .3 $IsMapTypeR(mt_1)(dt_1, rt_1) \wedge IsMapTypeR(mt_2)(dt_2, rt_2) \wedge$
- .4 $(SameValue(\pi)(rt_1, rt_2)(env) \vee IsDisjoint(\pi)(dt_1, dt_2)(env))$

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.

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 $(\exists mat, mrt : TypeR .$
 .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 $(\exists mat, mrt : TypeR .$
 .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 $(\exists mat, mrt : TypeR .$
 .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 $(\exists mat, mrt : TypeR .$
 .5 $IsMapTypeR(le_t)(mat, mrt) \wedge is\text{-}SetTypeR(re_t)) \wedge rt = le_t$
 .6 in
 .7 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

Compose and Iterate

249.0 $wf\text{-}COMPOSE : \Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}COMPOSE(\pi)(le, re)(type)(env) \triangleq$
- .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .3 $CP([le_t, re_t])(rt)(env) \triangleq$
- .4 $\exists at_1, rt_1, at_2, rt_2 : TypeR .$
- .5 $IsSubtypeR(\pi)(rt_2, at_1)(env)$
- .6 \wedge
- .7 $(IsMapTypeR(le_t)(at_1, rt_1) \wedge IsMapTypeR(re_t)(at_2, rt_2)) \wedge$
- .8 $\text{if } is\text{-InjectiveMapTypeR}(le_t) \wedge is\text{-InjectiveMapTypeR}(re_t)$
- .9 $\text{then } rt = mk\text{-InjectiveMapTypeR}(at_2, rt_1)$
- .10 $\text{else } rt = mk\text{-GeneralMapTypeR}(at_2, rt_1)$
- .11 \vee
- .12 $(IsFunctionTypeR(le_t)(at_1, rt_1) \wedge IsFunctionTypeR(re_t)(at_2, rt_2)) \wedge$
- .13 $\text{if } is\text{-TotalFnTypeR}(le_t) \wedge is\text{-TotalFnTypeR}(re_t)$
- .14 $\text{then } rt = mk\text{-TotalFnTypeR}(at_2, rt_1)$
- .15 $\text{else } rt = mk\text{-PartialFnTypeR}(at_2, rt_1))$
- .16 in
- .17 $wf\text{-}SCompExpr(\pi)(CP)([le, re])(type)(env)$

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 :  $\Pi \xrightarrow{t} Expr \times Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-ITERATE ( $\pi$ ) (le, re)(type)(env)  $\triangleq$ 
.2       let CP : TypeR*  $\xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3         CP ([let, ret])(rt)(env)  $\triangleq$ 
.4           let  $\pi'$  = NegWfClass( $\pi$ ) in
.5             ( $\exists t_1, t_2 : TypeR$  .
.6               IsFunctionTypeR(let)(t1, t2)  $\wedge$ 
.7               IsSubtypeR( $\pi$ )(ret, NAT)(env)  $\wedge$ 
.8               (IsGTE2( $\pi'$ )(ret)(env)  $\Rightarrow$  IsSubtypeR( $\pi$ )(t2, t1)(env))  $\wedge$ 
.9               if IsZero( $\pi'$ )(ret)(env)
.10              then rt =  $\mu$  (let, fnrng  $\mapsto$  t1)
.11              else rt = let)
.12              $\vee$ 
.13             ( $\exists t_1, t_2 : TypeR$  .
.14               IsMapTypeR(let)(t1, t2)  $\wedge$ 
.15               IsSubtypeR( $\pi$ )(ret, NAT)(env)  $\wedge$ 
.16               (IsGTE2( $\pi'$ )(ret)(env)  $\Rightarrow$  IsSubtypeR( $\pi$ )(t2, t1)(env))  $\wedge$ 
.17               if IsZero( $\pi'$ )(ret)(env)
.18               then rt =  $\mu$  (let, maprng  $\mapsto$  t1)
.19               else rt = let)
.20              $\vee$ 
.21             cases  $\pi$  :
.22               mk-DEF(-)  $\rightarrow$  let = NATONE  $\wedge$  ret = REAL  $\wedge$  rt = REAL,
.23               mk-POS(-)  $\rightarrow$  let = REAL  $\wedge$  ret = REAL  $\wedge$  rt = REAL
.24             end
.25             in
.26     wf-SCompExpr( $\pi$ )(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 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 :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     IsGTE2 ( $\pi$ )(type)(env)  $\triangleq$ 
.2       is-POS( $\pi$ )  $\wedge$  type = NATONE

```

IsGTE2 checks that *type* represents natural numbers greater than or equal to 2.

```

252.0   IsZero :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     IsZero ( $\pi$ )(type)(env)  $\triangleq$ 
.2       is-POS( $\pi$ )  $\wedge$  type = NAT

```

11.3.9 Quantified Expressions

- 253.0 $wf\text{-}AllExpr : \Pi \xrightarrow{t} AllExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}AllExpr(\pi)(mk\text{-}AllExpr(bindl, pred))(type)(env) \triangleq$
 - .2 $type = \text{BOOLEAN} \wedge$
 - .3 $\exists venv : ValEnv, bindattrs : BindAttrs .$
 - .4 $wf\text{-}BindList(\pi)(bindl)(venv, bindattrs)(env) \wedge$
 - .5 $(\text{EMPTY} \in bind attrs) \vee$
 - .6 $\text{let } wfe = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
 - .7 $wfe(pred)(\text{BOOLEAN})(UpdateEnv}(venv)(env));$
- 254.0 $wf\text{-}ExistsExpr : \Pi \xrightarrow{t} ExistsExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}ExistsExpr(\pi)(mk\text{-}ExistsExpr(bindl, pred))(type)(env) \triangleq$
 - .2 $type = \text{BOOLEAN} \wedge$
 - .3 $\exists venv : ValEnv, bindattrs : BindAttrs .$
 - .4 $wf\text{-}BindList(\pi)(bindl)(venv, bindattrs)(env) \wedge$
 - .5 $(\text{EMPTY} \in bind attrs) \vee$
 - .6 $\text{let } wfe = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
 - .7 $wfe(pred)(\text{BOOLEAN})(UpdateEnv}(venv)(env));$
- 255.0 $wf\text{-}ExistsUniqueExpr : \Pi \xrightarrow{t} ExistsUniqueExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}ExistsUniqueExpr(\pi)(mk\text{-}ExistsUniqueExpr(bind, pred))(type)(env) \triangleq$
 - .2 $type = \text{BOOLEAN} \wedge$
 - .3 $\exists venv : ValEnv, bindattrs : BindAttrs .$
 - .4 $wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \wedge$
 - .5 $(\text{EMPTY} \in bind attrs) \vee$
 - .6 $\text{let } wfe = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
 - .7 $wfe(pred)(\text{BOOLEAN})(UpdateEnv}(venv)(env))$

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 *bind attrs* (line 255.4).

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.

11.3.10 Iota Expression

- 256.0 $wf\text{-}IotaExpr : \Pi \xrightarrow{t} IotaExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}IotaExpr(\pi)(mk\text{-}IotaExpr(bind, pred))(type)(env) \triangleq$
 - .2 $\exists venv : ValEnv, n : Name, bindattrs : BindAttrs .$
 - .3 $wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \wedge$
 - .4 $\text{let } wfe = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi)) \text{ in}$
 - .5 $wfe(pred)(\text{BOOLEAN})(UpdateEnv}(venv)(env)) \wedge$
 - .6 $UniqueVal(\pi)(mk\text{-}IotaExpr(bind, pred))(type)(env) \wedge$
 - .7 $\text{dom } venv = \{n\} \wedge$
 - .8 $type = venv(n).\text{type}$

Unlike the quantified expressions (formulae 253, 254, 255) the iota expression is not well-formed if the set of bindings is empty, so the *bind attrs* 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 :  $\Pi \xrightarrow{t} IotaExpr \rightarrow Type \rightarrow Env \rightarrow \mathbb{B}$ 
.1    UniqueVal( $\pi$ )(mk-IotaExpr(bind, pred))(type)(env)  $\triangleq$ 
.2    is-POS( $\pi$ )
```

11.3.11 Set Expressions

```
258.0  wf-SetEnumeration :  $\Pi \xrightarrow{t} SetEnumeration \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1    wf-SetEnumeration( $\pi$ )(mk-SetEnumeration(exprl))(type)(env)  $\triangleq$ 
.2    let CP : TypeR*  $\xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3      CP(ts)(rt)(env)  $\triangleq$ 
.4      ( $\forall t \in \text{elems } ts \cdot \text{IsFlatType}(\pi)(t)(env)) \wedge$ 
.5      rt = if ts = [] then EMPTYSET else mk-SetTypeR(mk-UnionTypeR(elems ts))
.6      in
.7    wf-SCompExpr( $\pi$ )(CP)(exprl)(type)(env);

259.0  wf-SetComprehension :  $\Pi \xrightarrow{t} SetComprehension \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1    wf-SetComprehension( $\pi$ )(mk-SetComprehension(elem, bindl, opred))(type)(env)  $\triangleq$ 
.2     $\exists venv : ValEnv, bindattrs, interpratrs : BindAttrs, mk-UnionTypeR(etset) : TypeR \cdot$ 
.3      wf-BindList( $\pi$ )(bindl)(venv, bindattrs)(env)
.4       $\wedge$ 
.5      let newenv : Env = UpdateEnv(venv)(env),
.6        wfe = InScope[Expr]( $\pi$ )(dom venv)(wf-Expr( $\pi$ )) in
.7        EMPTY  $\in$  bindattrs  $\wedge$  type = EMPTYSET
.8       $\vee$ 
.9      wf-PredInScope( $\pi$ )(opred)(newenv, interpratrs)(env)  $\wedge$ 
.10     (EMPTY  $\in$  interpratrs  $\wedge$  type = EMPTYSET
.11      $\vee$ 
.12     (FINITE  $\in$  bindattrs  $\vee$  FINITE  $\in$  interpratrs)  $\wedge$ 
.13     type = mk-SetTypeR(mk-UnionTypeR(etset))  $\wedge$ 
.14      $\forall elem \in etset \cdot$ 
.15       wfe(elem)(elem)(newenv)  $\wedge$ 
.16       IsFlatTypeR( $\pi$ )(elem)(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 *interpratrs*).

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 $(t_1 \sqcup t_2)$ -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).

```

260.0   wf-SetRange :  $\Pi \xrightarrow{t} SetRange \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-SetRange ( $\pi$ )( $mk\text{-}SetRange(lb, ub)$ )( $type$ )( $env$ )  $\triangleq$ 
.2       let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3          $CP([lb, ub])(rt)(env) \triangleq$ 
.4            $(\forall bt \in [lb, ub] \cdot IsSubtypeR(\pi)(bt, REAL)(env)) \wedge$ 
.5           let  $et = \text{if } IsSubtypeR(\pi)(lb, INT)(env) \text{ then } lbt \text{ else } INT$  in
.6              $rt = mk\text{-}SetTypeR(et)$ 
.7           in
.8     wf-SCompExpr( $\pi$ )( $CP$ )( $[lb, ub]$ )( $type$ )( $env$ )

```

11.3.12 Sequence Expressions

```

261.0   wf-SeqEnumeration :  $\Pi \xrightarrow{t} SeqEnumeration \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-SeqEnumeration ( $\pi$ )( $mk\text{-}SeqEnumeration(exprl)$ )( $type$ )( $env$ )  $\triangleq$ 
.2       let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3          $CP(ts)(rt)(env) \triangleq$ 
.4            $rt = \text{if } ts = []$ 
.5           then EMPTYSEQ
.6           else  $mk\text{-}Seq1TypeR(mk\text{-}UnionTypeR(\text{elems } ts))$ 
.7           in
.8     wf-SCompExpr( $\pi$ )( $CP$ )( $exprl$ )( $type$ )( $env$ );

262.0   wf-SeqComprehension :  $\Pi \xrightarrow{t} SeqComprehension \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-SeqComprehension ( $\pi$ )( $mk\text{-}SeqComprehension(elem, setbind, opred)$ )( $type$ )( $env$ )  $\triangleq$ 
.2        $\exists venv : ValEnv, bindattrs, interprattr : BindAttrs, mk\text{-}UnionTypeR(etset) : TypeR \cdot$ 
.3          $wf\text{-}SetBind(\pi)(setbind)(venv, bindattrs)(env) \wedge$ 
.4          $\exists n : Name \cdot \text{dom } venv = \{n\} \wedge IsSubtypeR(\pi)(venv(n).type, REAL)(env)$ 
.5          $\wedge$ 
.6         let  $newenv : Env = UpdateEnv(venv)(env)$ ,
.7            $wfe = InScope[Expr](\pi)(\text{dom } venv)(wf-Expr(\pi))$  in
.8            $\text{EMPTY} \in bindattrs \wedge type = EMPTYSEQ$ 
.9            $\vee$ 
.10           $wf\text{-}PredInScope(\pi)(opred)(venv, interprattr)(env) \wedge$ 
.11           $(\text{EMPTY} \in interprattr \wedge type = EMPTYSEQ)$ 
.12           $\vee$ 
.13           $(\text{FINITE} \in bindattrs \vee \text{FINITE} \in interprattr) \wedge$ 
.14           $type = mk\text{-}Seq0TypeR(mk\text{-}UnionTypeR(etset)) \wedge$ 
.15           $\forall elem \in etset \cdot wfe(elem)(newenv)$ 

```

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).

263.0 $wf\text{-}SubSequence : \Pi \xrightarrow{t} SubSequence \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}SubSequence(\pi)(mk\text{-}SubSequence(seqe, frome, toe))(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP([seqt, fromt, tot])(rt)(env) \triangleq$
 .4 $(\forall t \in \{fromt, tot\} \cdot t = \text{REAL}) \wedge$
 .5 $\exists et : TypeR \cdot IsSeqTypeR(seqt)(et) \wedge$
 .6 $rt = mk\text{-Seq0}\ TypeR(seqt.elemtp)$
 .7 in
 .8 $wf\text{-}SCompExpr(\pi)(CP)([seqe, frome, toe])(type)(env)$

11.3.13 MapExpressions

264.0 $wf\text{-}MapEnumeration : \Pi \xrightarrow{t} MapEnumeration \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}MapEnumeration(\pi)(mk\text{-}MapEnumeration(maplets))(type)(env) \triangleq$
 .2 let $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP(pts)(rt)(env) \triangleq$
 .4 let $mdts = \{\text{let } mk\text{-ProductTypeR}([dt, -]) = pts(i) \text{ in } dt \mid i \in \text{inds } pts\}$,
 .5 $mrts = \{\text{let } mk\text{-ProductTypeR}([-, rt]) = pts(i) \text{ in } rt \mid i \in \text{inds } pts\}$ in
 .6 $(\forall dt \in mdts \cdot IsFlatTypeR(\pi)(dt)(env)) \wedge$
 .7 $\forall i_1, i_2 \in \text{inds } pts \cdot$
 .8 let $mk\text{-ProductTypeR}([dt1, rt1]) = pts(i_1)$,
 .9 $mk\text{-ProductTypeR}([dt2, rt2]) = pts(i_2)$ in
 .10 $i_1 \neq i_2 \wedge$
 .11 $SameValue(NegWfClass(\pi))(dt1, dt2)(env) \Rightarrow SameValue(\pi)(rt1, rt2)(env)$
 .12 \wedge
 .13 $rt =$
 .14 if $pts = []$ then EMPTYMAP
 .15 else $mk\text{-GeneralMapTypeR}(mk\text{-UnionTypeR}(mdts), mk\text{-UnionTypeR}(mrt))$
 .16 in
 .17 let $tuplseq : TupleConstructor^* =$
 .18 [let $mk\text{-Maplet}(de, re) = maplets(i)$ in
 .19 $mk\text{-TupleConstructor}([de, re]) \mid i \in \text{inds } maplets$] in
 .20 $wf\text{-}SCompExpr(\pi)(CP)(tuplseq)(type)(env)$

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 $wf\text{-}SCompExpr$ 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 $wf\text{-}SCompExpr$. 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.

265.0 $wf\text{-}MapComprehension : \Pi \xrightarrow{t} MapComprehension \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}MapComprehension(\pi)(mk\text{-}MapComprehension(ml, bindl, opred))(type)(env) \triangleq$
 .2 $\text{let } mk\text{-}Maplet(de, re) = ml \text{ in}$
 .3 $\exists venv : ValEnv, bindattrs, interprattr : BindAttrs .$
 .4 $wf\text{-}BindList(\pi)(bindl)(venv, bindattrs)(env) \wedge$
 .5 $\text{let newenv} : Env = UpdateEnv(venv)(env),$
 .6 $wfe = InScope[Expr](\pi)(\text{dom } venv)(wf\text{-}Expr(\pi)) \text{ in}$
 .7 $\text{EMPTY} \in bindattrs \wedge type = \text{EMPTYSMAP}$
 .8 \vee
 .9 $wf\text{-}PredInScope(\pi)(opred)(venv, interprattr)(env)$
 .10 \wedge
 .11 $(\text{EMPTY} \in interprattr \wedge type = \text{EMPTYSMAP}$
 .12 \vee
 .13 $(\text{FINITE} \in bindattrs \vee \text{FINITE} \in interprattr) \wedge$
 .14 $\exists mk\text{-}UnionTypeR(dtset), mk\text{-}UnionTypeR(rtset) : TypeR .$
 .15 $type = mk\text{-}GeneralMapTypeR(mk\text{-}UnionTypeR(dtset), mk\text{-}UnionTypeR(rtset)) \wedge$
 .16 $\forall dt \in dtset \cdot wfe(de)(dt)(newenv) \wedge IsFlatTypeR(\pi)(dt)(env) \wedge$
 .17 $\forall rt \in rtset \cdot wfe(re)(rt)(newenv) \wedge$
 .18 $\text{let } mc = mk\text{-}MapComprehension(ml, bindl, opred) \text{ in}$
 .19 $CompatibleMapLetBindings(\pi)(mc)(type)(env);$

266.0 $CompatibleMapLetBindings : \Pi \xrightarrow{t} MapComprehension \rightarrow Type \rightarrow Env \rightarrow \mathbb{B}$
 .1 $CompatibleMapLetBindings(\pi)(mk\text{-}MapComprehension(ml, bindl, opred))(-)(-) \triangleq$
 .2 $is\text{-}POS(\pi)$

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 $CompatibleMapLetBindings$.

11.3.14 Tuple Constructor

267.0 $wf\text{-}TupleConstructor : \Pi \xrightarrow{t} TupleConstructor \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}TupleConstructor(\pi)(mk\text{-}TupleConstructor(fields))(type)(env) \triangleq$
 .2 $\text{let } CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $CP(fieldtypes)(rt)(env) \triangleq$
 .4 $rt = mk\text{-}ProductTypeR(fieldtypes)$
 .5 in
 .6 $wf\text{-}SCompExpr(\pi)(CP)(fields)(type)(env)$

11.3.15 Record Expressions

```

268.0   wf-RecordConstructor :  $\Pi \xrightarrow{t} RecordConstructor \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-RecordConstructor ( $\pi$ )(mk-RecordConstructor( $tag, fields$ ))( $type$ )( $env$ )  $\triangleq$ 
.2     let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3        $CP(ftypes(rt))(env) \triangleq$ 
.4        $\exists t : TypeR .$ 
.5         IsTypeRwithTagInEnv( $t, tag$ )( $env$ )  $\wedge$  IsSubtypeR( $\pi$ )( $rt, t$ )( $env$ )  $\wedge$ 
.6         cases  $rt$  :
.7           mk-CompositeTypeR( $-$ ,  $fieldRs$ )  $\rightarrow$ 
.8             len  $fieldRs = len ftypes \wedge$ 
.9              $\forall i \in \text{inds } fieldRs . ftypes(i) = fieldRs(i).type,$ 
.10            others  $\rightarrow$  false
.11          end
.12        in
.13      wf-SCompExpr( $\pi$ )( $CP$ )( $fields$ )( $type$ )( $env$ )

```

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 :  $\Pi \xrightarrow{t} RecordModifier \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-RecordModifier ( $\pi$ )(mk-RecordModifier( $rec, modfs$ ))( $type$ )( $env$ )  $\triangleq$ 
.2     let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3        $CP([rec] \curvearrowright newfldts)(rt)(env) \triangleq$ 
.4       cases  $rect$  :
.5         mk-CompositeTypeR( $tag, fields$ )  $\rightarrow$ 
.6           let  $newfieldmap =$ 
.7              $\{modfs(i).field \mapsto newfldts(i) \mid i \in \text{inds } modfs\},$ 
.8              $newfields =$ 
.9               [if  $fields(i).id \in \text{dom } newfieldmap$ 
.10                 then mk-FieldR( $fields(i).id, newfieldmap(fields(i).id)$ )
.11                 else  $fields(i)$  |
.12                    $i \in \text{inds } fields$ ] in
.13              $rt = mk-CompositeTypeR(tag, newfields) \wedge$ 
.14              $\exists definedt : TypeR .$ 
.15               IsTypeRwithTagInEnv( $definedt, tag$ )( $env$ )  $\wedge$ 
.16               IsSubtypeR( $\pi$ )( $rt, definedt$ )( $env$ ),
.17               others  $\rightarrow$  false
.18             end
.19           in
.20         wf-SCompExpr( $\pi$ )( $CP$ )
.21         ([ $rec$ ]  $\curvearrowright [modfs(i).new \mid i \in \text{inds } modfs]$ )( $type$ )( $env$ )

```

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 record value to be modified might already be such a subtype, so the originally defined composite type corresponding to *tag* must be looked up.

11.3.16 Apply Expressions

```

270.0   wf-Apply :  $\Pi \xrightarrow{t} Apply \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-Apply ( $\pi$ )( $mk\text{-}Apply(fnc, args)$ )( $type$ )( $env$ )  $\triangleq$ 
.2       let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3          $CP ([fnc, argt])(rt)(env) \triangleq$ 
.4            $IsApplicationType(fnc)(argt, rt) \wedge$ 
.5           ( $is\text{-}POS(\pi) \vee is\text{-}TotalFnTypeR(fnc)$ )
.6         in
.7        $wf\text{-}SCompExpr(\pi)(CP)([fnc, Exprs2Expr(args)])(type)(env);$ 

271.0   IsApplicationType :  $TypeR \xrightarrow{t} TypeR \times TypeR \rightarrow \mathbb{B}$ 
.1     IsApplicationType ( $t$ )( $at, rt$ )  $\triangleq$ 
.2       cases  $t$  :
.3          $mk\text{-}PartialFnTypeR(at, rt), mk\text{-}TotalFnTypeR(at, rt) \rightarrow true,$ 
.4          $mk\text{-}GeneralMapTypeR(at, rt), mk\text{-}InjectiveMapTypeR(at, rt) \rightarrow true,$ 
.5          $mk\text{-}Seq1TypeR(rt) \rightarrow at = NATONE,$ 
.6         others  $\rightarrow false$ 
.7       end;

272.0   wf-FieldSelect :  $\Pi \xrightarrow{t} FieldSelect \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-FieldSelect ( $\pi$ )( $mk\text{-}FieldSelect(record, field)$ )( $type$ )( $env$ )  $\triangleq$ 
.2       let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3          $CP ([rect])(rt)(env) \triangleq$ 
.4         cases  $rect$  :
.5            $mk\text{-}CompositeTypeR(~, fields) \rightarrow$ 
.6              $\exists mk\text{-}FieldR(id, ft) \in \text{elems } fields \cdot rt = ft \wedge field = id,$ 
.7             others  $\rightarrow false$ 
.8         end
.9       in
.10       $wf\text{-}SCompExpr(\pi)(CP)([record])(type)(env);$ 

273.0   wf-FctTypeInst :  $\Pi \xrightarrow{t} FctTypeInst \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-FctTypeInst ( $\pi$ )( $mk\text{-}FctTypeInst(polyfctname, inst)$ )( $type$ )( $env$ )  $\triangleq$ 
.2        $\exists mk\text{-}PolyValR(tpars, valr) : PolyValR .$ 
.3          $IsBoundTo(polyfctname)(mk\text{-}PolyValR(tpars, valr))(env) \wedge$ 
.4          $\text{len } inst = \text{len } tpars \wedge (is\text{-}DEF(\pi) \Rightarrow is\text{-}DEF(valr.wfclass)) \wedge$ 
.5          $\exists ts : TypeR^* .$ 
.6            $\text{len } ts = \text{len } tpars \wedge$ 
.7            $\forall i \in \text{inds } inst \cdot wf\text{-}Type(\pi)(inst(i))(ts(i))(env) \wedge$ 
.8            $type = TypeSubst(\{tpars(i) \mapsto ts(i) \mid i \in \text{inds } tpars\})(valr.type)$ 

```

11.3.17 Lambda Expressions

274.0 $wf\text{-}Lambda : \Pi \xrightarrow{t} Lambda \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}Lambda(\pi)(mk\text{-}Lambda(tbl, body_1))(type)(env) \triangleq$
- .2 $\text{let } t = Types2Type([tbl(i).type \mid i \in \text{inds } tbl]),$
- .3 $ap = Pats2Pat([tbl(i).pat \mid i \in \text{inds } tbl]) \text{ in }$
- .4 $\exists id : Id, body_2 : Expr, at, et, mk\text{-}UnionTypeR(rtset) : TypeR .$
- .5 $IsLetExpansion(body_2)(ap, \text{nil}, mk\text{-}Name(id), body_1)(env) \wedge$
- .6 $wf\text{-}Type(\pi)(t)(at)(env) \wedge IsEssEquivTypeR(\pi)(et, at)(env) \wedge$
- .7 $\forall rt \in rtset .$
- .8 $wf\text{-}LambdaR(\pi)(mk\text{-}LambdaR(id, at, body_2, \text{nil}, \text{false}))(mk\text{-}PartialFnTypeR(at, rt))(env)$

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 $at \rightarrow t_1$, $at \rightarrow t_2$ and $at \rightarrow t_1 \mid t_2$. 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 $wf\text{-}LambdaR : \Pi \xrightarrow{t} LambdaR \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}LambdaR(\pi)(mk\text{-}LambdaR(id, at, body, ope, assertedtotal))(fnt)(env) \triangleq$
- .2 $\text{let } venv = \{mk\text{-}Name(id) \mapsto mk\text{-}ValR(at, \pi)\},$
- .3 $lenv = UpdateEnv(venv)(env) \text{ in }$
- .4 $\exists verifiedtotal : \mathbb{B} .$
- .5 $Is[FunctionTypeR, TypeR](fnt) \wedge \neg IsEmptyType(NegWfClass(\pi))(fnt)(env) \wedge$
- .6 $fnt.fndom = at$
- .7 \wedge
- .8 $(wf\text{-}FnBody(\pi)(body, ope, verifiedtotal, dom venv)(fnt.fnrng)(lenv) \vee$
- .9 $IsEmptyType(\pi)(at)(env))$
- .10 \wedge
- .11 $(assertedtotal \Rightarrow verifiedtotal) \wedge$
- .12 $(is\text{-}TotalFnTypeR(fnt) \Rightarrow verifiedtotal \wedge$
- .13 $(PreAlwaysTrue(\pi)(ope)(lenv) \vee IsEmptyType(\pi)(at)(env)))$

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.

```

276.0   wf-FnBody :  $\Pi \xrightarrow{t} Expr \times [Expr] \times \mathbb{B} \times Name\text{-set} \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-FnBody ( $\pi$ ) (body, opre, verifiedtotal, pars) (restp) (env)  $\triangleq$ 
.2     let wfpred = if verifiedtotal
.3       then  $\lambda p : \Pi \cdot InTotalFnScope(p)(pars)(wf-Expr(p))$ 
.4       else  $\lambda p : \Pi \cdot InScope[Expr](p)(pars)(wf-Expr(p))$  in
.5     PreNeverTrue( $\pi$ ) (opre) (env)  $\vee$ 
.6     cases  $\pi$  :
.7       mk-DEF(cs)  $\rightarrow$ 
.8         wfpred( $\pi$ ) (body) (restp) (env)  $\vee$ 
.9          $\neg$  verifiedtotal  $\wedge$ 
.10         $\neg \exists t : TypeR \cdot wfpred(mk-POS(cs))(body)(t)(env)$ ,
.11        mk-POS(cs)  $\rightarrow$ 
.12        PreAlwaysTrue(mk-DEF(cs)) (opre) (env)  $\Rightarrow$ 
.13        if verifiedtotal
.14        then  $\exists rtset : TypeR\text{-set}$  .
.15          restp = mk-UnionTypeR(rtset)  $\wedge$ 
.16           $\forall rt \in rtset$  .
.17          wfpred( $\pi$ ) (body) (rt) (env)
.18        else  $\neg \exists t : TypeR$  .
.19          wfpred(mk-DEF(cs)) (body) (t) (env)  $\wedge$ 
.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 $at \rightarrow t_1$, $at \rightarrow t_2$ and $at \rightarrow t_1 \mid t_2$. 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).

11.3.18 Is Expressions

```

277.0   wf-IsDefTypeExpr :  $\Pi \xrightarrow{t} IsDefTypeExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   wf-IsDefTypeExpr ( $\pi$ )( $mk\text{-}IsDefTypeExpr(tag, arg)$ )( $type$ )( $env$ )  $\triangleq$ 
.2   let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3    $CP([argt])(rt)(env) \triangleq$ 
.4    $rt = \text{BOOLEAN}$ 
.5   in
.6    $(\exists dt : TypeR \cdot IsTypeRwithTagInEnv(dt, tag)(env)) \wedge$ 
.7   wf-SCompExpr( $\pi$ )( $CP$ )([ $arg$ ])( $type$ )( $env$ );

278.0   wf-IsBasicTypeExpr :  $\Pi \xrightarrow{t} IsBasicTypeExpr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   wf-IsBasicTypeExpr ( $\pi$ )( $mk\text{-}IsBasicTypeExpr(type, arg)$ )( $type$ )( $env$ )  $\triangleq$ 
.2   let  $CP : TypeR^* \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3    $CP([argt])(rt)(env) \triangleq$ 
.4    $rt = \text{BOOLEAN}$ 
.5   in
.6   wf-SCompExpr( $\pi$ )( $CP$ )([ $arg$ ])( $type$ )( $env$ )

```

11.3.19 Names

```

279.0   wf-Name :  $\Pi \xrightarrow{t} Name \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   wf-Name ( $\pi$ )( $name$ )( $type$ )( $env$ )  $\triangleq$ 
.2   IsBoundToValTypeR( $\pi$ )( $name$ )( $type$ )( $env$ );

280.0   wf-OldName :  $\Pi \xrightarrow{t} OldName \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   wf-OldName ( $\pi$ )( $oldname$ )( $type$ )( $env$ )  $\triangleq$ 
.2   IsBoundToValTypeR( $\pi$ )( $oldname$ )( $type$ )( $env$ )

```

11.3.20 Literals

```

281.0   wf-Literal :  $\Pi \xrightarrow{t} Literal \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   wf-Literal ( $\pi$ )( $lit$ )( $type$ )( $env$ )  $\triangleq$ 
.2   cases  $lit$  :
.3    $mk\text{-}NumLit(-) \rightarrow wf\text{-}NumLit(\pi)(lit)(type)(env),$ 
.4    $mk\text{-}BoolLit(-) \rightarrow type = \text{BOOLEAN},$ 
.5    $mk\text{-}NilLit() \rightarrow type = \text{UNITTYPE},$ 
.6    $mk\text{-}CharLit(-) \rightarrow type = \text{CHAR},$ 
.7    $mk\text{-}TextLit(-) \rightarrow wf\text{-}TextLit(\pi)(lit)(type)(env),$ 
.8    $mk\text{-}QuoteLit(-) \rightarrow type = mk\text{-}QuoteTypeR(lit)$ 
.9   end;

282.0   wf-NumLit :  $\Pi \xrightarrow{t} NumLit \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   wf-NumLit ( $\pi$ )( $mk\text{-}NumLit(z)$ )( $type$ )( $env$ )  $\triangleq$ 
.2    $type =$ 
.3   if is-N1( $z$ ) then NATONE
.4   elseif is-N( $z$ ) then NAT
.5   elseif is-Z( $z$ ) then INT else RAT;

```

283.0 $wf\text{-}TextLit : \Pi \xrightarrow{t} TextLit \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .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(CHAR)

11.3.21 Auxiliary Well-formedness Predicates

284.0 $wf\text{-}PredInScope : \Pi \xrightarrow{t} [Expr] \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}PredInScope(\pi)(opred)(venv, bindattrs)(env) \triangleq$
 .2 $opred = \text{nil} \wedge bindattrs = \{\}$
 .3 \vee
 .4 let $wfp = (InScope[Expr](\pi)(\text{dom } venv)) \circ (InType[Expr](\pi))(wf\text{-}Expr(\pi))$ in
 .5 $wfp(opred)(\text{BOOLEAN})(UpdateEnv(venv)(env)) \wedge$
 .6 cases π :
 .7 $mk\text{-}POS(-) \rightarrow bindattrs = \{\text{EMPTY}, \text{FINITE}\},$
 .8 $mk\text{-}DEF(-) \rightarrow bindattrs = \{\}$
 .9 end

$wf\text{-}PredInScope$ 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 $is\text{-}DEF}(\pi)$ 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 $is\text{-}POS}(\pi)$ if this set of dynamic environments is empty or finite then EMPTY or FINITE respectively will be in $bindattrs$.

11.4 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 $wf\text{-}StateDesignator : \Pi \xrightarrow{t} StateDesignator \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}StateDesignator(\pi)(sd)(t)(env) \triangleq$
 .2 cases sd :
 .3 $mk\text{-}Name(-) \rightarrow \exists sd_t : TypeR .$
 .4 $IsBoundTo(sd)(mk\text{-}VarR(sd_t))(env) \wedge$
 .5 cases π :
 .6 $mk\text{-}DEF(-) \rightarrow sd_t = t,$
 .7 $mk\text{-}POS(-) \rightarrow IsSubtype(\pi)(t, sd_t)(env)$
 .8 end,
 .9 $mk\text{-}FieldRef(-, -) \rightarrow wf\text{-}FieldRef(\pi)(sd)(t)(env),$
 .10 $mk\text{-}MapOrSeqRef(-, -) \rightarrow wf\text{-}MapOrSeqRef(\pi)(sd)(t)(env)$
 .11 end;

```

286.0   wf-FieldRef :  $\Pi \xrightarrow{t} FieldRef \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-FieldRef ( $\pi$ )( $mk\text{-}FieldRef(var, sel)$ )( $t$ )( $env$ )  $\triangleq$ 
.2        $\exists var_t : TypeR .$ 
.3         wf-StateDesignator( $\pi$ )( $var$ )( $var_t$ )( $env$ )  $\wedge$ 
.4          $\forall var_{t'} : TypeR .$ 
.5           IsUnionComponent( $var_{t'}, var_t$ )( $env$ )  $\Rightarrow$ 
.6           is-CompositeTypeR( $var_{t'}$ )  $\wedge$  mk-FieldR( $sel, t$ )  $\in$  elems  $var_{t'}.fields$ ;

287.0   wf-MapOrSeqRef :  $\Pi \xrightarrow{t} MapOrSeqRef \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-MapOrSeqRef ( $\pi$ )( $mk\text{-}MapOrSeqRef(var, arg)$ )( $t$ )( $env$ )  $\triangleq$ 
.2       cases  $\pi$  :
.3         mk-DEF(-)  $\rightarrow$  false,
.4         mk-POS(-)  $\rightarrow$ 
.5            $\exists var_t, arg_t : TypeR .$ 
.6             wf-StateDesignator( $\pi$ )( $var$ )( $var_t$ )( $env$ )  $\wedge$ 
.7               wf-Expr( $\pi$ )( $arg$ )( $arg_t$ )( $env$ )  $\wedge$ 
.8                 cases  $var_t$  :
.9                   mk-Seq1TypeR( $et$ )  $\rightarrow$ 
.10                     IsSubtype( $\pi$ )( $arg_t$ , NATONE)( $env$ )  $\wedge$  Subsume( $\pi$ )( $et, t$ )( $env$ ),
.11                     mk-GeneralMapTypeR( $mapdom, maprng$ )  $\rightarrow$ 
.12                       IsSubtype( $\pi$ )( $arg_t, mapdom$ )( $env$ )  $\wedge$  Subsume( $\pi$ )( $maprng, t$ )( $env$ ),
.13                     mk-InjectiveMapTypeR( $mapdom, maprng$ )  $\rightarrow$ 
.14                       IsSubtype( $\pi$ )( $arg_t, mapdom$ )( $env$ )  $\wedge$  Subsume( $\pi$ )( $maprng, t$ )( $env$ ),
.15                     others  $\rightarrow$  false
.16               end
.17         end

```

11.5 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 possibly 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 $wf\text{-}Stmt : \Pi \xrightarrow{t} Stmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}Stmt(\pi)(s)(t)(env) \triangleq$
- .2 cases $s :$
- .3 $mk\text{-}LetStmt(-, -) \rightarrow wf\text{-}LetStmt(\pi)(s)(t)(env),$
- .4 $mk\text{-}LetBeSTStmt(-, -, -) \rightarrow wf\text{-}LetBeSTStmt(\pi)(s)(t)(env),$
- .5 $mk\text{-}DefStmt(-, -) \rightarrow wf\text{-}DefStmt(\pi)(s)(t)(env),$
- .6 $mk\text{-}BlockStmt(-, -) \rightarrow wf\text{-}BlockStmt(\pi)(s)(t)(env),$
- .7 $mk\text{-}AssignStmt(-, -) \rightarrow wf\text{-}AssignStmt(\pi)(s)(t)(env),$
- .8 $mk\text{-}IfStmt(-, -, -, -) \rightarrow wf\text{-}IfStmt(\pi)(s)(t)(env),$
- .9 $mk\text{-}CasesStmt(-, -, -) \rightarrow wf\text{-}CasesStmt(\pi)(s)(t)(env),$
- .10 $mk\text{-}SeqForLoop(-, -, -, -) \rightarrow wf\text{-}SeqForLoop(\pi)(s)(t)(env),$
- .11 $mk\text{-}SetForLoop(-, -, -) \rightarrow wf\text{-}SetForLoop(\pi)(s)(t)(env),$
- .12 $mk\text{-}IndexForLoop(-, -, -, -, -) \rightarrow wf\text{-}IndexForLoop(\pi)(s)(t)(env),$
- .13 $mk\text{-}WhileLoop(-, -) \rightarrow wf\text{-}WhileLoop(\pi)(s)(t)(env),$
- .14 $mk\text{-}NonDetStmt(-) \rightarrow wf\text{-}NonDetStmt(\pi)(s)(t)(env),$
- .15 $mk\text{-}Call(-, -, -) \rightarrow wf\text{-}Call(\pi)(s)(t)(env),$
- .16 $mk\text{-}ReturnStmt(-) \rightarrow wf\text{-}ReturnStmt(\pi)(s)(t)(env),$
- .17 $mk\text{-}AlwaysStmt(-, -) \rightarrow wf\text{-}AlwaysStmt(\pi)(s)(t)(env),$
- .18 $mk\text{-}TrapStmt(-, -, -) \rightarrow wf\text{-}TrapStmt(\pi)(s)(t)(env),$
- .19 $mk\text{-}RecTrapStmt(-, -) \rightarrow wf\text{-}RecTrapStmt(\pi)(s)(t)(env),$
- .20 $mk\text{-}ExitStmt(-) \rightarrow wf\text{-}ExitStmt(\pi)(s)(t)(env),$
- .21 $(SKIP) \rightarrow wf\text{-}IdentStmt(\pi)(s)(t)(env)$
- .22 end

11.5.1 Local Binding Statements

289.0 $wf\text{-}LetStmt : \Pi \xrightarrow{t} LetStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}LetStmt(\pi)(mk\text{-}LetStmt(letdefs, body))(t)(env) \triangleq$
- .2 $\exists venv : VisibleEnv .$
- .3 $wf\text{-}ValFnOpDefs(\pi)(letdefs)(venv)(env) \wedge$
- .4 $InScope[Stmt](\pi)(dom venv)(wf\text{-}Stmt(\pi))(body)(t)(UpdateEnv(venv)(env));$

290.0 $wf\text{-}LetBeSTStmt : \Pi \xrightarrow{t} LetBeSTStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}LetBeSTStmt(\pi)(mk\text{-}LetBeSTStmt(bind, cond, body))(t)(env) \triangleq$
- .2 $(\exists venv : ValEnv, bindattrs : BindAttrs .$
- .3 $wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \wedge$
- .4 $\text{let } newenv = UpdateEnv(venv)(env),$
- .5 $wfs = (InScope[Stmt](\pi)(dom venv)) \circ (InType[Stmt](\pi))(wf\text{-}Stmt(\pi)) \text{ in }$
- .6 $(cond \neq \text{nil} \Rightarrow$
- .7 $\exists truetp : StmtTypeR .$
- .8 $wfs(cond)(truetp)(newenv) \wedge IsTrueType(\pi)(truetp)(env)) \wedge$
- .9 $InScope[Stmt](\pi)(dom venv)(wf\text{-}Stmt(\pi))(body)(t)(newenv)) \wedge$
- .10 $\neg \exists venv : ValEnv, bindattrs : BindAttrs .$
- .11 $wf\text{-}Bind(NegWfClass(\pi))(bind)(venv, bindattrs)(env) \wedge$
- .12 $\text{EMPTY} \in bindattrs;$

291.0 $wf\text{-}DefStmt : \Pi \xrightarrow{t} DefStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}DefStmt(\pi)(mk\text{-}DefStmt(eqdefs, body))(t)(env) \triangleq$
- .2 $\exists venv : VisibleEnv, eqdef_t : StmtTypeR .$
- .3 $wf\text{-}EqDef(\pi)(hd eqdefs)(venv, eqdef_t)(env) \wedge$
- .4 $\forall eqdef_{t'} : StmtTypeR .$
- .5 $IsUnionComponent(eqdef_{t'}, eqdef_t)(env) \Rightarrow$
- .6 $\text{cases } eqdef_{t'} :$
- .7 $(\text{CONT}) \rightarrow \text{let } stmt = \text{if len eqdefs} = 1$
- .8 $\text{then } body$
- .9 $\text{else } mk\text{-}DefStmt(\text{tl eqdefs}, body),$
- .10 $wfs = InScope[Stmt](\pi)(\text{dom } venv)(wf\text{-}Stmt(\pi)) \text{ in}$
- .11 $wfs(stmt)(t)(UpdateEnv(venv)(env)),$
- .12 $\text{others } \rightarrow Subsume(\pi)(eqdef_{t'}, t)(env)$
- .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 $wf\text{-}EqDef : \Pi \xrightarrow{t} EqDef \rightarrow VisibleEnv \times StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}EqDef(\pi)(mk\text{-}EqDef(lhs, rhs))(venv, t)(env) \triangleq$
- .2 $\exists rhs_t : TypeR .$
- .3 $wf\text{-}CallOrExpr(\pi)(rhs)(rhs_t)(env) \wedge$
- .4 $\forall rhs_{t'} : TypeR .$
- .5 $IsUnionComponent(rhs_{t'}, rhs_t)(env) \wedge$
- .6 $\text{cases } rhs_{t'} :$
- .7 $(\text{CONT}) \rightarrow \text{false},$
- .8 $mk\text{-}RetTypeR((VOID)) \rightarrow \text{false},$
- .9 $mk\text{-}ExitTypeR(-) \rightarrow Subsume(\pi)(rhs_{t'}, t)(env),$
- .10 $mk\text{-}RetTypeR(rt) \rightarrow wf\text{-}Pattern(\pi)(lhs)(mk\text{-}TVE(rt, venv))(env) \wedge$
- .11 $Subsume(\pi)(\text{CONT}, t)(env),$
- .12 $\text{others } \rightarrow wf\text{-}Pattern(\pi)(lhs)(mk\text{-}TVE(rhs_{t'}, venv))(env) \wedge$
- .13 $Subsume(\pi)(\text{CONT}, t)(env)$
- .14 end

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).

11.5.2 Block and Assignment Statements

293.0 $wf\text{-}BlockStmt : \Pi \xrightarrow{t} BlockStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}BlockStmt(\pi)(mk\text{-}BlockStmt(dcls, stmts))(t)(env) \triangleq$
- .2 $\text{if } dcls = []$
- .3 $\text{then } wf\text{-}Sequence(\pi)(stmts)(t)(env)$
- .4 $\text{else } \exists venv : VisibleEnv, dcl_t : StmtTypeR .$
- .5 $wf\text{-}AssignDef(\pi)(hd dcls)(venv, dcl_t)(env) \wedge$
- .6 $\forall dcl_{t'} : StmtTypeR .$
- .7 $IsUnionComponent(dcl_{t'}, dcl_t)(env) \Rightarrow$
- .8 $\text{cases } dcl_{t'} :$
- .9 $(\text{CONT}) \rightarrow \text{let } stmt = mk\text{-}BlockStmt(\text{tl } dcls, stmts) \text{ in}$
- .10 $wf\text{-}Stmt(\pi)(stmt)(t)(UpdateEnv(venv)(env)),$
- .11 $\text{others } \rightarrow Subsume(\pi)(dcl_{t'}, t)(env)$
- .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 $wf\text{-Sequence} : \Pi \xrightarrow{t} Stmt^* \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-Sequence}(\pi)(stmts)(t)(env) \triangleq$
- .2 $\text{if } stmts = []$
- .3 $\text{then } Subsume(\pi)(CONT, t)(env)$
- .4 $\text{else } \exists stmt_t : StmtTypeR .$
- .5 $wf\text{-Stmt}(\pi)(hd\ stmts)(stmt_t)(env) \wedge$
- .6 $\forall stmt_{t'} : StmtTypeR .$
- .7 $IsUnionComponent(stmt_{t'}, stmt_t)(env) \Rightarrow$
- .8 $\text{cases } stmt_{t'} :$
- .9 $(CONT) \rightarrow wf\text{-Sequence}(\pi)(tl\ stmts)(t)(env),$
- .10 $\text{others } \rightarrow Subsume(\pi)(stmt_{t'}, t)(env)$
- .11 end;

295.0 $wf\text{-AssignDef} : \Pi \xrightarrow{t} AssignDef \rightarrow VisibleEnv \times StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-AssignDef}(\pi)(mk\text{-AssignDef}(var, tp, dclinit))(venv, t)(env) \triangleq$
- .2 $\exists tp_t, et : TypeR .$
- .3 $wf\text{-Type}(\pi)(tp)(tp_t)(env) \wedge$
- .4 $IsNonEmptyEssEquivTypeR(\pi)(et, tp_t)(env) \wedge$
- .5 $venv = \{ mk\text{-Name}(var) \mapsto mk\text{-VarR}(et) \} \wedge$
- .6 $\text{if } dclinit \neq \text{nil}$
- .7 $\text{then } \exists dclinit_t : TypeR .$
- .8 $wf\text{-CallOrExpr}(\pi)(dclinit)(dclinit_t)(env) \wedge$
- .9 $\forall dclinit_{t'} : TypeR .$
- .10 $IsUnionComponent(dclinit_{t'}, dclinit_t)(env) \Rightarrow$
- .11 $\text{cases } dclinit_{t'} :$
- .12 $(CONT) \rightarrow \text{false},$
- .13 $mk\text{-RetTypeR}(\text{VOID}) \rightarrow \text{false},$
- .14 $mk\text{-ExitTypeR}(-) \rightarrow Subsume(\pi)(dclinit_{t'}, t)(env),$
- .15 $mk\text{-RetTypeR}(rt) \rightarrow IsSubtypeR(\pi)(rt, tp_t)(env) \wedge$
- .16 $Subsume(\pi)(CONT, t)(env),$
- .17 $\text{others } \rightarrow IsSubtypeR(\pi)(dclinit_{t'}, tp_t)(env) \wedge$
- .18 $Subsume(\pi)(CONT, t)(env)$
- .19 end
- .20 $\text{else } Subsume(\pi)(CONT, t)(env)$

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 $wf\text{-CallOrExpr} : \Pi \xrightarrow{t} Call \mid Expr \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-CallOrExpr}(\pi)(c)(t)(env) \triangleq$
- .2 $\text{if } is\text{-Call}(c) \text{ then } wf\text{-Call}(\pi)(c)(t)(env)$
- .3 $\text{else } wf\text{-Expr}(\pi)(c)(t)(env);$

297.0 $wf\text{-}AssignStmt : \Pi \xrightarrow{t} AssignStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}AssignStmt(\pi)(mk\text{-}AssignStmt(lhs, rhs))(t)(env) \triangleq$
 .2 $\exists lhs_t, rhs_t : TypeR .$
 .3 $wf\text{-}StateDesignator(\pi)(lhs)(lhs_t)(env) \wedge$
 .4 $wf\text{-}CallOrExpr(\pi)(rhs)(rhs_t)(env) \wedge$
 .5 $\forall rhs_{t'} : TypeR .$
 .6 $IsUnionComponent(rhs_{t'}, rhs_t)(env) \wedge$
 .7 cases $rhs_{t'} :$
 .8 (CONT) $\rightarrow \text{false},$
 .9 $mk\text{-}RetTypeR((\text{VOID})) \text{ false},$
 .10 $mk\text{-}ExitTypeR(-) \rightarrow Subsume(\pi)(rhs_{t'}, t)(env),$
 .11 $mk\text{-}RetTypeR(rt) \rightarrow IsSubtype(\pi)(rt)(lhs_t)(env) \wedge$
 .12 $Subsume(\pi)(\text{CONT}, t)(env),$
 .13 others $\rightarrow IsSubtype(\pi)(rhs_t)(lhs_t)(env) \wedge$
 .14 $Subsume(\pi)(\text{CONT}, t)(env)$
 .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).

11.5.3 Conditional Statements

298.0 $wf\text{-}IfStmt : \Pi \xrightarrow{t} IfStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}IfStmt(\pi)(mk\text{-}IfStmt(test, cons, elsifaltn, altn))(t)(env) \triangleq$
 .2 let $altn' = \text{if } \text{elsifaltn} = []$
 .3 then $altn$
 .4 else $mk\text{-}IfStmt((\text{hd } \text{elsifaltn}).test, (\text{hd } \text{elsifaltn}).cons, \text{tl } \text{elsifaltn}, altn) \text{ in}$
 .5 InType[Expr](\pi)(wf-Expr(\pi))(test)(BOOLEAN)(env) \wedge

.6 cases $\pi :$
 .7 $mk\text{-}DEF(-) \rightarrow wf\text{-}Stmt(\pi)(cons)(t)(env) \wedge$
 .8 $wf\text{-}Stmt(\pi)(altn')(t)(env),$
 .9 $mk\text{-}POS(-) \rightarrow wf\text{-}Stmt(\pi)(cons)(t)(env) \vee$
 .10 $wf\text{-}Stmt(\pi)(altn')(t)(env)$
 .11 end;

```

299.0   wf-CasesStmt :  $\Pi \xrightarrow{t} CasesStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1   wf-CasesStmt ( $\pi$ )( $e$ )( $t$ )( $env$ )  $\triangleq$ 
.2     let mk-CasesStmt( $sel$ ,  $altns$ , (nil)) = CasesStmt2CasesStmt( $e$ ),
.3        $\pi_{POS} = mk-POS(\pi.checkset)$  in
.4      $\exists sel_t : TypeR .$ 
.5       wf-Expr( $\pi$ )( $sel$ )( $sel_t$ )( $env$ )  $\wedge$ 
.6       let  $altns' = SimplifyCaseStmtAlternatives(\pi.checkset)(altns)(sel_t)(env)$  in
.7       cases  $\pi :$ 
.8         mk-DEF(-)  $\rightarrow$ 
.9            $\forall alt \in \text{elems } altns' .$ 
.10           $\forall pat \in \text{elems } alt.match .$ 
.11             $\exists venv : ValEnv .$ 
.12              wf-Pattern( $\pi_{POS}$ )( $pat$ )(mk-TVE( $sel_t$ ,  $venv$ ))( $env$ )  $\Rightarrow$ 
.13                wf-Stmt( $\pi$ )( $alt.body$ )( $t$ )(UpdateEnv( $venv$ )( $env$ )),
.14          mk-POS(-)  $\rightarrow$ 
.15             $\exists alt \in \text{elems } altns' .$ 
.16             $\exists pat \in \text{elems } alt.match .$ 
.17             $\exists venv : ValEnv .$ 
.18              wf-Pattern( $\pi_{POS}$ )( $pat$ )(mk-TVE( $sel_t$ ,  $venv$ ))( $env$ )  $\Rightarrow$ 
.19                let wfs = InScope[Stmt]( $\pi$ )(dom  $venv$ )(wf-Stmt( $\pi$ )) in
.20                  wfs( $alt.body$ )( $t$ )(UpdateEnv( $venv$ )( $env$ ))
.21        end;
300.0   SimplifyCaseStmtAlternatives : Check-set  $\xrightarrow{t} CaseStmtAlternatives \rightarrow$ 
.1           TypeR  $\rightarrow Env \rightarrow CaseStmtAlternatives$ 
.2   SimplifyCaseStmtAlternatives ( $checkset$ )( $altns$ )( $sel_t$ )( $env$ )  $\triangleq$ 
.3     let  $match = (\text{hd } altns).match$ ,
.4        $\pi_{POS} = mk-POS(checkset)$ ,
.5        $\pi_{DEF} = mk-DEF(checkset)$ ,
.6        $match' = [match(i) | i \in \text{inds } match .$ 
.7          $\exists venv : ValEnv .$ 
.8           wf-Pattern( $\pi_{POS}$ )( $match(i)$ )(mk-TVE( $sel_t$ ,  $venv$ ))( $env$ ]),
.9            $altns' = \text{if len } altns = 1 \vee$ 
.10           $\exists venv : ValEnv, i \in \text{inds } match .$ 
.11            wf-Pattern( $\pi_{DEF}$ )( $match(i)$ )(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)]  $\curvearrowright altns'$ 

```

Given a type of the selector in the ‘cases’ statement, some of the alternatives may never come into consideration during execution.

```

301.0   CasesStmt2CasesStmt : CasesStmt  $\xrightarrow{t} CasesStmt$ 
.1   CasesStmt2CasesStmt (mk-CasesStmt( $sel$ ,  $altns$ , other))  $\triangleq$ 
.2     let  $otherbody = \text{if other} = \text{nil} \text{ then SKIP else } other$ ,
.3        $otheraltn = mk-CaseStmtAltn([mk-PatternId(nil)], otherbody)$  in
.4       mk-CasesStmt( $sel$ ,  $altns \curvearrowright [otheraltn]$ , nil)

```

11.5.4 Loop Statements

302.0 $wf\text{-}SeqForLoop : \Pi \xrightarrow{t} SeqForLoop \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}SeqForLoop(\pi)(mk\text{-}SeqForLoop(cv, dirm, forseq, body))(t)(env) \triangleq$
 .2 $\exists forseq_t : SeqTypeR \mid EMPTYSEQ \cdot$
 .3 $wf\text{-}Expr(\pi)(forseq)(forseq_t)(env) \wedge$
 .4 $\text{cases } forseq_t :$
 .5 $(EMPTYSEQ) \rightarrow$
 .6 $Subsume(\pi)(CONT, t)(env),$
 .7 $mk\text{-}Seq1TypeR(elemtp) \rightarrow$
 .8 $\exists venv : ValEnv \cdot$
 .9 $wf\text{-}PatternBind(\pi)(cv)(mk\text{-}TVE(elemtp, venv))(env) \wedge$
 .10 $InScope[Stmt](\pi)(\text{dom } venv)(wf\text{-}Stmt(\pi))(body)(t)(UpdateEnv(venv)(env)),$
 .11 $\text{others} \rightarrow \text{false}$
 .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 $wf\text{-}SetForLoop : \Pi \xrightarrow{t} SetForLoop \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

.1 $wf\text{-}SetForLoop(\pi)(mk\text{-}SetForLoop(cv, forset, body))(t)(env) \triangleq$
 .2 $\exists forset_t : SetTypeR \mid EMPTYSET \cdot$
 .3 $wf\text{-}Expr(\pi)(forset)(forset_t)(env) \wedge$
 .4 $\text{cases } forset_t :$
 .5 $(EMPTYSET) \rightarrow$
 .6 $Subsume(\pi)(CONT, t)(env),$
 .7 $mk\text{-}SetTypeR(elemtp) \rightarrow$
 .8 $\exists venv : ValEnv \cdot$
 .9 $wf\text{-}Pattern(\pi)(cv)(mk\text{-}TVE(elemtp, venv))(env) \wedge$
 .10 $InScope[Stmt](\pi)(\text{dom } venv)(wf\text{-}Stmt(\pi))(body)(t)(UpdateEnv(venv)(env)),$
 .11 $\text{others} \rightarrow \text{false}$
 .12 end

Iteration over sets is almost similar to iteration over sequences.

```

304.0 wf-IndexForLoop :  $\Pi \xrightarrow{t} IndexForLoop \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1 wf-IndexForLoop ( $\pi$ ) ( $mk\text{-}IndexForLoop(cv, lb, ub, step, body)$ ) ( $t$ ) ( $env$ )  $\triangleq$ 
.2   let  $CP : TypeR^* \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.3      $CP([lb_t, ub_t, step_t])(t)(env) \triangleq$ 
.4        $IsSubtype(\pi)(lb_t, INT)(env) \wedge$ 
.5        $IsSubtype(\pi)(ub_t, INT)(env) \wedge$ 
.6        $IsSubtype(\pi)(step_t, INT)(env) \wedge$ 
.7        $t = mk\text{-}UnionTypeR(\{lb_t, ub_t\})$ 
.8     in
.9    $\exists index_t : TypeR .$ 
.10    let  $step' = \text{if } step = \text{nil} \text{ then } mk\text{-}NumLit(1) \text{ else } step$  in
.11    wf-SCompExpr ( $\pi$ ) ( $CP$ ) ( $[lb, ub, step']$ ) ( $index_t$ ) ( $env$ )  $\wedge$ 
.12    let  $env_{body} = UpdateEnv(\{mk\text{-}Name}(cv) \mapsto mk\text{-}ValR(index_t, \pi)\})(env)$  in
.13    cases  $\pi$  :
.14       $mk\text{-}DEF(-) \rightarrow Subsume(\pi)(CONT, t)(env) \wedge$ 
.15         $wf\text{-}Stmt(\pi)(body)(t)(env_{body}),$ 
.16       $mk\text{-}POS(-) \rightarrow Subsume(\pi)(CONT, t)(env) \vee$ 
.17         $InScope[Stmt](\pi)(\{mk\text{-}Name}(cv)\})(wf\text{-}Stmt(\pi))(body)(t)(env_{body})$ 
.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-WhileLoop :  $\Pi \xrightarrow{t} WhileLoop \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1 wf-WhileLoop ( $\pi$ ) ( $mk\text{-}WhileLoop(test, body)$ ) ( $t$ ) ( $env$ )  $\triangleq$ 
.2    $InType[Expr](\pi)(wf\text{-}Expr(\pi))(test)(BOOLEAN)(env) \wedge$ 
.3   cases  $\pi$  :
.4      $mk\text{-}DEF(-) \rightarrow Subsume(\pi)(CONT, t)(env) \wedge$ 
.5        $\exists body_t : StmtTypeR .$ 
.6          $wf\text{-}Stmt(\pi)(body)(body_t)(env) \wedge$ 
.7          $IsDisjoint(\pi)(CONT, body_t)(env) \wedge$ 
.8          $Subsume(\pi)(body_t, t)(env),$ 
.9      $mk\text{-}POS(-) \rightarrow Subsume(\pi)(CONT, t)(env) \vee$ 
.10        $wf\text{-}Stmt(\pi)(body)(t)(env)$ 
.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).

11.5.5 Non-Deterministic Statement

```

306.0 wf-NonDetStmt :  $\Pi \xrightarrow{t} NonDetStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1 wf-NonDetStmt ( $\pi$ ) ( $mk\text{-}NonDetStmt(stmts)$ ) ( $t$ ) ( $env$ )  $\triangleq$ 
.2   cases  $\pi$  :
.3      $mk\text{-}DEF(-) \rightarrow \forall stmts' : Stmt^* .$ 
.4        $IsPermutationOf[Stmt](stmts', stmts) \Rightarrow$ 
.5          $wf\text{-}Sequence(\pi)(stmts')(t)(env),$ 
.6      $mk\text{-}POS(-) \rightarrow \exists stmts' : Stmt^* .$ 
.7        $IsPermutationOf[Stmt](stmts', stmts) \Rightarrow$ 
.8          $wf\text{-}Sequence(\pi)(stmts')(t)(env)$ 
.9   end

```

11.5.6 Call and Return Statements

307.0 $wf\text{-}Call : \Pi \xrightarrow{t} Call \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}Call(\pi)(mk\text{-}Call(oprt, args, callst))(t)(env) \triangleq$
- .2 $\exists oprt_t : OpTypeR, arg_t, callst_t : TypeR .$
- .3 $wf\text{-}Expr(\pi)(oprt)(oprt_t)(env) \wedge$
- .4 $wf\text{-}Expr(\pi)(arg)(arg_t)(env) \wedge$
- .5 $IsSubtype(\pi)(arg_t, oprt_t.opdom)(env) \wedge$
- .6 $Subsume(\pi)(oprt_t.oprng, t)(env) \wedge$
- .7 $(\text{if } callst = \text{nil}$
- .8 $\text{then } oprt_t.opstate = \text{nil}$
- .9 $\text{else } oprt_t.opstate \neq \text{nil} \wedge$
- .10 $wf\text{-}StateDesignator(\pi)(callst)(callst_t)(env) \wedge$
- .11 $IsSubtype(\pi)(callst_t, oprt_t.opstate)(env) \wedge$
- .12 $is\text{-}POS(\pi)$

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 $wf\text{-}ReturnStmt : \Pi \xrightarrow{t} ReturnStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}ReturnStmt(\pi)(mk\text{-}ReturnStmt(expr))(t)(env) \triangleq$
- .2 $\exists expr_t : [TypeR] .$
- .3 $(\text{if } expr = \text{nil}$
- .4 $\text{then } expr_t = \text{VOID}$
- .5 $\text{else } wf\text{-}Expr(\pi)(expr)(expr_t)(env) \wedge$
- .6 $Subsume(\pi)(mk\text{-}RetTypeR(expr_t), t)(env)$

11.5.7 Exception Handling Statements

309.0 $wf\text{-}AlwaysStmt : \Pi \xrightarrow{t} AlwaysStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}AlwaysStmt(\pi)(mk\text{-}AlwaysStmt(alwpost, body))(t)(env) \triangleq$
- .2 $\exists alwpost_t : StmtTypeR .$
- .3 $wf\text{-}Stmt(\pi)(alwpost)(alwpost_t)(env) \wedge$
- .4 $\forall alwpost_{t'} : StmtTypeR .$
- .5 $IsUnionComponent(alwpost_{t'}, alwpost_t)(env) \Rightarrow$
- .6 $\text{cases } alwpost_{t'} :$
- .7 $\quad mk\text{-}ExitTypeR(-) \rightarrow Subsume(\pi)(alwpost_{t'}, t)(env),$
- .8 $\quad \text{others} \quad \rightarrow wf\text{-}Stmt(\pi)(body)(t)(env)$
- .9 $\quad \text{end}$

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).

310.0 $wf\text{-TrapStmt} : \Pi \xrightarrow{t} TrapStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-TrapStmt}(\pi)(mk\text{-TrapStmt}(pat, trappost, body))(t)(env) \triangleq$
 .2 let $\pi_{DEF} = mk\text{-DEF}(\pi.\text{checkset})$,
 .3 $\pi_{POS} = mk\text{-POS}(\pi.\text{checkset})$ in
 .4 $\exists body_t : StmtTypeR$ ·
 .5 $wf\text{-Stmt}(\pi)(body)(body_t)(env) \wedge$
 .6 $\forall body_{t'} : StmtTypeR$ ·
 .7 $IsUnionComponent(body_{t'}, body_t)(env) \Rightarrow$
 .8 cases $body_{t'}$:
 .9 $mk\text{-ExitTypeR}((VOID)) \rightarrow \text{false}$,
 .10 $mk\text{-ExitTypeR}(et) \rightarrow$
 .11 cases π :
 .12 $mk\text{-DEF}(-) \rightarrow$
 .13 $(\forall venv : ValEnv$ ·
 .14 $wf\text{-Pattern}(\pi_{POS})(pat)(mk\text{-TVE}(et, venv))(env) \Rightarrow$
 .15 $wf\text{-Stmt}(\pi)(trappost)(t)(UpdateEnv(venv)(env)) \wedge$
 .16 $(\neg \exists venv : ValEnv$ ·
 .17 $wf\text{-Pattern}(\pi_{DEF})(pat)(mk\text{-TVE}(et, venv))(env) \Rightarrow$
 .18 $Subsume(\pi)(body_{t'}, t)(env)$),
 .19 $mk\text{-POS}(-) \rightarrow$
 .20 $(\exists venv : ValEnv$ ·
 .21 let $wfs = InScope[Stmt](\pi)(\text{dom } venv)(wf\text{-Stmt}(\pi))$ in
 .22 $wf\text{-Pattern}(\pi_{POS})(pat)(mk\text{-TVE}(et, venv))(env) \wedge$
 .23 $wfs(trappost)(t)(UpdateEnv(venv)(env)) \vee$
 .24 $\neg \exists venv : ValEnv$ ·
 .25 $wf\text{-Pattern}(\pi_{DEF})(pat)(mk\text{-TVE}(et, venv))(env) \wedge$
 .26 $Subsume(\pi)(body_{t'}, t)(env)$
 .27 end,
 .28 others $\rightarrow Subsume(\pi)(body_{t'}, t)(env)$
 .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 $wf\text{-RecTrapStmt} : \Pi \xrightarrow{t} RecTrapStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-RecTrapStmt}(\pi)(mk\text{-RecTrapStmt}(traps, body))(t)(env) \triangleq$
 .2 $\exists body_t : StmtTypeR$ ·
 .3 $wf\text{-Stmt}(\pi)(body)(body_t)(env) \wedge$
 .4 $wf\text{-Traps}(\pi)(traps)(body_t)(t)(env)$

The result of the execution of the body of a recursive trap statement is passed to the traps.

312.0 $wf\text{-}Traps : \Pi \xrightarrow{t} Traps \rightarrow StmtTypeR \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}Traps(\pi)(traps)(body_t)(t)(env) \triangleq$
- .2 let $\pi_{DEF} = mk\text{-}DEF(\pi.\text{checkset})$,
- .3 $\pi_{POS} = mk\text{-}POS(\pi.\text{checkset})$ in
- .4 $\forall body_{t'} : StmtTypeR$.
- .5 $IsUnionComponent(body_{t'}, body_t)(env) \Rightarrow$
- .6 cases $body_{t'}$:
- .7 $mk\text{-}ExitTypeR((VOID)) \rightarrow \text{false}$,
- .8 $mk\text{-}ExitTypeR(et) \rightarrow$
- .9 cases π :
- .10 $mk\text{-}DEF(cs) \rightarrow$
- .11 $\neg \exists trp \in \text{elems } traps, venv : ValEnv .$
- .12 $wf\text{-}PatternBind}(\pi_{POS})(trp.match)(mk\text{-}TVE(et, venv))(env),$
- .13 $mk\text{-}POS(cs) \rightarrow$
- .14 $(\exists trp \in \text{elems } traps, venv : ValEnv, trappost_t : StmtTypeR .$
- .15 let $newenv = UpdateEnv(venv)(env)$,
- .16 $pat = trp.match,$
- .17 $wfs = InScope[Stmt](\pi)(\text{dom } venv)(wf\text{-}Stmt(\pi))$ in
- .18 $wf\text{-}PatternBind}(\pi_{POS})(pat)(mk\text{-}TVE(et, venv))(env) \wedge$
- .19 $wfs(trp.trappost)(trappost_t)(newenv) \wedge$
- .20 $wf\text{-}Traps(\pi)(traps)(trappost_t)(t)(env)) \vee$
- .21 $\neg \exists trp \in \text{elems } traps, venv : ValEnv .$
- .22 $wf\text{-}PatternBind}(\pi_{DEF})(trp.match)(mk\text{-}TVE(et, venv))(env) \wedge$
- .23 $Subsume(\pi)(body_{t'}, t)(env)$
- .24 end,
- .25 others $\rightarrow Subsume(\pi)(body_{t'}, t)(env)$
- .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 $wf\text{-}ExitStmt : \Pi \xrightarrow{t} ExitStmt \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}ExitStmt(\pi)(mk\text{-}ExitStmt(expr))(t)(env) \triangleq$
- .2 $\exists expr_t : [TypeR] .$
- .3 (if $expr_t = \text{nil}$
- .4 then $expr_t = \text{VOID}$
- .5 else $wf\text{-}Expr(\pi)(expr)(expr_t)(env)) \wedge$
- .6 $Subsume(\pi)(mk\text{-}ExitTypeR(expr_t), t)(env)$

11.5.8 Identity Statements

314.0 $wf\text{-}IdentStmt : \Pi \xrightarrow{t} SKIP \rightarrow StmtTypeR \rightarrow Env \rightarrow \mathbb{B}$

- .1 $wf\text{-}IdentStmt(\pi)((SKIP))(t)(env) \triangleq$
- .2 $Subsume(\pi)(CONT, t)(env)$

11.6 Patterns and Bindings

In this section the well-formedness predicates for patterns and bindings are defined.

11.6.1 Patterns

The Static Semantics defines for each pattern class (*XPattern*) a basic well-formedness predicate indexed by the the well-formedness classification:

$$\begin{aligned} wf\text{-}XPattern : \Pi \xrightarrow{t} Pattern \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B} \\ wf\text{-}XPattern(\pi)(pat)(mk\text{-}TVE(t, venv))(env) \triangleq \\ \text{some-expr-to-be-explained-in-the-following} \end{aligned}$$

where

$$\begin{aligned} 315.0 \quad ValEnv &= Name \xrightarrow{m} ValR; \\ 316.0 \quad TVE :: type &: TypeR \\ .1 \quad venv &: ValEnv \end{aligned}$$

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 *denv* conforms to a static semantics value environment *venv* : *ValEnv* if *denv* and *venv* have the same domain, and for every identifier *id* in the domain is *denv(id)* a value in the set of values described by the type *venv(id)*. If a pattern is well-formed for some *mk-TVE(t, venv)*, then the generated dynamic value environments *conform* with *venv*.

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 sub-component 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*.

$$\begin{aligned} 317.0 \quad PatCP &= TVE^+ \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}; \\ 318.0 \quad PatTCP &= TypeR^+ \rightarrow TypeR \rightarrow Env \rightarrow \mathbb{B} \end{aligned}$$

```

319.0   wf-CompPatn :  $\Pi \xrightarrow{t} PatTCP \rightarrow Pattern^+ \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$ 
.1      wf-CompPatn ( $\pi$ ) (PTCP) (cp-seq) (mtve) (env)  $\triangleq$ 
.2      let PCP :  $TVE^+ \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$ 
.3          PCP (tve-seq) (mk-TVE (mt, mvenv)) (env)  $\triangleq$ 
.4          PTCP ([tve-seq(i).type | i  $\in$  inds tve-seq]) (mt) (env)  $\wedge$ 
.5          IsVenuSetMerge ( $\pi$ ) ({tve-seq(i).venv | i  $\in$  inds tve-seq}) (mvenv) (env)
.6          in
.7           $\exists$  ctve-seq :  $TVE^+$ .
.8              len ctve-seq = len cp-seq  $\wedge$ 
.9               $\forall i \in$  inds cp-seq .
.10             wf-Pattern ( $\pi$ ) (cp-seq(i)) (ctve-seq(i)) (env)  $\wedge$ 
.11             cases  $\pi$  :
.12                 mk-DEF ({UNIONCLOSE})  $\cup$  -  $\rightarrow$ 
.13                 UnionClosePCP (PCP) (ctve-seq) (mtve) (env),
.14                 mk-DEF (-)  $\rightarrow$  PCP (ctve-seq) (mtve) (env),
.15                 mk-POS (-)  $\rightarrow$  ComplPCP ( $\pi$ ) (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 :  $\Pi \xrightarrow{t} PatCP \rightarrow PatCP$ 
.1      ComplPCP ( $\pi$ ) (PCP) (ctve-seq) (mk-TVE (t, venv)) (env)  $\triangleq$ 
.2      let liftedtoPCP :  $TVE^+ \xrightarrow{t} \mathbb{B}$ 
.3          liftedtoPCP (mtves)  $\triangleq$ 
.4          len mtves = len ctve-seq  $\wedge$ 
.5           $\forall i \in$  inds mtves . IsSubTVE ( $\pi$ ) (ctve-seq(i), mtves(i)) (env)  $\wedge$ 
.6           $\exists$  mtve :  $TVE \cdot PCP (mtves) (mtve) (env)$ 
.7          in
.8           $\exists$  mtves :  $TVE^+$ .
.9              liftedtoPCP (mtves)  $\wedge$  PCP (mtves) (mk-TVE (t, venv)) (env)  $\wedge$ 
.10              $\forall$  omtves :  $TVE^+$ .
.11                 liftedtoPCP (omtves)  $\Rightarrow$ 
.12                  $\forall i \in$  inds omtves .
.13                     IsSubTVE ( $\pi$ ) (omtves(i), mtves(i)) (env)  $\Rightarrow$ 
.14                     IsSubTVE ( $\pi$ ) (mtves(i), omtves(i)) (env);

```

```

321.0   IsSubTVE :  $\Pi \xrightarrow{t} TVE \times TVE \rightarrow Env \rightarrow \mathbb{B}$ 
.1      IsSubTVE ( $\pi$ ) (mk-TVE (t, venv), mk-TVE (mt, mvenv)) (env)  $\triangleq$ 
.2      IsSubtypeR ( $\pi$ ) (t, mt) (env)  $\wedge$  IsSubValEnv ( $\pi$ ) (venv, mvenv) (env)

```

Relaxation of Pattern Characteristic Predicates

- 322.0 $\text{UnionClosePCP} : \text{PatCP} \xrightarrow{t} \text{TVE}^+ \rightarrow \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{UnionClosePCP}(\text{PCP})(\text{ctveseq})(\text{rtve})(\text{env}) \triangleq$
 $\text{PCP}(\text{ctveseq})(\text{rtve})(\text{env}) \vee$
 - .3 $\exists \text{rtve}_1, \text{rtve}_2 : \text{TVE}, \text{ctveseq}_1, \text{ctveseq}_2 : \text{TVE}^+$
 $\text{IsTveUnion}(\text{rtve}_1, \text{rtve}_2)(\text{rtve})(\text{env}) \wedge$
 - .5 $\text{IsTveSeqUnion}(\text{ctveseq}_1, \text{ctveseq}_2)(\text{ctveseq})(\text{env}) \wedge$
 - .6 $\text{UnionClosePCP}(\text{PCP})(\text{ctveseq}_1)(\text{rtve}_1)(\text{env}) \wedge$
 - .7 $\text{UnionClosePCP}(\text{PCP})(\text{ctveseq}_2)(\text{rtve}_2)(\text{env})$

UnionClosePCP relaxes pattern characteristic predicates so they may be able to accept match types which are union types.

- 323.0 $\text{IsTveSeqUnion} : \text{TVE}^+ \times \text{TVE}^+ \xrightarrow{t} \text{TVE}^+ \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{IsTveSeqUnion}(\text{tveseq}_1, \text{tveseq}_2)(\text{tveseq})(\text{env}) \triangleq$
 $\text{len tveseq}_1 = \text{len tveseq} \wedge \text{len tveseq}_2 = \text{len tveseq} \wedge$
 - .3 $\forall i \in \text{inds tveseq} .$
 $\text{IsTveUnion}(\text{tveseq}_1(i), \text{tveseq}_2(i))(\text{tveseq}(i))(\text{env});$
- 324.0 $\text{IsTveUnion} : \text{TVE} \times \text{TVE} \xrightarrow{t} \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{IsTveUnion}(\text{mk-TVE}(t_1, \text{ve}_1), \text{mk-TVE}(t_2, \text{ve}_2))(\text{mk-TVE}(t, \text{ve}))(\text{env}) \triangleq$
 $t = \text{mk-UnionTypeR}(\{t_1, t_2\}) \wedge \text{IsVenvUnion}(\text{ve}_1, \text{ve}_2)(\text{ve})(\text{env})$

Well-formedness of Patterns

- 325.0 $\text{wf-PatternBasic} : \Pi \xrightarrow{t} \text{Pattern} \rightarrow \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$
- .1 $\text{wf-PatternBasic}(\pi)(\text{pat})(\text{tve})(\text{env}) \triangleq$
 - .2 cases pat :
 - .3 $\text{mk-PatternId}(-) \rightarrow \text{wf-PatternId}(\pi)(\text{pat})(\text{tve})(\text{env}),$
 - .4 $\text{mk-MatchVal}(-) \rightarrow \text{wf-MatchVal}(\pi)(\text{pat})(\text{tve})(\text{env}),$
 - .5 $\text{mk-SetEnumPattern}(-) \rightarrow \text{wf-SetEnumPattern}(\pi)(\text{pat})(\text{tve})(\text{env}),$
 - .6 $\text{mk-SetUnionPattern}(-, -) \rightarrow \text{wf-SetUnionPattern}(\pi)(\text{pat})(\text{tve})(\text{env}),$
 - .7 $\text{mk-SeqEnumPattern}(-) \rightarrow \text{wf-SeqEnumPattern}(\pi)(\text{pat})(\text{tve})(\text{env}),$
 - .8 $\text{mk-SeqConcPattern}(-, -) \rightarrow \text{wf-SeqConcPattern}(\pi)(\text{pat})(\text{tve})(\text{env}),$
 - .9 $\text{mk-TuplePattern}(-) \rightarrow \text{wf-TuplePattern}(\pi)(\text{pat})(\text{tve})(\text{env}),$
 - .10 $\text{mk-RecordPattern}(-, -) \rightarrow \text{wf-RecordPattern}(\pi)(\text{pat})(\text{tve})(\text{env})$
 - .11 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.

```

326.0   wf-Pattern :  $\Pi \xrightarrow{t} \text{Pattern} \rightarrow \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   wf-Pattern ( $\pi$ ) (pat) (tve) (env)  $\triangleq$ 
.2   cases  $\pi$  :
.3     mk-DEF (-)  $\rightarrow$  wf-PatternBasic ( $\pi$ ) (pat) (tve) (env),
.4     mk-POS (-)  $\rightarrow$ 
.5        $\exists \text{mtve} : \text{TVE} .$ 
.6         wf-PatternBasic ( $\pi$ ) (pat) (mtve) (env)  $\wedge$ 
.7         SubsumeTVE ( $\pi$ ) (mtve, tve) (env)
.8   end;

327.0   SubsumeTVE :  $\Pi \xrightarrow{t} \text{TVE} \times \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   SubsumeTVE ( $\pi$ ) (mk-TVE (mt, mvenv), mk-TVE (t, venv)) (env)  $\triangleq$ 
.2     IsSubtypeR ( $\pi$ ) (t, mt) (env)  $\wedge$ 
.3     IsSubValEnv ( $\pi$ ) (venv, mvenv) (env);

328.0   IsSubValEnv :  $\Pi \xrightarrow{t} \text{ValEnv} \times \text{ValEnv} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   IsSubValEnv ( $\pi$ ) (svenv, venv) (env)  $\triangleq$ 
.2     dom svenv = dom venv  $\wedge$ 
.3      $\forall id \in \text{dom svenv} .$ 
.4       IsSubtypeR ( $\pi$ ) (svenv(id).type, venv(id).type) (env);

329.0   wf-PatternBind :  $\Pi \xrightarrow{t} \text{PatternBind} \rightarrow \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   wf-PatternBind ( $\pi$ ) (patbind) (tve) (env)  $\triangleq$ 
.2   cases patbind :
.3     mk-SetBind (-, -)  $\rightarrow$  wf-SetConstrPattern ( $\pi$ ) (patbind) (tve) (env),
.4     mk-TypeBind (-, -)  $\rightarrow$  wf-TypeConstrPattern ( $\pi$ ) (patbind) (tve) (env),
.5     others  $\rightarrow$  wf-Pattern ( $\pi$ ) (patbind) (tve) (env)
.6   end

```

The *PatternBind* construction is only used as a pattern, in spite of its name.

```

330.0   wf-PatternId :  $\Pi \xrightarrow{t} \text{PatternId} \rightarrow \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   wf-PatternId ( $\pi$ ) (mk-PatternId (id)) (mk-TVE (t, ve)) (env)  $\triangleq$ 
.2   cases id :
.3     (nil)  $\rightarrow$  ve = EmptyValEnv,
.4     id  $\rightarrow$  ve = {mk-Name (id)  $\mapsto \text{mk-ValR}$  (t, \pi)}
.5   end;

331.0   wf-MatchVal :  $\Pi \xrightarrow{t} \text{MatchVal} \rightarrow \text{TVE} \rightarrow \text{Env} \rightarrow \mathbb{B}$ 
.1   wf-MatchVal ( $\pi$ ) (mk-MatchVal (e)) (tve) (env)  $\triangleq$ 
.2     tve.venv = EmptyValEnv  $\wedge$ 
.3      $\exists et : \text{TypeR} .$ 
.4       wf-Expr ( $\pi$ ) (e) (et) (env)  $\wedge$  SameValue ( $\pi$ ) (et, tve.type) (env);

```

332.0 $wf\text{-}SetEnumPattern : \Pi \xrightarrow{t} SetEnumPattern \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}SetEnumPattern(\pi)(mk\text{-}SetEnumPattern(pats))(tve)(env) \triangleq$
 .2 let $PTCP : TypeR^+ \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $PTCP(ct\text{-}seq)(t)(env) \triangleq$
 .4 $Shape(t) = mk\text{-}SetTypeR(mk\text{-}UnionTypeR(\{ct\text{-}seq(i) \mid i \in \text{inds } ct\text{-}seq\})) \wedge$
 .5 $\text{CardofSet}(\pi)(t, \text{len } ct\text{-}seq)(env)$
 .6 in
 .7 $wf\text{-}CompPatn(\pi)(PTCP)(pats)(tve)(env)$

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 $wf\text{-}SetUnionPattern : \Pi \xrightarrow{t} SetUnionPattern \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}SetUnionPattern(\pi)(mk\text{-}SetUnionPattern(lp, rp))(tve)(env) \triangleq$
 .2 let $PTCP : TypeR^+ \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $PTCP([t_1, t_2])(t)(env) \triangleq$
 .4 $(is\text{-}SetTypeR(t) \vee t = \text{EMPTYSET}) \wedge t_1 = t \wedge t_2 = t$
 .5 in
 .6 $wf\text{-}CompPatn(\pi)(PTCP)([lp, rp])(tve)(env)$

Notice that information about cardinality has been dropped, so component patterns making requirements about cardinality cannot be checked.

334.0 $wf\text{-}SeqEnumPattern : \Pi \xrightarrow{t} SeqEnumPattern \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}SeqEnumPattern(\pi)(mk\text{-}SeqEnumPattern(pats))(tve)(env) \triangleq$
 .2 let $PTCP : TypeR^+ \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $PTCP(ct\text{-}seq)(t)(env) \triangleq$
 .4 $Shape(t) = mk\text{-}Seq1TypeR(mk\text{-}UnionTypeR(\{ct\text{-}seq(i) \mid i \in \text{inds } ct\text{-}seq\})) \wedge$
 .5 $\text{LengthofSeq}(\pi)(t, \text{len } ct\text{-}seq)(env)$
 .6 in
 .7 $wf\text{-}CompPatn(\pi)(PTCP)(pats)(tve)(env)$

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 $wf\text{-}SeqConcPattern : \Pi \xrightarrow{t} SeqConcPattern \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$.
 .1 $wf\text{-}SeqConcPattern(\pi)(mk\text{-}SeqConcPattern(lp, rp))(tve)(env) \triangleq$
 .2 let $PTCP : TypeR^+ \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $PTCP([t_1, t_2])(t)(env) \triangleq$
 .4 cases $\{t_1, t_2\}$:
 .5 $\{mk\text{-}Seq0TypeR(et)\} \rightarrow t = mk\text{-}Seq0TypeR(et),$
 .6 $\{mk\text{-}Seq0TypeR(et), mk\text{-}Seq1TypeR(et)\} \rightarrow t = mk\text{-}Seq1TypeR(et),$
 .7 $\{mk\text{-}Seq1TypeR(et)\} \rightarrow t = mk\text{-}Seq1TypeR(et),$
 .8 others \rightarrow false
 .9 end
 .10 in
 .11 $wf\text{-}CompPatn(\pi)(PTCP)([lp, rp])(tve)(env)$

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 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 $wf\text{-}TuplePattern : \Pi \xrightarrow{t} TuplePattern \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}TuplePattern(\pi)(mk\text{-}TuplePattern(fdःpats))(tve)(env) \triangleq$
 .2 let $PTCP : TypeR^+ \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $PTCP(ft\text{-}seq)(t)(env) \triangleq$
 .4 $t = mk\text{-}ProductTypeR([ft\text{-}seq}(i) \mid i \in \text{inds } ft\text{-}seq])$
 .5 in
 .6 $wf\text{-}CompPatn(\pi)(PTCP)(fdःpats)(tve)(env)$

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 $wf\text{-}RecordPattern : \Pi \xrightarrow{t} RecordPattern \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}RecordPattern(\pi)(mk\text{-}RecordPattern(tag, fdःpats))(tve)(env) \triangleq$
 .2 let $PTCP : TypeR^+ \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$
 .3 $PTCP(ft\text{-}seq)(t)(env) \triangleq$
 .4 $t = mk\text{-}CompositeTypeR(tag, [mk\text{-}FieldR(nil, ft\text{-}seq}(i)) \mid i \in \text{inds } ft\text{-}seq])$
 .5 in
 .6 $wf\text{-}CompPatn(\pi)(PTCP)(fdःpats)(tve)(env);$
 338.0 $wf\text{-}SetConstrPattern : \Pi \xrightarrow{t} SetBind \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}SetConstrPattern(\pi)(mk\text{-}SetBind(pat, setexpr))(tve)(env) \triangleq$
 .2 $wf\text{-}Pattern(\pi)(pat)(tve)(env) \wedge$
 .3 $\exists elemtype : TypeR .$
 .4 $wf\text{-}Expr(\pi)(setexpr)(mk\text{-}SetTypeR(elemtype))(env) \wedge$
 .5 $SameValue(\pi)(elemtype, tve.type)(env);$
 339.0 $wf\text{-}TypeConstrPattern : \Pi \xrightarrow{t} TypeBind \rightarrow TVE \rightarrow Env \rightarrow \mathbb{B}$
 .1 $wf\text{-}TypeConstrPattern(\pi)(mk\text{-}TypeBind(pat, constrtype))(tve)(env) \triangleq$
 .2 $wf\text{-}Pattern(\pi)(pat)(tve)(env) \wedge$
 .3 $\exists cstrt : TypeR .$
 .4 $wf\text{-}Type(\pi)(constrtype)(cstrt)(env) \wedge IsSubtypeR(\pi)(tve.type, cstrt)(env)$

11.6.2 Bindings

The static semantics defines for each bind construct a well-formedness predicate (as usually indexed by the well-formedness classification):

$wf\text{-}Bind : \Pi \xrightarrow{t} Bind \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$
 $wf\text{-}Bind(\pi)(bind)(venv, bindattrs)(env) \triangleq$
 $body$

where *ValEnv* is defined in 315 and *BindAttrs* is defined below:

340.0 $BindAttrs = (\text{EMPTY} \mid \text{FINITE})\text{-set}$

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 looseness). The number of bindings produced is described by some binding attributes *bindattrs* : *BindAttrs*.

```

341.0   wf-Bind :  $\Pi \xrightarrow{t} Bind \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$ 
.1      wf-Bind ( $\pi$ )(bind)(venv, bindattrs)(env)  $\triangleq$ 
.2      cases bind :
.3          mk-SetBind(-, -)  $\rightarrow$  wf-SetBind( $\pi$ )(bind)(venv, bindattrs)(env),
.4          mk-TypeBind(-, -)  $\rightarrow$  wf-TypeBind( $\pi$ )(bind)(venv, bindattrs)(env)
.5      end;

342.0   wf-SetBind :  $\Pi \xrightarrow{t} SetBind \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$ 
.1      wf-SetBind ( $\pi$ )(mk-SetBind(pat, setexpr))(venv, bindattrs)(env)  $\triangleq$ 
.2      let  $\pi' = NegWfClass(\pi)$ ,
.3          SetMatch :  $\Pi \xrightarrow{t} TVE \rightarrow \mathbb{B}$ 
.4          SetMatch ( $\pi$ )(mk-TVE(elemt, venv))  $\triangleq$ 
.5           $\exists$  sett : TypeR .
.6              wf-Expr( $\pi$ )(setexpr)(sett)(env)  $\wedge$ 
.7              wf-Pattern( $\pi$ )(pat)(mk-TVE(elemt, venv))(env)  $\wedge$ 
.8              IsNonEmptySet( $\pi$ )(sett)(env)  $\wedge$ 
.9              Shape(sett) = mk-SetTypeR(elemt)
.10         in
.11     cases  $\pi$  :
.12         mk-DEF(-)  $\rightarrow$ 
.13              $(\exists et : TypeR \cdot wf-Expr(\pi)(setexpr)(mk-SetTypeR(et))(env)) \wedge$ 
.14              $\exists tve : TVE \cdot wf-Pattern(\pi)(pat)(tve)(env) \wedge$ 
.15              $\forall ptve : TVE .$ 
.16                 SetMatch( $\pi'$ )(ptve)  $\Rightarrow$  IsSubValEnv( $\pi$ )(ptve.venv, venv)(env)  $\wedge$ 
.17                 (EMPTY  $\in$  bindattrs  $\Rightarrow$   $\neg \exists ptve : TVE . SetMatch(\pi')(ptve)$ ),
.18         mk-POS(-)  $\rightarrow$ 
.19              $(\exists sett, et : TypeR .$ 
.20                 wf-Expr( $\pi$ )(setexpr)(sett)(env)  $\wedge$ 
.21                 Shape(sett) = mk-SetTypeR(et)  $\wedge$ 
.22                 (wf-Pattern( $\pi$ )(pat)(mk-TVE(et, venv))(env)  $\vee$ 
.23                   venv = { $\mapsto$ }))
.24              $\wedge$ 
.25              $(\neg \exists dtve : TVE . SetMatch(\pi')(dtve) \Leftrightarrow$  EMPTY  $\in$  bindattrs)  $\wedge$ 
.26             FINITE  $\in$  bindattrs
.27     end;

343.0   IsNonEmptySet :  $\Pi \xrightarrow{t} TypeR \rightarrow Env \rightarrow \mathbb{B}$ 
.1      IsNonEmptySet ( $\pi$ )(sett)(env)  $\triangleq$ 
.2      is-POS( $\pi$ )

```

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 definitely 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 $mk\text{-}TVE(elemt, venv)$ 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 to a value which possibly matches the pattern. The Static Semantics now requires that for all $mk\text{-}TVE(pelemnt, pvenv)$ (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 definitely 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 :  $\Pi \xrightarrow{t} TypeBind \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$ 
.1     wf-TypeBind ( $\pi$ ) ( $mk\text{-}TypeBind(pat, type)$ ) ( $venv, bindattrs$ ) ( $env$ )  $\triangleq$ 
.2       let  $\pi' = NegWfClass(\pi)$ ,
.3          $TypeMatch : \Pi \rightarrow TVE \rightarrow TypeR \rightarrow \mathbb{B}$ 
.4          $TypeMatch(\pi)(mk\text{-}TVE(stp, venv))(t) \triangleq$ 
.5            $IsSubtypeR(\pi)(stp, t)(env) \wedge$ 
.6            $wf\text{-}Pattern(\pi)(pat)(mk\text{-}TVE(stp, venv))(env)$ 
.7           in
.8              $\exists t : TypeR .$ 
.9                $wf\text{-}Type(\pi)(type)(t)(env) \wedge$ 
.10              cases  $\pi$  :
.11                 $mk\text{-}DEF(-) \rightarrow$ 
.12                   $(\exists tve : TVE . wf\text{-}Pattern(\pi)(pat)(tve)(env)) \wedge$ 
.13                   $\forall ptve : TVE .$ 
.14                     $TypeMatch(\pi')(ptve)(t) \Rightarrow IsSubValEnv(\pi)(ptve.venv, venv)(env) \wedge$ 
.15                     $(\text{EMPTY} \in bindattrs \Rightarrow \neg \exists ptve : TVE . TypeMatch(\pi')(ptve)(t)) \wedge$ 
.16                     $(\text{FINITE} \in bindattrs \Rightarrow IsFiniteTypeR(\pi)(t)(env)),$ 
.17                 $mk\text{-}POS(-) \rightarrow$ 
.18                   $((\exists stp : TypeR . TypeMatch(\pi)(mk\text{-}TVE(stp, venv))(t)) \vee$ 
.19                   $\exists ptve : TVE . wf\text{-}Pattern(\pi)(pat)(ptve)(env) \wedge venv = \{\mapsto\})$ 
.20                   $\wedge$ 
.21                   $(\neg \exists dtve : TVE . TypeMatch(\pi')(dtve)(t) \Rightarrow \text{EMPTY} \in bindattrs) \wedge$ 
.22                   $(IsFiniteTypeR(\pi)(t)(env) \Rightarrow \text{FINITE} \in bindattrs)$ 
.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 $wf\text{-}BindList : \Pi \xrightarrow{t} BindList \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}BindList(\pi)(multbindlist)(venv, bindattrs)(env) \triangleq$
 - .2 $\exists venv\text{-}battrss : (ValEnv \times BindAttrs)^*$.
 - .3 $\text{len } venv\text{-}battrss = \text{len } multbindlist \wedge$
 - .4 $\text{IsBindSetMerge}(\pi)(\text{elems } venv\text{-}battrss)(venv, bindattrs)(env) \wedge$
 - .5 $\forall i \in \text{inds } multbindlist \cdot$
 - .6 $wf\text{-}MultBind}(\pi)(multbindlist(i))(venv\text{-}battrss(i))(env);$
- 346.0 $wf\text{-}MultBind : \Pi \xrightarrow{t} MultBind \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}MultBind(\pi)(multbind)(venv, bindattrs)(env) \triangleq$
 - .2 $\text{cases } multbind :$
 - .3 $mk\text{-}MultSetBind(-, -) \rightarrow wf\text{-}MultSetBind}(\pi)(multbind)(venv, bindattrs)(env),$
 - .4 $mk\text{-}MultTypeBind(-, -) \rightarrow wf\text{-}MultTypeBind}(\pi)(multbind)(venv, bindattrs)(env)$
 - .5 $\text{end};$
- 347.0 $wf\text{-}MultSetBind : \Pi \xrightarrow{t} MultSetBind \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}MultSetBind(\pi)(mk\text{-}MultSetBind(pats, setexpr))(venv, bindattrs)(env) \triangleq$
 - .2 $\exists venv\text{-}battrss : (ValEnv \times BindAttrs)^*$.
 - .3 $\text{len } venv\text{-}battrss = \text{len } pats \wedge$
 - .4 $\text{IsBindSetMerge}(\pi)(\text{elems } venv\text{-}battrss)(venv, bindattrs)(env) \wedge$
 - .5 $\forall i \in \text{inds } pats \cdot$
 - .6 $wf\text{-}SetBind}(\pi)(mk\text{-}SetBind(pats(i), setexpr))(venv\text{-}battrss(i))(env);$
- 348.0 $wf\text{-}MultTypeBind : \Pi \xrightarrow{t} MultTypeBind \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$
- .1 $wf\text{-}MultTypeBind(\pi)(mk\text{-}MultTypeBind(pats, type))(venv, bindattrs)(env) \triangleq$
 - .2 $\exists venv\text{-}battrss : (ValEnv \times BindAttrs)^*$.
 - .3 $\text{len } venv\text{-}battrss = \text{len } pats \wedge$
 - .4 $\text{IsBindSetMerge}(\pi)(\text{elems } venv\text{-}battrss)(venv, bindattrs)(env) \wedge$
 - .5 $\forall i \in \text{inds } pats \cdot$
 - .6 $wf\text{-}TypeBind}(\pi)(mk\text{-}TypeBind(pats(i), type))(venv\text{-}battrss(i))(env)$

11.6.3 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 $IsVenvUnion : ValEnv \times ValEnv \xrightarrow{t} ValEnv \rightarrow Env \rightarrow \mathbb{B}$
- .1 $IsVenvUnion(ve_1, ve_2)(ve)(env) \triangleq$
 - .2 $(\forall id \in \text{dom } ve_1 \cap \text{dom } ve_2 \cdot$
 - .3 $ve(id).\text{type} = mk\text{-}UnionTypeR(\{ve_1(id).\text{type}, ve_2(id).\text{type}\})) \wedge$
 - .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 $IsVenvMerge : \Pi \xrightarrow{t} ValEnv \times ValEnv \rightarrow ValEnv \rightarrow Env \rightarrow \mathbb{B}$
 .1 $IsVenvMerge(\pi)(ve_1, ve_2)(ve)(env) \triangleq$
 .2 $\text{dom } ve = \text{dom } ve_1 \cup \text{dom } ve_2 \wedge$
 .3 $\forall id \in \text{dom } ve_1 \cap \text{dom } ve_2 .$
 .4 $\text{let } mk\text{-}(t_1, t_2, t) = mk\text{-}(ve_1(id).type, ve_2(id).type, ve(id).type) \text{ in}$
 .5 $\text{SameValue}(\pi)(t_1, t_2)(env) \wedge$
 .6 $IsIntersectionType(\pi)(t_1, t_2)(t)(env) \wedge$
 .7 $\forall id \in \text{dom } ve_1 \setminus \text{dom } ve_2 . ve(id) = ve_1(id) \wedge$
 .8 $\forall id \in \text{dom } ve_2 \setminus \text{dom } ve_1 . ve(id) = ve_2(id)$

$IsVenvMerge(\pi)(ve_1, ve_2)(ve)(env)$ 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 $IsVenvSetMerge : \Pi \xrightarrow{t} ValEnv\text{-set} \rightarrow ValEnv \rightarrow Env \rightarrow \mathbb{B}$
 .1 $IsVenvSetMerge(\pi)(ve\text{-set})(ve)(env) \triangleq$
 .2 cases $ve\text{-set} :$
 .3 $(\{\}) \rightarrow ve = EmptyValEnv,$
 .4 $\{ve_1\} \rightarrow ve = ve_1,$
 .5 $ves_1 \cup ves_2 \rightarrow$
 .6 $\exists ve_1, ve_2 : ValEnv .$
 .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 $IsBindSetMerge : \Pi \xrightarrow{t} (ValEnv \times BindAttrs)\text{-set} \rightarrow ValEnv \times BindAttrs \rightarrow Env \rightarrow \mathbb{B}$
 .1 $IsBindSetMerge(\pi)(ve\text{-bats-set})(ve, bats)(env) \triangleq$
 .2 $\text{let } ve\text{-set} = \{ve \mid mk\text{-}(ve, -) \in ve\text{-bats-set}\},$
 .3 $bats\text{-set} = \{bats \mid mk\text{-}(-, bats) \in ve\text{-bats-set}\} \text{ in}$
 .4 $IsVenvSetMerge(\pi)(ve\text{-set})(ve)(env) \wedge$
 .5 $(\text{EMPTY} \in \bigcup bats\text{-set} \Leftrightarrow \text{EMPTY} \in bats) \wedge$
 .6 $(\text{FINITE} \in \bigcap bats\text{-set} \Leftrightarrow \text{FINITE} \in bats)$

11.7 Auxiliary Functions

11.7.1 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 $\{mk\text{-}(x, y) \mid x, y \in \text{elems } ds .$
 .3 $FreeNamesInDef(x) \cap DefNamesInDef(y) \neq \{\}\};$

- 358.0 $\text{TransTypeRDep} : \Pi \xrightarrow{t} \text{TypeR} \rightarrow \text{Env} \rightarrow \text{TypeRDepRel}$
- .1 $\text{TransTypeRDep}(\pi)(\text{type})(\text{env}) \triangleq$
 .2 $\text{TransClosure}[\text{TypeR}](\text{TypeRDep}(\pi)(\text{type})(\text{env}));$
- 359.0 $\text{TypeRDep} : \Pi \xrightarrow{t} \text{TypeR} \rightarrow \text{Env} \rightarrow \text{TypeRDepRel}$
- .1 $\text{TypeRDep}(\pi)(\text{type})(\text{env}) \triangleq$
 .2 $\text{TypeRSubExprDep}(\pi)(\text{type}) \cup$
 .3 $\{mk-(x, \text{env.typemap}(x)) \mid x \in \text{dom env.typemap}\} \cup$
 .4 $\bigcup \{\text{TypeRSubExprDep}(\pi)(t) \mid t \in \text{rng env.typemap}\};$
- 360.0 $\text{TypeRSubExprDep} : \Pi \xrightarrow{t} \text{TypeR} \rightarrow \text{TypeRDepRel}$
- .1 $\text{TypeRSubExprDep}(\pi)(x) \triangleq$
 .2 $\{mk-(x, y) \mid y \in \text{TypeRDefBasis}(\pi)(x) \setminus \{x\}\};$
- 361.0 $\text{TransClosure}[@T] (\text{rel} : (@T \times @T)\text{-set}) \text{ res} : (@T \times @T)\text{-set}$
- .1 $\text{post rel} \subseteq \text{res} \wedge$
 .2 $\text{IsTransitivelyClosed}[@T](\text{res}) \wedge$
 .3 $\forall r : (@T \times @T)\text{-set}.$
 .4 $\text{rel} \subseteq r \wedge r \subset \text{res} \Rightarrow \neg \text{IsTransitivelyClosed}[@T](r);$
- 362.0 $\text{IsTransitivelyClosed}[@T] : (@T \times @T)\text{-set} \xrightarrow{t} \mathbb{B}$
- .1 $\text{IsTransitivelyClosed}(\text{rel}) \triangleq$
 .2 $\forall mk-(x, y), mk-(z, v) \in \text{rel}.$
 .3 $y = z \Rightarrow mk-(x, v) \in \text{rel};$
- 363.0 $\text{ReflExtension}[@T] : (@T \times @T)\text{-set} \xrightarrow{t} (@T \times @T)\text{-set}$
- .1 $\text{ReflExtension}(\text{rel}) \triangleq$
 .2 $\text{rel} \cup$
 .3 $\{mk-(x, x) \mid x : @T \cdot \exists y : @T.$
 .4 $mk-(x, y) \in \text{rel} \vee mk-(y, x) \in \text{rel}\};$
- 364.0 $\text{IsPermutationOf}[@T] : @T^* \times @T^* \xrightarrow{t} \mathbb{B}$
- .1 $\text{IsPermutationOf}(s_1, s_2) \triangleq$
 .2 $\exists m : \mathbb{N}_1 \xrightarrow{m} \mathbb{N}_1.$
 .3 $\text{card} \{\text{dom } m, \text{rng } m, \text{inds } s_1, \text{inds } s_2\} = 1 \wedge$
 .4 $\forall i \in \text{inds } s_1 \cdot s_1(i) = s_2(m(i))$

Free and Defined Names

- 365.0 $\text{FreeNamesInDefs} : \text{ValFnOpTpDefs} \xrightarrow{t} \text{Name-set}$
- .1 $\text{FreeNamesInDefs}(\text{ds}) \triangleq$
 .2 $\bigcup \{\text{FreeNamesInDef}(d) \mid d \in \text{elems ds}\} \setminus \text{DefNamesInDefs}(\text{ds});$

366.0 $\text{FreeNamesInDef} : \text{ValFnOpTpDef} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInDef}(d) \triangleq$
 .2 cases $d :$
 .3 $\text{mk-ValueDef}(pat, -, val) \rightarrow$
 .4 $\text{FreeNamesInPat}(pat) \cup \text{FreeNamesInExpr}(val),$
 .5 $\text{mk-ExplFnDef}(-, -, -, -, \text{parmslist}, body, fpre) \rightarrow$
 .6 let $bnames = \text{FreeNamesInExpr}(body),$
 .7 $pnames = \text{if } fpre = \text{nil} \text{ then } \{\} \text{ else } \text{FreeNamesInExpr}(fpre),$
 .8 $dnames = \text{DefNamesInPats}(\text{conc parmslist}) \text{ in}$
 .9 $bnames \cup pnames \setminus dnames$
 .10 $\cup \text{FreeNamesInPats}(\text{conc parmslist}),$
 .11 $\text{mk-ImplFnDef}(-, -, \text{partps}, \text{mk-IdType}(ri, -), fpre, fpost) \rightarrow$
 .12 let $ponames = \text{FreeNamesInExpr}(fpost),$
 .13 $prnames = \text{if } fpre = \text{nil} \text{ then } \{\} \text{ else } \text{FreeNamesInExpr}(fpre),$
 .14 $dnames = \text{DefNamesInParTypes}(\text{partps}) \cup \{\text{mk-Name}(ri)\} \text{ in}$
 .15 $ponames \cup prnames \setminus dnames$
 .16 $\cup \text{FreeNamesInParTypes}(\text{partps}),$
 .17 $\text{mk-ExplOprtDef}(-, -, -, \text{parms}, body, opre) \rightarrow$
 .18 let $bnames = \text{FreeNamesInStmt}(body),$
 .19 $pnames = \text{if } opre = \text{nil} \text{ then } \{\} \text{ else } \text{FreeNamesInExpr}(opre),$
 .20 $dnames = \text{DefNamesInPats}(\text{parms}) \text{ in}$
 .21 $bnames \cup pnames \setminus dnames$
 .22 $\cup \text{FreeNamesInPats}(\text{parms}),$
 .23 $\text{mk-ImplOprtDef}(-, \text{partps}, ritp, oexts, opre, opost, oexcp) \rightarrow$
 .24 let $ponames = \text{FreeNamesInExpr}(opost),$
 .25 $prnames = \text{if } opre = \text{nil} \text{ then } \{\} \text{ else } \text{FreeNamesInExpr}(opre),$
 .26 $enames = \text{FreeNamesInExceptions}(oexcp),$
 .27 $etnames = \text{FreeNamesInExternals}(oexts),$
 .28 $rnames = \text{if } ritp = \text{nil} \text{ then } \{\} \text{ else } \{\text{mk-Name}(ritp.id)\},$
 .29 $dnames = \text{DefNamesInParTypes}(\text{partps}) \cup rnames \text{ in}$
 .30 $ponames \cup prnames \cup enames \cup etnames \setminus dnames$
 .31 $\cup \text{FreeNamesInParTypes}(\text{partps}),$
 .32 others \rightarrow let $\{t\} = \text{rng ExtractTypeMap}([d])$ in
 .33 $\text{FreeNamesInTypeR}(t)$
 .34 end;

 367.0 $\text{FreeNamesInType} : [\text{Type}] \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInType}(t_1) \triangleq$
 .2 if $t_1 = \text{nil} \text{ then } \{\} \text{ else } \text{FreeNamesInTypeR}(\text{ExtractTypeR}(t_1)(\text{EmptyEnv}));$

 368.0 $\text{FreeNamesInTypeR} : \text{TypeR} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInTypeR}(t_1) \triangleq$
 .2 $\{n \mid n : \text{Name} .$
 .3 $\exists t_2 : \text{TypeR} .$
 .4 $\text{IsTypeRSubExpr}(t_2, t_1) \wedge$
 .5 $\text{is-Name}(t_2) \wedge n = t_2 \vee$
 .6 $\text{is-InvTypeR}(t_2) \wedge n \in \text{FreeNamesInInvInitFn}(t_2.\text{invariant})\};$

 369.0 $\text{FreeNamesInInvInitFn} : \text{InvInitFn} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInInvInitFn}(\text{mk-InvInitFn}(pat, expr)) \triangleq$
 .2 $\text{FreeNamesInExpr}(expr) \setminus \text{DefNamesInPat}(pat) \cup \text{FreeNamesInPat}(pat);$

370.0 $\text{FreeNamesInExprs} : [\text{Expr}]^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInExprs}(es) \triangleq$
 .2 $\bigcup \{\text{FreeNamesInExpr}(e) \mid e \in \text{elems } es\};$

371.0 $\text{FreeNamesInExpr} : [\text{Expr}] \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInExpr}(e) \triangleq$
 .2 cases $e :$
 .3 $\text{mk-BracketedExpr}(e_1) \rightarrow \text{FreeNamesInExpr}(e_1),$
 .4 $\text{mk-DefExpr}(defs, body) \rightarrow$
 .5 $\text{FreeNamesInExpr}(body) \setminus \text{DefNamesInDefBinds}(defs) \cup$
 .6 $\text{FreeNamesInDefBinds}(defs),$
 .7 $\text{mk-LetExpr}(ldefs, body) \rightarrow$
 .8 $\text{FreeNamesInExpr}(body) \setminus \text{DefNamesInDefs}(ldefs) \cup$
 .9 $\text{FreeNamesInDefs}(ldefs),$
 .10 $\text{mk-LetBeSExpr}(bind, cond, body) \rightarrow$
 .11 $\text{FreeNamesInExpr}(body) \cup \text{FreeNamesInExpr}(cond) \setminus \text{DefNamesInBind}(bind)$
 .12 $\cup \text{FreeNamesInBind}(bind),$
 .13 $\text{mk-IfExpr}(test, cons, altns, altn) \rightarrow$
 .14 $\text{FreeNamesInExpr}(test) \cup \text{FreeNamesInExpr}(cons) \cup$
 .15 $\text{FreeNamesInExpr}(altn) \cup$
 .16 $\bigcup \{\text{FreeNamesInExpr}(t) \cup \text{FreeNamesInExpr}(c) \mid$
 .17 $\text{mk-ElsifExpr}(t, c) \in \text{elems } altns\},$
 .18 $\text{mk-CasesExpr}(sel, altns, other) \rightarrow$
 .19 $\text{FreeNamesInExpr}(sel) \cup \text{FreeNamesInExpr}(other) \cup$
 .20 $\bigcup \{\text{FreeNamesInExpr}(body) \setminus \text{DefNamesInPats}(ps) \cup \text{FreeNamesInPats}(ps) \mid$
 .21 $\text{mk-CaseAltn}(ps, body) \in \text{elems } altns\},$
 .22 $\text{mk-PrefixExpr}(-, e_1) \rightarrow \text{FreeNamesInExpr}(e_1),$
 .23 $\text{mk-BinaryExpr}(left, -, right) \rightarrow \text{FreeNamesInExpr}(left) \cup \text{FreeNamesInExpr}(right),$
 .24 others $\rightarrow \text{FreeNamesInSpecExpr}(e)$
 .25 end;

372.0 $\text{FreeNamesInSpecExpr} : [\text{Expr}] \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInSpecExpr}(e) \triangleq$
 .2 cases $e :$
 .3 $\text{mk-MapInverseExpr}(e_1) \rightarrow \text{FreeNamesInExpr}(e_1),$
 .4 $\text{mk-ExistsExpr}(bs, p), \text{mk-ExistsUniqueExpr}(bs, p) \rightarrow$
 .5 $\text{FreeNamesInExpr}(p) \setminus \text{DefNamesInBind}(bs) \cup$
 .6 $\text{FreeNamesInBind}(bs),$
 .7 $\text{mk-AllExpr}(bs, p), \text{mk-IotaExpr}(bs, p) \rightarrow$
 .8 $\text{FreeNamesInExpr}(p) \setminus \text{DefNamesInBind}(bs) \cup$
 .9 $\text{FreeNamesInBind}(bs),$
 .10 $\text{mk-SetEnumeration}(es) \rightarrow \text{FreeNamesInExprs}(es),$
 .11 $\text{mk-SetComprehension}(e, bs, p) \rightarrow$
 .12 $\text{FreeNamesInExpr}(e) \cup \text{FreeNamesInExpr}(p) \setminus \text{DefNamesInBind}(bs)$
 .13 $\cup \text{FreeNamesInBind}(bs),$
 .14 $\text{mk-SetRange}(lb, ub) \rightarrow \text{FreeNamesInExpr}(lb) \cup \text{FreeNamesInExpr}(ub),$
 .15 $\text{mk-SeqEnumeration}(es) \rightarrow \text{FreeNamesInExprs}(es),$
 .16 $\text{mk-SeqComprehension}(e, b, p) \rightarrow$
 .17 $\text{FreeNamesInExpr}(e) \cup \text{FreeNamesInExpr}(p) \setminus \text{DefNamesInBind}(b)$
 .18 $\cup \text{FreeNamesInBind}(b),$
 .19 $\text{mk-SubSequence}(s, lb, ub) \rightarrow \text{FreeNamesInExpr}(lb) \cup \text{FreeNamesInExpr}(ub)$
 .20 $\cup \text{FreeNamesInExpr}(s),$
 .21 $\text{mk-MapEnumeration}(mls) \rightarrow$
 .22 $\bigcup \{ \text{FreeNamesInExpr}(d) \cup \text{FreeNamesInExpr}(r) \mid$
 .23 $\text{mk-Maplet}(d, r) \in \text{elems } mls \},$
 .24 $\text{mk-MapComprehension}(\text{mk-Maplet}(d, r), bs, p) \rightarrow$
 .25 $\text{FreeNamesInExpr}(d) \cup \text{FreeNamesInExpr}(r) \cup \text{FreeNamesInExpr}(p)$
 .26 $\setminus \text{DefNamesInBind}(bs)$
 .27 $\cup \text{FreeNamesInBind}(bs),$
 .28 $\text{mk-TupleConstructor}(es) \rightarrow \text{FreeNamesInExprs}(es),$
 .29 $\text{mk-RecordConstructor}(-, es) \rightarrow \text{FreeNamesInExprs}(es),$
 .30 $\text{mk-RecordModifier}(rec, rms) \rightarrow$
 .31 $\text{FreeNamesInExpr}(rec) \cup$
 .32 $\bigcup \{ \text{FreeNamesInExpr}(new) \mid$
 .33 $\text{mk-RecordModification}(-, new) \in \text{elems } rms \},$
 .34 $\text{mk-Apply}(e_1, es) \rightarrow \text{FreeNamesInExpr}(e_1) \cup \text{FreeNamesInExprs}(es),$
 .35 $\text{mk-FieldSelect}(e_1, -) \rightarrow \text{FreeNamesInExpr}(e_1),$
 .36 $\text{mk-Lambda}(tbs, body) \rightarrow \text{FreeNamesInExpr}(body) \setminus \text{DefNamesInBind}(tbs)$
 .37 $\cup \text{FreeNamesInBind}(tbs),$
 .38 $\text{mk-IsDefTypeExpr}(-, e_1) \rightarrow \text{FreeNamesInExpr}(e_1),$
 .39 $\text{mk-IsBasicTypeExpr}(-, e_1) \rightarrow \text{FreeNamesInExpr}(e_1),$
 .40 others \rightarrow if $\text{is-Name}(e)$ then $\{e\}$ else $\{\}$
 .41 end;

 373.0 $\text{FreeNamesInDefBinds} : \text{DefBind}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInDefBinds}(ds) \triangleq$
 .2 if $ds = []$ then $\{\}$
 .3 else $\text{FreeNamesInDefBind}(\text{hd } ds) \cup$
 .4 $(\text{FreeNamesInDefBinds}(\text{tl } ds) \setminus \text{DefNamesInDefBind}(\text{hd } ds));$

 374.0 $\text{FreeNamesInDefBind} : \text{DefBind} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInDefBind}(\text{mk-DefBind}(pb, rhs)) \triangleq$
 .2 $\text{FreeNamesInPatBind}(pb) \cup \text{FreeNamesInExpr}(rhs);$

375.0 $\text{FreeNamesInPatBind} : \text{PatternBind} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInPatBind}(pb) \triangleq$
 .2 if $\text{IsPattern}(pb)$ then $\text{FreeNamesInPat}(pb)$ else $\text{FreeNamesInBind}(pb);$

376.0 $\text{FreeNamesInBind} : \text{Bind} \mid \text{MultBind} \mid \text{BindList} \mid \text{TypeBindList} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInBind}(b) \triangleq$
 .2 cases $b :$
 .3 $\text{mk-SetBind}(pat, bset) \rightarrow \text{FreeNamesInPat}(pat) \cup \text{FreeNamesInExpr}(bset),$
 .4 $\text{mk-TypeBind}(pat, type) \rightarrow \text{FreeNamesInPat}(pat) \cup \text{FreeNamesInType}(type),$
 .5 $\text{mk-MultSetBind}(pats, bset) \rightarrow \text{FreeNamesInPats}(pats) \cup \text{FreeNamesInExpr}(bset),$
 .6 $\text{mk-MultTypeBind}(pats, type) \rightarrow \text{FreeNamesInPats}(pats) \cup \text{FreeNamesInType}(type),$
 .7 others $\rightarrow \bigcup \{\text{FreeNamesInBind}(b') \mid b' \in \text{elems } b\}$
 .8 end;

377.0 $\text{FreeNamesInPats} : \text{Pattern}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInPats}(ps) \triangleq$
 .2 $\bigcup \{\text{FreeNamesInPat}(p) \mid p \in \text{elems } ps\};$

378.0 $\text{FreeNamesInPat} : \text{Pattern} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInPat}(p) \triangleq$
 .2 cases $p :$
 .3 $\text{mk-MatchVal}(e) \rightarrow \text{FreeNamesInExpr}(e),$
 .4 $\text{mk-EnumPattern}(ps) \rightarrow \text{FreeNamesInPats}(ps),$
 .5 $\text{mk-UnionPattern}(lp, rp) \rightarrow \text{FreeNamesInPats}([lp, rp]),$
 .6 $\text{mk-SeqEnumPattern}(ps) \rightarrow \text{FreeNamesInPats}(ps),$
 .7 $\text{mk-SeqConcPattern}(lp, rp) \rightarrow \text{FreeNamesInPats}([lp, rp]),$
 .8 $\text{mk-TuplePattern}(ps) \rightarrow \text{FreeNamesInPats}(ps),$
 .9 $\text{mk-RecordPattern}(-, ps) \rightarrow \text{FreeNamesInPats}(ps),$
 .10 $\text{mk-PatternId}(n) \rightarrow \{\}$
 .11 end;

379.0 $\text{FreeNamesInStmts} : \text{Stmt}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInStmts}(ss) \triangleq$
 .2 $\bigcup \{\text{FreeNamesInStmt}(s) \mid s \in \text{elems } ss\};$

380.0 $\text{FreeNamesInStmt} : \text{Stmt} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInStmt}(s) \triangleq$
 .2 cases $s :$
 .3 $\text{mk-BlockStmt}(\text{decls}, \text{stmts}) \rightarrow$
 .4 $\text{FreeNamesInStmts}(\text{stmts}) \setminus \text{DefNamesInAssignDfs}(\text{decls}) \cup$
 .5 $\text{FreeNamesInAssignDfs}(\text{decls}),$
 .6 $\text{mk-AlwaysStmt}(\text{alwpost}, \text{body}) \rightarrow$
 .7 $\text{FreeNamesInStmt}(\text{alwpost}) \cup \text{FreeNamesInStmt}(\text{body}),$
 .8 $\text{mk-TrapStmt}(\text{pat}, \text{trappost}, \text{body}) \rightarrow$
 .9 $\text{FreeNamesInStmt}(\text{trappost}) \cup \text{FreeNamesInStmt}(\text{body}) \cup$
 .10 $\text{FreeNamesInPat}(\text{pat}),$
 .11 $\text{mk-RecTrapStmt}(\text{traps}, \text{body}) \rightarrow$
 .12 $\text{FreeNamesInStmt}(\text{body}) \cup$
 .13 $\cup \{\text{FreeNamesInPatBind}(\text{match}) \cup \text{FreeNamesInStmt}(\text{trappost}) \mid$
 .14 $\text{mk-Trap}(\text{match}, \text{trappost}) \in \text{elems } \text{traps}\},$
 .15 $\text{mk-DefStmt}(\text{eqdefs}, \text{body}) \rightarrow$
 .16 $\text{FreeNamesInStmt}(\text{body}) \setminus \text{DefNamesInEqDfs}(\text{eqdefs}) \cup$
 .17 $\text{FreeNamesInEqDfs}(\text{eqdefs}),$
 .18 $\text{mk-LetStmt}(\text{letdefs}, \text{body}) \rightarrow$
 .19 $\text{FreeNamesInStmt}(\text{body}) \setminus \text{DefNamesInDfs}(\text{letdefs}) \cup$
 .20 $\text{FreeNamesInDfs}(\text{letdefs}),$
 .21 $\text{mk-LetBeSTStmt}(\text{bind}, \text{cond}, \text{body}) \rightarrow$
 .22 $\text{FreeNamesInStmt}(\text{body}) \cup \text{FreeNamesInExpr}(\text{cond})$
 .23 $\setminus \text{DefNamesInBind}(\text{bind})$
 .24 $\cup \text{FreeNamesInBind}(\text{bind}),$
 .25 $\text{mk-AssignStmt}(\text{lhs}, \text{rhs}) \rightarrow$
 .26 $\text{FreeNamesInStateDesignator}(\text{lhs}) \cup \text{FreeNamesInExprOrCall}(\text{rhs}),$
 .27 $\text{mk-NonDetStmt}(\text{stmts}) \rightarrow \text{FreeNamesInStmts}(\text{stmts}),$
 .28 $\text{mk-SeqForLoop}(\text{cv}, \text{-}, \text{forseq}, \text{body}) \rightarrow$
 .29 $\text{FreeNamesInStmt}(\text{body}) \setminus \text{DefNamesInPatBind}(\text{cv}) \cup$
 .30 $\text{FreeNamesInExpr}(\text{forseq}) \cup \text{FreeNamesInPatBind}(\text{cv}),$
 .31 $\text{mk-SetForLoop}(\text{cv}, \text{forset}, \text{body}) \rightarrow$
 .32 $\text{FreeNamesInStmt}(\text{body}) \setminus \text{DefNamesInPat}(\text{cv}) \cup$
 .33 $\text{FreeNamesInExpr}(\text{forset}) \cup \text{FreeNamesInPat}(\text{cv}),$
 .34 $\text{mk-IndexForLoop}(\text{cv}, \text{lb}, \text{ub}, \text{step}, \text{body}) \rightarrow$
 .35 $\text{FreeNamesInStmt}(\text{body}) \setminus \{\text{mk-Name}(\text{cv})\} \cup$
 .36 $\text{FreeNamesInExpr}(\text{lb}) \cup \text{FreeNamesInExpr}(\text{ub}) \cup$
 .37 $\text{FreeNamesInExpr}(\text{step}),$
 .38 $\text{mk-WhileLoop}(\text{test}, \text{body}) \rightarrow \text{FreeNamesInExpr}(\text{test}) \cup \text{FreeNamesInStmt}(\text{body}),$
 .39 $\text{mk-IfStmt}(\text{test}, \text{cons}, \text{altns}, \text{altn}) \rightarrow$
 .40 $\text{FreeNamesInExpr}(\text{test}) \cup \text{FreeNamesInStmt}(\text{cons}) \cup$
 .41 $\text{FreeNamesInStmt}(\text{altn}) \cup$
 .42 $\cup \{\text{FreeNamesInExpr}(\text{t}) \cup \text{FreeNamesInStmt}(\text{c}) \mid$
 .43 $\text{mk-ElsifStmt}(\text{t}, \text{c}) \in \text{elems } \text{altns}\},$
 .44 $\text{mk-CasesStmt}(\text{sel}, \text{altns}, \text{other}) \rightarrow$
 .45 $\text{FreeNamesInExpr}(\text{sel}) \cup \text{FreeNamesInStmt}(\text{other}) \cup$
 .46 $\cup \{\text{FreeNamesInStmt}(\text{body}) \setminus \text{DefNamesInPats}(\text{ps}) \cup \text{FreeNamesInPats}(\text{ps}) \mid$
 .47 $\text{mk-CaseStmtAltn}(\text{ps}, \text{body}) \in \text{elems } \text{altns}\},$
 .48 $\text{mk-ReturnStmt}(\text{-}) \rightarrow \{\},$
 .49 $\text{mk-ExitStmt}(\text{-}) \rightarrow \{\}$
 .50 end;

381.0 $\text{FreeNamesInAssignDefs} : \text{AssignDef}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInAssignDefs}(\text{ads}) \triangleq$
 .2 $\text{if } \text{ads} = [] \text{ then } \{\}$
 .3 $\text{else } \text{FreeNamesInAssignDef}(\text{hd ads}) \cup$
 .4 $(\text{FreeNamesInAssignDefs}(\text{tl ads}) \setminus \text{DefNamesInAssignDef}(\text{hd ads}))$;

 382.0 $\text{FreeNamesInAssignDef} : \text{AssignDef} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInAssignDef}(\text{mk-AssignDef}(-, \text{tp}, \text{dcli})) \triangleq$
 .2 $\text{FreeNamesInType}(\text{tp}) \cup \text{FreeNamesInExprOrCall}(\text{dcli})$;

 383.0 $\text{FreeNamesInEqDefs} : \text{EqDef}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInEqDefs}(\text{eqds}) \triangleq$
 .2 $\text{if } \text{eqds} = [] \text{ then } \{\}$
 .3 $\text{else } \text{FreeNamesInEqDef}(\text{hd eqds}) \cup$
 .4 $(\text{FreeNamesInEqDefs}(\text{tl eqds}) \setminus \text{DefNamesInEqDef}(\text{hd eqds}))$;

 384.0 $\text{FreeNamesInEqDef} : \text{EqDef} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInEqDef}(\text{mk-EqDef}(\text{lhs}, \text{rhs})) \triangleq$
 .2 $\text{FreeNamesInPatBind}(\text{lhs}) \cup \text{FreeNamesInExprOrCall}(\text{rhs})$;

 385.0 $\text{FreeNamesInStateDesignator} : \text{StateDesignator} \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInStateDesignator}(\text{std}) \triangleq$
 .2 cases std :
 .3 $\text{mk-FieldRef}(v, -) \rightarrow \text{FreeNamesInStateDesignator}(v)$,
 .4 $\text{mk-MapOrSeqRef}(v, a) \rightarrow$
 .5 $\text{FreeNamesInStateDesignator}(v) \cup \text{FreeNamesInExpr}(a)$,
 .6 $\text{mk-Name}(-) \rightarrow \{\text{std}\}$
 .7 end ;

 386.0 $\text{FreeNamesInExprOrCall} : [\text{Expr} \mid \text{Call}] \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInExprOrCall}(\text{ec}) \triangleq$
 .2 $\text{if } \text{ec} = \text{nil} \text{ then } \{\}$
 .3 $\text{elseif Is-Expr}(\text{ec}) \text{ then } \text{FreeNamesInExpr}(\text{ec})$
 .4 $\text{else let } \text{mk-Call}(\text{o}, \text{a}, \text{c}) = \text{ec} \text{ in}$
 .5 $\{\text{o}\} \cup \text{FreeNamesInExprs}(\text{a}) \cup \text{FreeNamesInStateDesignator}(\text{c})$;

 387.0 $\text{FreeNamesInExceptions} : \text{Error}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInExceptions}(\text{es}) \triangleq$
 .2 $\bigcup \{\text{FreeNamesInExpr}(\text{c}) \cup \text{FreeNamesInExpr}(\text{a}) \mid$
 .3 $\text{mk-Error}(-, \text{c}, \text{a}) \in \text{elems es}\}$;

 388.0 $\text{FreeNamesInExternals} : \text{VarInf}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInExternals}(\text{es}) \triangleq$
 .2 $\bigcup \{\text{FreeNamesInType}(\text{t}) \mid \text{mk-VarInf}(-, -, \text{t}) \in \text{elems es}\}$;

 389.0 $\text{FreeNamesInParTypes} : \text{PatTypePair}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{FreeNamesInParTypes}(\text{ptps}) \triangleq$
 .2 $\bigcup \{\text{FreeNamesInPats}(\text{ps}) \cup \text{FreeNamesInType}(\text{t}) \mid$
 .3 $\text{mk-PatTypePair}(\text{ps}, \text{t}) \in \text{elems ptps}\}$;;

390.0 $\text{DefNamesInDefs} : \text{ValFnOpTpDefs} \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInDefs}(ds) \triangleq$
 .2 $\bigcup \{\text{DefNamesInDef}(d) \mid d \in \text{elems } ds\};$

391.0 $\text{DefNamesInDef} : \text{ValFnOpTpDef} \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInDef}(d) \triangleq$
 .2 cases $d :$
 .3 $\text{mk-ValueDef}(pat, -, -) \rightarrow \text{DefNamesInPat}(pat),$
 .4 $\text{mk-ExplFnDef}(id, -, -, -, -, -, -) \rightarrow \{\text{mk-Name}(id)\},$
 .5 $\text{mk-ImplFnDef}(id, -, -, -, -, -, -) \rightarrow \{\text{mk-Name}(id)\},$
 .6 $\text{mk-ExplOprtDef}(id, -, -, -, -, -, -) \rightarrow \{\text{mk-Name}(id)\},$
 .7 $\text{mk-ImplOprtDef}(id, -, -, -, -, -, -, -) \rightarrow \{\text{mk-Name}(id)\},$
 .8 others $\rightarrow \text{dom ExtractTypeMap}([d])$
 .9 end;

392.0 $\text{DefNamesInPats} : \text{Pattern}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInPats}(ps) \triangleq$
 .2 $\bigcup \{\text{DefNamesInPat}(p) \mid p \in \text{elems } ps\};$

393.0 $\text{DefNamesInPat} : \text{Pattern} \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInPat}(p) \triangleq$
 .2 cases $p :$
 .3 $\text{mk-MatchVal}(e) \rightarrow \{\},$
 .4 $\text{mk-SetEnumPattern}(ps) \rightarrow \text{DefNamesInPats}(ps),$
 .5 $\text{mk-SetUnionPattern}(lp, rp) \rightarrow \text{DefNamesInPats}([lp, rp]),$
 .6 $\text{mk-SeqEnumPattern}(ps) \rightarrow \text{DefNamesInPats}(ps),$
 .7 $\text{mk-SeqConcPattern}(lp, rp) \rightarrow \text{DefNamesInPats}([lp, rp]),$
 .8 $\text{mk-TuplePattern}(ps) \rightarrow \text{DefNamesInPats}(ps),$
 .9 $\text{mk-RecordPattern}(-, ps) \rightarrow \text{DefNamesInPats}(ps),$
 .10 $\text{mk-PatternId}(n) \rightarrow \text{if } n = \text{nil} \text{ then } \{\} \text{ else } \{n\}$
 .11 end;

394.0 $\text{DefNamesInPatBind} : \text{PatternBind} \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInPatBind}(pb) \triangleq$
 .2 if $\text{IsPattern}(pb)$ then $\text{DefNamesInPat}(pb)$ else $\text{DefNamesInBind}(pb);$

395.0 $\text{DefNamesInBind} : \text{Bind} \mid \text{MultBind} \mid \text{BindList} \mid \text{TypeBindList} \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInBind}(b) \triangleq$
 .2 cases $b :$
 .3 $\text{mk-SetBind}(pat, -) \rightarrow \text{DefNamesInPat}(pat),$
 .4 $\text{mk-TypeBind}(pat, -) \rightarrow \text{DefNamesInPat}(pat),$
 .5 $\text{mk-MultSetBind}(pats, -) \rightarrow \text{DefNamesInPats}(pats),$
 .6 $\text{mk-MultTypeBind}(pats, -) \rightarrow \text{DefNamesInPats}(pats),$
 .7 others $\rightarrow \bigcup \{\text{DefNamesInBind}(b') \mid b' \in \text{elems } b\}$
 .8 end;

396.0 $\text{DefNamesInDefBinds} : \text{DefBind}^* \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInDefBinds}(ds) \triangleq$
 .2 $\bigcup \{\text{DefNamesInDefBind}(db) \mid db \in \text{elems } ds\};$

397.0 $\text{DefNamesInDefBind} : \text{DefBind} \xrightarrow{t} \text{Name-set}$
 .1 $\text{DefNamesInDefBind}(\text{mk-DefBind}(pb, -)) \triangleq$
 .2 $\text{DefNamesInPatBind}(pb);$

- 398.0 $\text{DefNamesInAssignDefs} : \text{AssignDef}^* \xrightarrow{t} \text{Name-set}$
- .1 $\text{DefNamesInAssignDefs}(\text{ads}) \triangleq$
 - .2 $\bigcup \{\text{DefNamesInAssignDef}(\text{ad}) \mid \text{ad} \in \text{elems ads}\};$
- 399.0 $\text{DefNamesInAssignDef} : \text{AssignDef} \xrightarrow{t} \text{Name-set}$
- .1 $\text{DefNamesInAssignDef}(\text{mk-AssignDef}(v, -, -)) \triangleq$
 - .2 $\{\text{mk-Name}(v)\};$
- 400.0 $\text{DefNamesInEqDfs} : \text{EqDef}^* \xrightarrow{t} \text{Name-set}$
- .1 $\text{DefNamesInEqDfs}(\text{eqds}) \triangleq$
 - .2 $\bigcup \{\text{DefNamesInEqDef}(\text{eqd}) \mid \text{eqd} \in \text{elems eqds}\};$
- 401.0 $\text{DefNamesInEqDef} : \text{EqDef} \xrightarrow{t} \text{Name-set}$
- .1 $\text{DefNamesInEqDef}(\text{mk-EqDef}(\text{lhs}, -)) \triangleq$
 - .2 $\text{DefNamesInPatBind}(\text{lhs});$
- 402.0 $\text{DefNamesInParTypes} : \text{PatTypePair}^* \xrightarrow{t} \text{Name-set}$
- .1 $\text{DefNamesInParTypes}(\text{ptps}) \triangleq$
 - .2 $\bigcup \{\text{DefNamesInPats}(\text{ps}) \mid$
 - .3 $\text{mk-PatTypePair}(\text{ps}, -) \in \text{elems ptps}\}$

NOTE: The names defined in patterns are those formed from embedded identifiers (however not including identifiers in match value expressions).

Syntactical Sub-Components

- 403.0 $\text{Syn} = \text{ValFnOpDef} \mid \text{Bind} \mid \text{Expr} \mid \text{Stmt} \mid \text{Pattern} \mid$
 - .1 $\text{Type} \mid \text{OpType} \mid \text{TotalFnType} \mid \text{TypeR} \mid \text{Call} \mid \text{AssignDef} \mid \text{StateDesignator} \mid \text{Id};$

404.0 $\text{SynCompRel} = ([\text{Syn}] \times \text{Syn})\text{-set}$

405.0 $\text{SynComps}[@T] : \text{Syn} \xrightarrow{t} [\text{Syn}]\text{-set}$

 - .1 $\text{SynComps}(d) \triangleq$
 - .2 $\{s \mid \text{mk-}(s, -) \in \text{TransRefSynSubComp}[\text{Syn}](d) \cdot \exists x : @T \cdot x = s\};$

406.0 $\text{TransRefSynSubComp}[@T] : \text{Syn} \rightarrow \text{SynCompRel}$

 - .1 $\text{TransRefSynSubComp}(s) \triangleq$
 - .2 $\text{ReflExtension}[\text{Syn}](\text{TransClosure}[\text{Syn}](\text{SynSubComp}[@T](s)));$

407.0 $\text{SynSubComp}[@T] : \text{Syn} \xrightarrow{t} \text{SynCompRel}$

 - .1 $\text{SynSubComp}(s) \triangleq$
 - .2 $\text{if } \forall x : @T \cdot x \neq s \text{ then } \{\}$
 - .3 $\text{else let } \text{rel} = \text{SynComp}(s) \text{ in}$
 - .4 $\text{rel} \cup \bigcup \{\text{SynSubComp}[@T](c) \mid \text{mk-}(c, -) \in \text{rel}\};$

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 $TypeRTypRel(s) \cup$
- .9 $StmtSynComp(s) \cup$
- .10 $AssignDefSynComp(s) \cup$
- .11 $CallSynComp(s) \cup$
- .12 $StateDesignatorSynComp(s);$

409.0 $ValFnOpDefSynComp(d : Syn) screl : SynCompRel$

- .1 **post** $\forall sc : Syn .$
- .2 $mk-(sc, d) \in screl \Leftrightarrow$
- .3 **cases** $d :$
- .4 $mk\text{-}ValueDef(pat, tp, val) \rightarrow sc \in \{pat, tp, val\},$
- .5 $mk\text{-}ExplFnDef(i_1, tvs, tp, i_2, parms, body, fnpre) \rightarrow$
 $sc \in \{i_1, i_2, tp, body, fnpre\} \cup \text{elems conc } parms \cup \text{elems tvs},$
- .6 $mk\text{-}ImplFnDef(i, tvs, partps, idtp, fnpre, fnpost) \rightarrow$
 $sc \in \{i, idtp.id, idtp.type, fnpre, fnpost\} \cup \text{elems tvs} \cup$
 $\cup \{\text{elems pt.pat} \cup \{pt.type\} \mid pt \in \text{elems partps}\},$
- .7 $mk\text{-}ExplOprtDef(i_1, tp, i_2, parms, body, oppre) \rightarrow$
 $sc \in \{i_1, tp, i_2, body, oppre\} \cup \text{elems params},$
- .8 $mk\text{-}ImplOprtDef(i, partps, idtp, exts, oppre, ooppst, excps) \rightarrow$
 $sc \in \{oppre, ooppst\} \cup$
- .9 $\text{if } idtp \neq \text{nil} \text{ then } \{idtp.id, idtp.type\} \text{ else } \{\} \cup$
 $\cup \{\text{elems ps} \cup \{t\} \mid mk\text{-}PatTypePair(ps, t) \in \text{elems partps}\} \cup$
- .10 $\cup \{\text{elems vs} \cup \{t\} \mid mk\text{-}VarInf(-, vs, t) \in \text{elems exts}\} \cup$
 $\cup \{\{i, c, a\} \mid mk\text{-}Error(i, c, a) \in \text{elems excps}\},$
- .11 $\text{others} \rightarrow \text{false}$
- .12 **end ;**

410.0 $BindSynComp(b : Syn) screl : SynCompRel$

- .1 **post** $\forall sc : Syn .$
- .2 $mk-(sc, b) \in screl \Leftrightarrow$
- .3 **cases** $b :$
- .4 $mk\text{-}SetBind(pat, bset) \rightarrow sc \in \{pat, bset\},$
- .5 $mk\text{-}TypeBind(pat, type) \rightarrow sc \in \{pat, type\},$
- .6 $\text{others} \rightarrow \text{false}$
- .7 **end ;**

411.0 $GenExprSynComp(e : Syn) screl : SynCompRel$

```

.1  post  $\forall sc : Syn .$ 
.2       $mk\text{-}(sc, e) \in screl \Leftrightarrow$ 
.3      cases  $e :$ 
.4           $mk\text{-BracketedExpr}((sc)) \rightarrow \text{true},$ 
.5           $mk\text{-DefExpr}(defs, body) \rightarrow$ 
.6               $sc = body \vee$ 
.7               $\exists mk\text{-DefBind}(pb, rhs) \in \text{elems } defs .$ 
.8                   $sc \in \{pb, rhs\},$ 
.9                   $mk\text{-LetExpr}(ldefs, body) \rightarrow$ 
.10                  $sc = body \vee sc \in \text{elems } ldefs,$ 
.11                  $mk\text{-LetBeSTExpr}(bind, cond, body) \rightarrow$ 
.12                  $sc \in \{bind, cond, body\},$ 
.13                  $mk\text{-IfExpr}(test, cons, altns, altn) \rightarrow$ 
.14                  $sc \in \{test, cons, altn\} \vee$ 
.15                  $\exists mk\text{-ElsifExpr}(test, cons) \in \text{elems } altns .$ 
.16                  $sc \in \{test, cons\},$ 
.17                  $mk\text{-CasesExpr}(sel, altns, other) \rightarrow$ 
.18                  $sc \in \{sel, other\} \vee$ 
.19                  $\exists mk\text{-CaseAltn}(ps, body) \in \text{elems } altns .$ 
.20                  $sc \in \{body\} \cup \text{elems } ps,$ 
.21                  $mk\text{-PrefixExpr}(\text{-}, (sc)) \rightarrow \text{true},$ 
.22                  $mk\text{-BinaryExpr}(left, \text{-}, right) \rightarrow sc \in \{left, right\},$ 
.23                 others  $\rightarrow \text{false}$ 
.24             end ;

```

412.0 $SpecExprSynComp(e : Syn) screl : SynCompRel$

```

.1  post  $\forall sc : Syn .$ 
.2     $mk\text{-}(sc, e) \in screl \Leftrightarrow$ 
.3    cases  $e :$ 
.4       $mk\text{-}MapInverseExpr(sc) \rightarrow \text{true},$ 
.5       $mk\text{-}ExistsExpr(bs, p), mk\text{-}ExistsUniqueExpr(bs, p) \rightarrow$ 
.6         $sc \in \{p\} \cup \text{elems } bs,$ 
.7         $mk\text{-}AllExpr(bs, p), mk\text{-}IotaExpr(bs, p) \rightarrow$ 
.8         $sc \in \{p\} \cup \text{elems } bs,$ 
.9         $mk\text{-}SetEnumeration(es) \rightarrow sc \in \text{elems } es,$ 
.10        $mk\text{-}SetComprehension(e, bs, p) \rightarrow$ 
.11        $sc \in \{e, p\} \cup \text{elems } bs,$ 
.12        $mk\text{-}SetRange(lb, ub) \rightarrow sc \in \{lb, ub\},$ 
.13        $mk\text{-}SeqEnumeration(es) \rightarrow sc \in \text{elems } es,$ 
.14        $mk\text{-}SeqComprehension(e, b, p) \rightarrow$ 
.15        $sc \in \{e, b, p\},$ 
.16        $mk\text{-}SubSequence(s, lb, ub) \rightarrow sc \in \{s, lb, ub\},$ 
.17        $mk\text{-}MapEnumeration(mls) \rightarrow$ 
.18          $\exists mk\text{-}Maplet(d, r) \in \text{elems } mls \cdot sc \in \{d, r\},$ 
.19          $mk\text{-}MapComprehension(mk\text{-}Maplet(d, r), bs, p) \rightarrow$ 
.20          $sc \in \{d, r, p\} \cup \text{elems } bs,$ 
.21          $mk\text{-}TupleConstructor(es) \rightarrow sc \in \text{elems } es,$ 
.22          $mk\text{-}RecordConstructor(i, es) \rightarrow sc \in \{i\} \cup \text{elems } es,$ 
.23          $mk\text{-}RecordModifier(rec, rms) \rightarrow$ 
.24            $sc \in \{rec\} \cup$ 
.25            $\bigcup \{\{f, new\} \mid mk\text{-}RecordModification(f, new) \in \text{elems } rms\},$ 
.26            $mk\text{-}Apply(f, args) \rightarrow sc \in \{f\} \cup \text{elems } args,$ 
.27            $mk\text{-}FieldSelect(r, f) \rightarrow sc \in \{r, f\},$ 
.28            $mk\text{-}Lambda(tbs, body) \rightarrow sc \in \{body\} \cup \text{elems } tbs,$ 
.29            $mk\text{-}IsDefTypeExpr(-, (sc)) \rightarrow \text{true},$ 
.30            $mk\text{-}IsBasicTypeExpr(-, (sc)) \rightarrow \text{true},$ 
.31            $mk\text{-}Name(i), mk\text{-}OldName(i) \rightarrow sc = i,$ 
.32           others  $\rightarrow \text{false}$ 
.33       end ;

```

413.0 $StmtSynComp(s : Syn) \ screl : SynCompRel$

```

.1 post  $\forall sc : Syn$  .
.2      $mk-(sc, s) \in screl \Leftrightarrow$ 
.3     cases  $s$  :
.4          $mk\text{-BlockStmt}(ds, ss) \rightarrow sc \in \text{elems } ds \cup \text{elems } ss$ ,
.5          $mk\text{-AlwaysStmt}(alwpost, body) \rightarrow sc \in \{alwpost, body\}$ ,
.6          $mk\text{-TrapStmt}(pat, trappost, body) \rightarrow sc \in \{pat, trappost, body\}$ ,
.7          $mk\text{-RecTrapStmt}(traps, body) \rightarrow$ 
.8              $(\exists mk\text{-Trap}(match, trappost) \in \text{elems } traps \cdot sc \in \{match, trappost\}) \vee$ 
.9              $sc = body$ ,
.10             $mk\text{-DefStmt}(eqdefs, body) \rightarrow$ 
.11                 $(\exists mk\text{-EqDef}(lhs, rhs) \in \text{elems } eqdefs \cdot sc \in \{lhs, rhs\}) \vee$ 
.12                 $sc = body$ ,
.13             $mk\text{-LetStmt}(letdefs, body) \rightarrow sc \in \text{elems } letdefs \cup \{body\}$ ,
.14             $mk\text{-LetBeSTStmt}(bind, cond, body) \rightarrow sc \in \{bind, cond, body\}$ ,
.15             $mk\text{-AssignStmt}(lhs, rhs) \rightarrow sc \in \{lhs, rhs\}$ ,
.16             $mk\text{-NonDetStmt}(ss) \rightarrow sc \in \text{elems } ss$ ,
.17             $mk\text{-SeqForLoop}(cv, d, forseq, body) \rightarrow sc \in \{cv, d, forseq, body\}$ ,
.18             $mk\text{-SetForLoop}(cv, forset, body) \rightarrow sc \in \{cv, forset, body\}$ ,
.19             $mk\text{-IndexForLoop}(cv, lb, ub, step, body) \rightarrow sc \in \{cv, lb, ub, step, body\}$ ,
.20             $mk\text{-WhileLoop}(test, body) \rightarrow sc \in \{test, body\}$ ,
.21             $mk\text{-IfStmt}(test, cons, elsifaltn, altn) \rightarrow$ 
.22                 $sc \in \{test, cons, altn\} \vee$ 
.23                 $\exists mk\text{-ElsifStmt}(test, cons) \in \text{elems } elsifaltn \cdot sc \in \{test, cons\}$ ,
.24             $mk\text{-CasesStmt}(sel, altns, other) \rightarrow$ 
.25                 $sc \in \{sel, other\} \vee$ 
.26                 $\exists mk\text{-CaseStmtAltn}(match, body) \in \text{elems } altns \cdot$ 
.27                     $sc \in \text{elems } match \cup \{body\}$ ,
.28             $mk\text{-ReturnStmt}((sc)) \rightarrow \text{true}$ ,
.29             $mk\text{-ExitStmt}((sc)) \rightarrow \text{true}$ ,
.30            others  $\rightarrow \text{false}$ 
.31        end ;

```

414.0 $AssignDefSynComp(s : Syn) screl : SynCompRel$

```

.1 post  $\forall sc : Syn$  .
.2      $mk-(sc, s) \in screl \Leftrightarrow$ 
.3     cases  $s$  :
.4          $mk\text{-AssignDef}(v, tp, dcli) \rightarrow sc \in \{v, tp, dcli\}$ ,
.5         others  $\rightarrow \text{false}$ 
.6     end ;

```

415.0 $CallSynComp(s : Syn) screl : SynCompRel$

```

.1 post  $\forall sc : Syn$  .
.2      $mk-(sc, s) \in screl \Leftrightarrow$ 
.3     cases  $s$  :
.4          $mk\text{-Call}(n, el, std) \rightarrow sc \in \{n, el, std\}$ ,
.5         others  $\rightarrow \text{false}$ 
.6     end ;

```

416.0 $StateDesignatorSynComp(s : Syn) screl : SynCompRel$

```

.1  post  $\forall sc : Syn .$ 
.2       $mk-(sc, s) \in screl \Leftrightarrow$ 
.3      cases  $s :$ 
.4           $mk\text{-}MapOrSeqRef(v, e) \rightarrow sc \in \{v, e\},$ 
.5           $mk\text{-}FieldRef(v, i) \rightarrow sc \in \{v, i\},$ 
.6          others  $\rightarrow$  false
.7      end ;
417.0   $PatSynComp(p : Syn) screl : SynCompRel$ 
.1  post  $\forall sc : Syn .$ 
.2       $mk-(sc, p) \in screl \Leftrightarrow$ 
.3      cases  $p :$ 
.4           $mk\text{-}PatternId((sc)) \rightarrow \text{true},$ 
.5           $mk\text{-}MatchVal((sc)) \rightarrow \text{true},$ 
.6           $mk\text{-}SetEnumPattern(ps) \rightarrow sc \in \text{elems } ps,$ 
.7           $mk\text{-}SetUnionPattern(lp, rp) \rightarrow sc \in \{lp, rp\},$ 
.8           $mk\text{-}SeqEnumPattern(ps) \rightarrow sc \in \text{elems } ps,$ 
.9           $mk\text{-}SeqConcPattern(lp, rp) \rightarrow sc \in \{lp, rp\},$ 
.10          $mk\text{-}TuplePattern(ps) \rightarrow sc \in \text{elems } ps,$ 
.11          $mk\text{-}RecordPattern(i, ps) \rightarrow sc \in \{i\} \cup \text{elems } ps,$ 
.12         others  $\rightarrow$  false
.13     end ;
418.0   $TypeRSynComp(t : Syn) screl : SynCompRel$ 
.1  post  $\forall sc : Syn .$ 
.2       $mk-(sc, t) \in screl \Leftrightarrow$ 
.3      cases  $t :$ 
.4           $mk\text{-}InvTypeR(sh, mk\text{-}InvInitFn(p, e)) \rightarrow sc \in \{sh, p, e\},$ 
.5           $mk\text{-}CompositeTypeR(i, fs) \rightarrow$ 
.6               $sc = i \vee \exists mk\text{-}FieldR(f, t) \in \text{elems } fs \cdot sc \in \{f, t\},$ 
.7           $mk\text{-}UnionTypeR(ts) \rightarrow sc \in ts,$ 
.8           $mk\text{-}ProductTypeR(ts) \rightarrow sc \in \text{elems } ts,$ 
.9           $mk\text{-}SetTypeR((sc)) \rightarrow \text{true},$ 
.10          $mk\text{-}Seq0TypeR((sc)) \rightarrow \text{true},$ 
.11          $mk\text{-}Seq1TypeR((sc)) \rightarrow \text{true},$ 
.12          $mk\text{-}GeneralMapTypeR(dt, rt) \rightarrow sc \in \{dt, rt\},$ 
.13          $mk\text{-}InjectiveMapTypeR(dt, rt) \rightarrow sc \in \{dt, rt\},$ 
.14          $mk\text{-}PartialFnTypeR(dt, rt) \rightarrow sc \in \{dt, rt\},$ 
.15          $mk\text{-}TotalFnTypeR(dt, rt) \rightarrow sc \in \{dt, rt\},$ 
.16          $mk\text{-}OpTypeR(i, dt, rt) \rightarrow sc \in \{i, dt, rt\},$ 
.17         others  $\rightarrow$  false
.18     end ;
419.0   $TypeRTypRel : Syn \xrightarrow{t} SynCompRel$ 
.1   $TypeRTypRel(s) \triangleq$ 
.2  if  $IsType(s) \vee is\text{-}OpType(s) \vee is\text{-}TotalFnType(s)$ 
.3  then  $\{mk\text{-}(ExtractTypeR(s)(EmptyEnv), s)\}$ 
.4  else  $\{\}$ 

```

The Definitional Basis of Types

```

420.0   TypeRDefBasis :  $\Pi \xrightarrow{t} TypeR \rightarrow TypeR\text{-set}$ 
.1     TypeRDefBasis( $\pi$ )(type)  $\triangleq$ 
.2       {type}  $\cup$ 
.3       cases type :
.4         mk-InvTypeR(t, -)  $\rightarrow$  if is-DEF( $\pi$ ) then {} else TypeRDefBasis( $\pi$ )(t),
.5         mk-CompositeTypeR(id, fs)  $\rightarrow$ 
.6            $\bigcup \{ TypeRDefBasis(\pi)(t) \mid mk-FieldR(-, t) \in \text{elems } fs \},$ 
.7         mk-UnionTypeR(ts)  $\rightarrow$   $\bigcup \{ TypeRDefBasis(\pi)(t) \mid t \in ts \},$ 
.8         mk-ProductTypeR(ts)  $\rightarrow$   $\bigcup \{ TypeRDefBasis(\pi)(t) \mid t \in \text{elems } ts \},$ 
.9         mk-SetTypeR(t)  $\rightarrow$  TypeRDefBasis( $\pi$ )(t),
.10        mk-Seq0TypeR(t)  $\rightarrow$  TypeRDefBasis( $\pi$ )(t),
.11        mk-Seq1TypeR(t)  $\rightarrow$  TypeRDefBasis( $\pi$ )(t),
.12        mk-GeneralMapTypeR(dt, rt)  $\rightarrow$  TypeRDefBasis( $\pi$ )(dt)  $\cup$  TypeRDefBasis( $\pi$ )(rt),
.13        mk-InjectiveMapTypeR(dt, rt)  $\rightarrow$  TypeRDefBasis( $\pi$ )(dt)  $\cup$  TypeRDefBasis( $\pi$ )(rt),
.14        mk-PartialFnTypeR(dt, rt)  $\rightarrow$  TypeRDefBasis( $\pi$ )(dt)  $\cup$  TypeRDefBasis( $\pi$ )(rt),
.15        mk-TotalFnTypeR(dt, rt)  $\rightarrow$  TypeRDefBasis( $\pi$ )(dt)  $\cup$  TypeRDefBasis( $\pi$ )(rt),
.16        others  $\rightarrow$  {}
.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.

11.7.2 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 :  $LambdaR \xrightarrow{t}$ 
.1     Parameters+  $\times$  FunctionTypeR  $\times$  [Expr]  $\times$  Expr  $\rightarrow$  Env  $\rightarrow$   $\mathbb{B}$ 
.2     IsCurriedFn(mk-LambdaR(id, argtp, body1, fnpref1, tot))(patseqseq, tp, fnpref2, body2)(env)  $\triangleq$ 
.3       let pat = Pats2Pat(hd patseqseq) in
.4        $\exists xbody : Expr .$ 
.5         tot = is-TotalFnTypeR(tp)  $\wedge$ 
.6         argtp = tp.fndom  $\wedge$ 
.7         IsLetExpansion(fnpref1)(pat, nil, mk-Name(id), fnpref2)(env)  $\wedge$ 
.8         IsLetExpansion(xbody)(pat, nil, mk-Name(id), body2)(env)  $\wedge$ 
.9         if len patseqseq = 1
.10        then body1 = xbody
.11        else Is[FunctionTypeR, TypeR](tp.fnrng)  $\wedge$ 
.12          IsCurriedFn(body1)(tl patseqseq, tp.fnrng, nil, xbody)(env);

```

422.0 $\text{IsLetExpandedParTypes} : [\text{Stmt} \mid \text{Expr}] \xrightarrow{t} \text{ParameterTypes} \times \text{Name} \times [\text{Stmt} \mid \text{Expr}] \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsLetExpandedParTypes}(\text{pts}, n, \text{body})(\text{env}) \triangleq$
 .2 $\text{let } p = \text{ParameterTypes2Pattern}(\text{pts}),$
 .3 $t = \text{ParameterTypes2Type}(\text{pts}) \text{ in}$
 .4 $\text{IsLetExpansion}(\text{letse})(p, t, n, \text{body})(\text{env});$
 .5

 423.0 $\text{IsLetExpansion} : [\text{Stmt} \mid \text{Expr}] \xrightarrow{t} \text{Pattern} \times [\text{TypeR}] \times \text{Name} \times [\text{Stmt} \mid \text{Expr}] \rightarrow \text{Env} \rightarrow \mathbb{B}$
 .1 $\text{IsLetExpansion}(\text{letse})(p, t, n, \text{body})(\text{env}) \triangleq$
 .2 $\text{let } \text{bodynames} = \text{if } \text{Is}[\text{Stmt}, [\text{Stmt} \mid \text{Expr}]](\text{body})$
 .3 $\text{then FreeNamesInStmt}(\text{body})$
 .4 $\text{else FreeNamesInExpr}(\text{body}) \text{ in}$
 .5 $n \notin \text{dom env.visibleenv} \cup \text{StateVarNames}(\text{env}) \cup \text{bodynames} \wedge$
 .6 $\{\text{letse, body}\} = \{\text{nil}\} \vee$
 .7 $\text{if Is}[\text{Stmt}, [\text{Stmt} \mid \text{Expr}]](\text{body})$
 .8 $\text{then letse} = \text{mk-LetExpr}([\text{mk-ValueDef}(p, t, n)], \text{body})$
 .9 $\text{else letse} = \text{mk-LetStmt}([\text{mk-ValueDef}(p, t, n)], \text{body});$
 .10

 424.0 $\text{ParameterTypes2Pattern} : \text{ParameterTypes} \xrightarrow{t} \text{Pattern}$
 .1 $\text{ParameterTypes2Pattern}(\text{pts}) \triangleq$
 .2 $\text{Pats2Pat}(\text{ParameterTypes2Pats}(\text{pts}));$

 425.0 $\text{ParameterTypes2Pats} : \text{ParameterTypes} \xrightarrow{t} \text{Pattern}^*$
 .1 $\text{ParameterTypes2Pats}(\text{pts}) \triangleq$
 .2 $\text{conc} [[\text{pts}(i).\text{pat}(j) \mid j \in \text{inds pts}(i).\text{pat}] \mid i \in \text{inds pts}];$

 426.0 $\text{Pats2Pat} : \text{Pattern}^* \xrightarrow{t} \text{Pattern}$
 .1 $\text{Pats2Pat}(\text{pats}) \triangleq$
 .2 $\text{cases len pats} :$
 .3 $(0) \rightarrow \text{mk-MatchVal}(\text{mk-NullLit}()),$
 .4 $(1) \rightarrow \text{hd pats},$
 .5 $\text{others} \rightarrow \text{mk-TuplePattern}(\text{pats})$
 .6 $\text{end};$

 427.0 $\text{ParameterTypes2Type} : \text{ParameterTypes} \xrightarrow{t} \text{Type}$
 .1 $\text{ParameterTypes2Type}(\text{pts}) \triangleq$
 .2 $\text{Types2Type}(\text{ParameterTypes2Types}(\text{pts}));$

 428.0 $\text{ParameterTypes2Types} : \text{ParameterTypes} \xrightarrow{t} \text{Type}^*$
 .1 $\text{ParameterTypes2Types}(\text{pts}) \triangleq$
 .2 $[\text{pts}(i).\text{type} \mid i \in \text{inds pts}];$

 429.0 $\text{Types2Type} : \text{Type}^* \xrightarrow{t} \text{Type}$
 .1 $\text{Types2Type}(\text{ts}) \triangleq$
 .2 $\text{cases len ts} :$
 .3 $(0) \rightarrow \text{UNITTYPE},$
 .4 $(1) \rightarrow \text{hd ts},$
 .5 $\text{others} \rightarrow \text{mk-ProductType}(\text{ts})$
 .6 $\text{end};$

```

430.0    $OType2Types : [Type] \xrightarrow{t} Type^*$ 
.1    $OType2Types(t) \triangleq$ 
.2   cases t :
.3   (nil)  $\rightarrow []$ ,
.4   mk-ProductType(ts)  $\rightarrow ts$ ,
.5   others  $\rightarrow [t]$ 
.6   end;

431.0    $Exprs2Expr : Expr^* \xrightarrow{t} Expr$ 
.1    $Exprs2Expr(es) \triangleq$ 
.2   cases len es :
.3   (0)  $\rightarrow mk-NilLit()$ ,
.4   (1)  $\rightarrow hd\ es$ ,
.5   others  $\rightarrow mk-TupleConstructor(es)$ 
.6   end;

432.0    $PreOExceptions2Pre : [Expr] \times OExceptions \xrightarrow{t} [Expr]$ 
.1    $PreOExceptions2Pre(oppre, opcps) \triangleq$ 
.2   let oppre-s = if oppre = nil then {} else {oppre} in
.3   GetOrExpr(oppre-s  $\cup \{opcps(i).cond \mid i \in \text{inds } opcps\}$ );

433.0    $PostOExceptions2Post : Expr \times OExceptions \xrightarrow{t} Expr$ 
.1    $PostOExceptions2Post(oppost, opcps) \triangleq$ 
.2   let err-post-s = {mk-BinaryExpr(cond, AND, action) |
.3     mk-Error(-, cond, action)  $\in \text{elems } opcps\}$ ,
.4   not-err-pre = GetOrExpr({opcps(i).cond  $\mid i \in \text{inds } opcps\}$ ),
.5   post1 = mk-BinaryExpr(oppost, AND, mk-PrefixExpr(NOT, not-err-pre)) in
.6   GetOrExpr({post1}  $\cup err-post-s$ );

434.0    $GetOrExpr : Expr\text{-set} \xrightarrow{t} [Expr]$ 
.1    $GetOrExpr(es) \triangleq$ 
.2   if es = {}
.3   then nil
.4   elseif card es = 1 then let e  $\in$  es in e
.5   else let e  $\in$  es in mk-BinaryExpr(e, OR, GetOrExpr(es \ {e}))

```

11.7.3 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.

```

435.0   $TypeSubst : TypeR \xrightarrow{m} TypeR \xrightarrow{t} TypeR \rightarrow TypeR$ 
.1    $TypeSubst(tm)(t) \triangleq$ 
.2   if  $t \in \text{dom } tm$ 
.3   then  $TypeSubst(\{t\} \triangleleft tm)(tm(t))$ 
.4   else cases  $t$  :
.5      $mk\text{-}InvTypeR(t_1, inva) \rightarrow mk\text{-}InvTypeR(TypeSubst(tm)(t_1), inva),$ 
.6      $mk\text{-}CompositeTypeR(id, fields) \rightarrow$ 
.7     let  $newfields =$ 
.8       [let  $mk\text{-}FieldR(fid, ft) = fields(i)$  in
.9         $mk\text{-}FieldR(fid, TypeSubst(tm)(ft)) \mid i \in \text{inds } fields$ ] in
.10       $mk\text{-}CompositeTypeR(id, newfields),$ 
.11       $mk\text{-}UnionTypeR(summands) \rightarrow$ 
.12       $mk\text{-}UnionTypeR(\{TypeSubst(tm)(sumt) \mid sumt \in summands\}),$ 
.13       $mk\text{-}ProductTypeR(factors) \rightarrow$ 
.14       $mk\text{-}ProductTypeR([TypeSubst(tm)(factors(i)) \mid i \in \text{inds } factors]),$ 
.15       $mk\text{-}SetTypeR(et) \rightarrow mk\text{-}SetTypeR(TypeSubst(tm)(et)),$ 
.16       $mk\text{-}Seq0TypeR(et) \rightarrow mk\text{-}Seq0TypeR(TypeSubst(tm)(et)),$ 
.17       $mk\text{-}Seq1TypeR(et) \rightarrow mk\text{-}Seq1TypeR(TypeSubst(tm)(et)),$ 
.18       $mk\text{-}GeneralMapTypeR(dt, rt) \rightarrow$ 
.19       $mk\text{-}GeneralMapTypeR(TypeSubst(tm)(dt), TypeSubst(tm)(rt)),$ 
.20       $mk\text{-}InjectiveMapTypeR(dt, rt) \rightarrow$ 
.21       $mk\text{-}InjectiveMapTypeR(TypeSubst(tm)(dt), TypeSubst(tm)(rt)),$ 
.22       $mk\text{-}PartialFnTypeR(dt, rt) \rightarrow$ 
.23       $mk\text{-}PartialFnTypeR(TypeSubst(tm)(dt), TypeSubst(tm)(rt)),$ 
.24       $mk\text{-}TotalFnTypeR(dt, rt) \rightarrow$ 
.25       $mk\text{-}TotalFnTypeR(TypeSubst(tm)(dt), TypeSubst(tm)(rt)),$ 
.26       $mk\text{-}RetTypeR(et) \rightarrow mk\text{-}RetTypeR(TypeSubst(tm)(et)),$ 
.27       $mk\text{-}ExitTypeR(et) \rightarrow mk\text{-}ExitTypeR(TypeSubst(tm)(et)),$ 
.28       $mk\text{-}OpTypeR(stt, dt, rt) \rightarrow$ 
.29       $mk\text{-}OpTypeR(TypeSubst(tm)(stt), TypeSubst(tm)(dt), TypeSubst(tm)(rt)),$ 
.30      others  $\rightarrow t$ 
.31    end;

```

436.0 $ONInExprsSubst : OldName \xrightarrow{m} Name \rightarrow Expr^* \rightarrow Expr^*$

- .1 $ONInExprsSubst(nm)(es) \triangleq$
- .2 $[ONInExprSubst(nm)(es(i)) \mid i \in \text{inds } es];$

437.0 $ONInExprSubst : OldName \xrightarrow{m} Name \xrightarrow{t} Expr \rightarrow Expr$

.1 $ONInExprSubst(nm)(e) \triangleq$
 .2 if $e \in \text{dom } nm$ then $nm(e)$
 .3 else let $S = ONInExprSubst(nm)$ in
 .4 cases e :
 .5 $mk\text{-BracketedExpr}(e_1) \rightarrow mk\text{-BracketedExpr}(S(e_1)),$
 .6 $mk\text{-DefExpr}(defs, body) \rightarrow$
 .7 $mk\text{-DefExpr}(ONInDefBindsSubst(nm)(defs), S(body)),$
 .8 $mk\text{-LetExpr}(ldefs, body) \rightarrow$
 .9 $mk\text{-LetExpr}(ONInLocalDefsSubst(nm)(ldefs), S(body)),$
 .10 $mk\text{-LetBeSE Expr}(bind, cond, body) \rightarrow$
 .11 $mk\text{-LetBeSE Expr}(ONInBindSubst(nm)(bind), S(cond), S(body)),$
 .12 $mk\text{-IfExpr}(test, cons, altns, altn) \rightarrow$
 .13 $mk\text{-IfExpr}(S(test), S(cons),$
 .14 $[mk\text{-ElsifExpr}(S(altns(i).test), S(altns(i).cons)) \mid i \in \text{inds } altns],$
 .15 $S(altn)),$
 .16 $mk\text{-CasesExpr}(sel, altns, other) \rightarrow$
 .17 $mk\text{-CasesExpr}(S(sel),$
 .18 $[mk\text{-CaseAltn}(ONInPatsSubst(nm)(altns(i).match), S(altns(i).body)) \mid$
 .19 $i \in \text{inds } altns],$
 .20 $S(other)),$
 .21 $mk\text{-PrefixExpr}(op, e_1) \rightarrow mk\text{-PrefixExpr}(op, S(e_1)),$
 .22 $mk\text{-BinaryExpr}(left, op, right) \rightarrow mk\text{-BinaryExpr}(S(left), op, S(right)),$
 .23 others $\rightarrow ONInSpecExprSubst(nm)(e)$
 .24 end;

438.0 $ONInSpecExprSubst : OldName \xrightarrow{m} Name \xrightarrow{t} Expr \rightarrow Expr$

.1 $ONInSpecExprSubst(nm)(e) \triangleq$
 .2 let $S = ONInExprSubst(nm)$ in
 .3 cases e :
 .4 $mk\text{-}MapInverseExpr(e_1) \rightarrow mk\text{-}MapInverseExpr(S(e_1)),$
 .5 $mk\text{-}ExistsExpr(bs, p) \rightarrow$
 $mk\text{-}ExistsExpr(ONInBindsSubst(nm)(bs), S(p)),$
 .6 $mk\text{-}ExistsUniqueExpr(bs, p) \rightarrow$
 $mk\text{-}ExistsUniqueExpr(ONInBindSubst(nm)(bs), S(p)),$
 .7 $mk\text{-}AllExpr(bs, p) \rightarrow$
 $mk\text{-}AllExpr(ONInBindsSubst(nm)(bs), S(p)),$
 .8 $mk\text{-}IotaExpr(bs, p) \rightarrow$
 $mk\text{-}IotaExpr(ONInBindSubst(nm)(bs), S(p)),$
 .9 $mk\text{-}SetEnumeration(es) \rightarrow mk\text{-}SetEnumeration(ONInExprsSubst(nm)(es)),$
 .10 $mk\text{-}SetComprehension(e, bs, p) \rightarrow$
 $mk\text{-}SetComprehension(S(e), ONInBindsSubst(nm)(bs), S(p)),$
 .11 $mk\text{-}SetRange(lb, ub) \rightarrow mk\text{-}SetRange(S(lb), S(ub)),$
 .12 $mk\text{-}SeqEnumeration(es) \rightarrow mk\text{-}SeqEnumeration(ONInExprsSubst(nm)(es)),$
 .13 $mk\text{-}SeqComprehension(e, b, p) \rightarrow$
 $mk\text{-}SeqComprehension(S(e), ONInBindSubst(nm)(b), S(p)),$
 .14 $mk\text{-}SubSequence(s, lb, ub) \rightarrow mk\text{-}SubSequence(S(s), S(lb), S(ub)),$
 .15 $mk\text{-}MapEnumeration(mls) \rightarrow$
 $mk\text{-}MapEnumeration([mk\text{-}Maplet(S(mls(i).mapdom), S(mls(i).maprng)) |$
 .16 $i \in \text{inds } mls]),$
 .17 $mk\text{-}MapComprehension(mk\text{-}Maplet(d, r), bs, p) \rightarrow$
 $mk\text{-}MapComprehension(mk\text{-}Maplet(S(d), S(r)), ONInBindsSubst(nm)(bs), S(p)),$
 .18 $mk\text{-}TupleConstructor(es) \rightarrow mk\text{-}TupleConstructor(ONInExprsSubst(nm)(es)),$
 .19 $mk\text{-}RecordConstructor(t, es) \rightarrow mk\text{-}RecordConstructor(t, ONInExprsSubst(nm)(es)),$
 .20 $mk\text{-}RecordModifier(rec, rms) \rightarrow$
 $mk\text{-}RecordModifier(S(rec),$
 $[mk\text{-}RecordModification(rms(i).field, S(rms(i).new)) | i \in \text{inds } rms]),$
 .21 $mk\text{-}Apply(e_1, es) \rightarrow mk\text{-}Apply(S(e_1), ONInExprsSubst(nm)(es)),$
 .22 $mk\text{-}FieldSelect(e_1, f) \rightarrow mk\text{-}FieldSelect(S(e_1), f),$
 .23 $mk\text{-}Lambda(tbs, body) \rightarrow mk\text{-}Lambda(ONInTypeBindsSubst(nm)(tbs), S(body)),$
 .24 $mk\text{-}IsDefTypeExpr(dt, e_1) \rightarrow mk\text{-}IsDefTypeExpr(dt, S(e_1)),$
 .25 $mk\text{-}IsBasicTypeExpr(bt, e_1) \rightarrow mk\text{-}IsBasicTypeExpr(bt, S(e_1)),$
 .26 others $\rightarrow e$
 .27 end;
 .28

439.0 $ONInDefsSubst : OldName \xrightarrow{m} Name \rightarrow EqDef^+ \rightarrow EqDef^+$

.1 $ONInDefsSubst(nm)(eqdefs) \triangleq$
 .2 $[mk\text{-}EqDef(ONInPatBindSubst(nm)(eqdefs(i).lhs),$
 .3 $ONInExprOrCallSubst(nm)(eqdefs(i).rhs)) |$
 .4 $i \in \text{inds } eqdefs];$

440.0 $ONInDefBindsSubst : OldName \xrightarrow{m} Name \rightarrow DefBind^* \rightarrow DefBind^*$

.1 $ONInDefBindsSubst(nm)(dbs) \triangleq$
 .2 $[mk\text{-}DefBind(ONInPatBindSubst(nm)(dbs(i).lhs),$
 .3 $ONInExprSubst(nm)(dbs(i).rhs)) |$
 .4 $i \in \text{inds } dbs];$

- 441.0 $ONInPatBindSubst : OldName \xrightarrow{m} Name \rightarrow PatternBind \rightarrow PatternBind$
- .1 $ONInPatBindSubst(nm)(pb) \triangleq$
 - .2 if $Is[Pattern, PatternBind](pb)$ then $ONInPatternSubst(nm)(pb)$
 - .3 else $ONInBindSubst(nm)(pb);$
- 442.0 $ONInExprOrCallSubst : OldName \xrightarrow{m} Name \rightarrow Expr \mid Call \rightarrow Expr \mid Call$
- .1 $ONInExprOrCallSubst(nm)(ec) \triangleq$
 - .2 if $Is[Expr, Expr \mid Call](ec)$ then $ONInExprSubst(nm)(ec)$
 - .3 else $\mu(ec, args \mapsto ONInExprsSubst(nm)(ec.args));$
- 443.0 $ONInOExprSubst : OldName \xrightarrow{m} Name \rightarrow [Expr] \rightarrow [Expr]$
- .1 $ONInOExprSubst(nm)(oe) \triangleq$
 - .2 if $oe = \text{nil}$ then nil
 - .3 else $ONInExprSubst(nm)(oe);$
- 444.0 $ONInLocalDefsSubst : OldName \xrightarrow{m} Name \rightarrow LocalDef^+ \rightarrow LocalDef^+$
- .1 $ONInLocalDefsSubst(nm)(ds) \triangleq$
 - .2 $[ONInLocalDefSubst(nm)(ds(i)) \mid i \in \text{inds } ds];$
- 445.0 $ONInLocalDefSubst : OldName \xrightarrow{m} Name \rightarrow LocalDef \rightarrow LocalDef$
- .1 $ONInLocalDefSubst(nm)(d) \triangleq$
 - .2 cases $d :$
 - .3 $mk\text{-ValueDef}(p, t, v) \rightarrow$
 $mk\text{-ValueDef}(ONInPatternSubst(nm)(p), t, ONInExprSubst(nm)(v)),$
 - .4 $mk\text{-ExplFnDef}(i_1, tv, t, i_2, p, b, pr) \rightarrow$
 $mk\text{-ExplFnDef}(i_1, tv, t, i_2,$
 $ONInParListSubst(nm)(p), ONInExprSubst(nm)(b), ONInOExprSubst(nm)(pr)),$
 - .5 $mk\text{-ImplFnDef}(i, tv, pt, it, fpre, fpost) \rightarrow$
 $mk\text{-ImplFnDef}(i, tv, ONInParTypesSubst(nm)(pt), it,$
 $ONInOExprSubst(nm)(fpre), ONInExprSubst(nm)(fpost))$
 - .6 end;
- 446.0 $ONInParTypesSubst : OldName \xrightarrow{m} Name \rightarrow PatTypePair^* \rightarrow PatTypePair^*$
- .1 $ONInParTypesSubst(nm)(pts) \triangleq$
 - .2 $[\mu(pts(i), pat \mapsto ONInPatsSubst(nm)(pts(i).pat)) \mid i \in \text{inds } pts];$
- 447.0 $ONInPatsSubst : OldName \xrightarrow{m} Name \rightarrow Pattern^* \rightarrow Pattern^*$
- .1 $ONInPatsSubst(nm)(ps) \triangleq$
 - .2 $[ONInPatternSubst(nm)(ps(i)) \mid i \in \text{inds } ps];$
- 448.0 $ONInParListSubst : OldName \xrightarrow{m} Name \rightarrow ParametersList \rightarrow ParametersList$
- .1 $ONInParListSubst(nm)(pl) \triangleq$
 - .2 $[ONInPatsSubst(nm)(pl(i)) \mid i \in \text{inds } pl];$

449.0 $ONInPatternSubst : OldName \xrightarrow{m} Name \rightarrow Pattern \rightarrow Pattern$
 .1 $ONInPatternSubst(nm)(p) \triangleq$
 .2 let $S = ONInPatternSubst(nm)$,
 .3 $Ss = ONInPatsSubst(nm)$ in
 .4 cases p :
 .5 $mk-MatchVal(e) \rightarrow mk-MatchVal(ONInExprSubst(nm)(e))$,
 .6 $mk-SetEnumPattern(ps) \rightarrow mk-SetEnumPattern(Ss(ps))$,
 .7 $mk-SetUnionPattern(lp, rp) \rightarrow mk-SetUnionPattern(S(lp), S(rp))$,
 .8 $mk-SeqEnumPattern(ps) \rightarrow mk-SeqEnumPattern(Ss(ps))$,
 .9 $mk-SeqConcPattern(lp, rp) \rightarrow mk-SeqConcPattern(S(lp), S(rp))$,
 .10 $mk-TuplePattern(ps) \rightarrow mk-TuplePattern(Ss(ps))$,
 .11 $mk-RecordPattern(i, ps) \rightarrow mk-RecordPattern(i, Ss(ps))$,
 .12 $mk-PatternId(-) \rightarrow p$
 .13 end;

 450.0 $ONInBindsSubst : OldName \xrightarrow{m} Name \rightarrow MultBind^* \rightarrow MultBind^*$
 .1 $ONInBindsSubst(nm)(mbs) \triangleq$
 .2 $[ONInMultBindSubst(nm)(mbs(i)) \mid i \in \text{inds } mbs]$;

 451.0 $ONInTypeBindsSubst : OldName \xrightarrow{m} Name \rightarrow TypeBind^* \rightarrow TypeBind^*$
 .1 $ONInTypeBindsSubst(nm)(tbs) \triangleq$
 .2 $[ONInBindSubst(nm)(tbs(i)) \mid i \in \text{inds } tbs]$;

 452.0 $ONInMultBindSubst : OldName \xrightarrow{m} Name \rightarrow MultBind \rightarrow MultBind$
 .1 $ONInMultBindSubst(nm)(mb) \triangleq$
 .2 cases mb :
 .3 $mk-MultSetBind(ps, e) \rightarrow mk-MultSetBind(ONInPatsSubst(nm)(ps), ONInExprSubst(nm)(e))$,
 .4 $mk-MultTypeBind(ps, t) \rightarrow mk-MultTypeBind(ONInPatsSubst(nm)(ps), t)$
 .5 end;

 453.0 $ONInBindSubst : OldName \xrightarrow{m} Name \rightarrow Bind \rightarrow Bind$
 .1 $ONInBindSubst(nm)(b) \triangleq$
 .2 cases b :
 .3 $mk-SetBind(p, e) \rightarrow mk-SetBind(ONInPatternSubst(nm)(p), ONInExprSubst(nm)(e))$,
 .4 $mk-TypeBind(p, t) \rightarrow mk-TypeBind(ONInPatternSubst(nm)(p), t)$
 .5 end

11.7.4 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 $ExtractFnPrePostDefs : FunctionDef^* \xrightarrow{t} ExplFnDef^*$
 .1 $ExtractFnPrePostDefs(fndefs) \triangleq$
 .2 if $fndefs = []$
 .3 then []
 .4 else let $prefn = \text{if } is-ExplFnDef(\text{hd } fndefs)$
 .5 then $ExtractFnPreDefFromExplDef(\text{hd } fndefs)$
 .6 else $ExtractFnPreDefFromImplDef(\text{hd } fndefs)$,
 .7 $postfns = \text{if } is-ExplFnDef(\text{hd } fndefs)$
 .8 then []
 .9 else $[ExtractFnPostDefFromImplDef(\text{hd } fndefs)]$ in
 .10 $[prefn] \curvearrowright postfns \curvearrowright ExtractFnPrePostDefs(tl fndefs);$

455.0 $\text{ExtractFnPreDefFromExplDef} : \text{ExplFnDef} \xrightarrow{t} \text{ExplFnDef}$
 .1 $\text{ExtractFnPreDefFromExplDef}(\text{mk-ExplFnDef}(id, \text{tpparms}, type, -, parms, -, fnpre)) \triangleq$
 .2 let $idpre = \text{mk-PreId}(\text{GetToken}(id))$,
 .3 $tp = \text{mk-PartialFnType}(type.\text{fndom}, \text{BOOLEAN})$,
 .4 $body = \text{if } fnpre \neq \text{nil}$
 .5 then $fnpre$
 .6 else $\text{mk-BoolLit}(\text{true})$ in
 .7 $\text{mk-ExplFnDef}(idpre, \text{tpparms}, tp, idpre, parms, body, \text{nil})$;

 456.0 $\text{ExtractFnPreDefFromImplDef} : \text{ImplFnDef} \xrightarrow{t} \text{ExplFnDef}$
 .1 $\text{ExtractFnPreDefFromImplDef}(\text{mk-ImplFnDef}(id, \text{tpparms}, partps, -, fnpre, -)) \triangleq$
 .2 let $idpre = \text{mk-PreId}(\text{GetToken}(id))$,
 .3 $tp = \text{mk-PartialFnType}(\text{ParameterTypes2Type}(partps), \text{BOOLEAN})$,
 .4 $body = \text{if } fnpre \neq \text{nil}$
 .5 then $fnpre$
 .6 else $\text{mk-BoolLit}(\text{true})$,
 .7 $parms = \text{ParameterTypes2Pats}(partps)$ in
 .8 $\text{mk-ExplFnDef}(idpre, \text{tpparms}, tp, idpre, [parms], body, \text{nil})$;

 457.0 $\text{ExtractFnPostDefFromImplDef} : \text{ImplFnDef} \xrightarrow{t} \text{ExplFnDef}$
 .1 $\text{ExtractFnPostDefFromImplDef}(\text{mk-ImplFnDef}(id, \text{tpparms}, partps, residtype, -, fnpost)) \triangleq$
 .2 let $idpost = \text{mk-PostId}(\text{GetToken}(id))$,
 .3 $tp = \text{mk-TotalFnType}(\text{Types2Type}(\text{ParameterTypes2Types}(partps) \curvearrowright [\text{residtype.type}]), \text{BOOLEAN})$,
 .4 $parms = \text{ParameterTypes2Pats}(partps)$ in
 .5 $\text{mk-ExplFnDef}(idpost, \text{tpparms}, tp, idpost, [parms \curvearrowright [\text{mk-PatternId}(\text{residtype.id})]], fnpost, \text{nil})$;

 458.0 $\text{ExtractOpPrePostDefs} : \text{OperationDef}^* \times \text{StateDef}^* \times \text{Id-set} \xrightarrow{t} \text{ExplFnDef}^*$
 .1 $\text{ExtractOpPrePostDefs}(\text{opdefs}, \text{stdefs}, \text{used}) \triangleq$
 .2 if $opdefs = []$
 .3 then []
 .4 else let $sttypes = \text{if } \text{StateName}(stdefs) = \text{nil} \text{ then } [] \text{ else } [\text{StateName}(stdefs)]$,
 .5 $\text{mk-(stpats, onsubst)} = \text{StatePats}(stdefs, \text{used})$,
 .6 $\text{prefns} = \text{if } \text{Is}[\text{ExplOprtDef}, \text{OperationDef}](\text{hd opdefs})$
 .7 then $\text{ExtractOpPreDefFromExplDef}(\text{hd opdefs})(sttypes, \text{stpats}(1, \dots, 1))$
 .8 else $\text{ExtractOpPreDefFromImplDef}(\text{hd opdefs})(sttypes, \text{stpats}(1, \dots, 1))$,
 .9 $\text{postfns} = \text{if } \text{Is}[\text{ExplOprtDef}, \text{OperationDef}](\text{hd opdefs})$
 .10 then []
 .11 else $[\text{ExtractOpPostDefFromImplDef}(\text{hd opdefs})(sttypes, \text{stpats}, \text{onsubst})]$ in
 .12 $\text{prefns} \curvearrowright \text{postfns} \curvearrowright \text{ExtractOpPrePostDefs}(\text{tl opdefs}, \text{stdefs}, \text{used})$;

 459.0 $\text{ExtractOpPreDefFromExplDef} : \text{ExplOprtDef} \xrightarrow{t} \text{TypeR}^* \times \text{Pattern}^* \rightarrow \text{ExplFnDef}^*$
 .1 $\text{ExtractOpPreDefFromExplDef}$
 .2 $(\text{mk-ExplOprtDef}(id, \text{mk-OpType}(odom, -, -, params, body, oppre))(sttypes, \text{stpats})) \triangleq$
 .3 let $idpre = \text{mk-PreId}(\text{GetToken}(id))$,
 .4 $tp = \text{mk-PartialFnType}(\text{Types2Type}(OType2Types(odom) \curvearrowright sttypes), \text{BOOLEAN})$,
 .5 $parms = \text{params} \curvearrowright \text{stpats}$,
 .6 $body = \text{if } oppre \neq \text{nil}$
 .7 then $oppre$
 .8 else $\text{mk-BoolLit}(\text{true})$ in
 .9 $[\text{mk-ExplFnDef}(idpre, [], tp, idpre, [parms], body, \text{nil})]$;

460.0 $\text{ExtractOpPreDefFromImplDef} : \text{ImplOprtDef} \xrightarrow{t} \text{TypeR}^* \times \text{Pattern}^* \rightarrow \text{ExplFnDef}^*$
 .1 $\text{ExtractOpPreDefFromImplDef}$
 .2 $(\text{mk-ImplOprtDef}(id, \text{partps}, \text{residtype}, -, \text{oppre}, -, -))(sttypes, stpats) \triangleq$
 .3 let $\text{idpre} = \text{mk-PreId}(\text{GetToken}(id))$,
 .4 $\text{tp} = \text{mk-PartialFnType}(\text{Types2Type}(\text{ParameterTypes2Types}(\text{partps}) \curvearrowright sttypes), \text{BOOLEAN})$,
 .5 $\text{parms} = \text{ParameterTypes2Pats}(\text{partps}) \curvearrowright stpats$,
 .6 $\text{body} = \text{if } \text{oppre} \neq \text{nil}$
 .7 $\text{then } \text{oppre}$
 .8 $\text{else } \text{mk-BoolLit}(\text{true}) \text{ in}$
 .9 $[\text{mk-ExplFnDef}(\text{idpre}, [], \text{tp}, \text{idpre}, [\text{parms}], \text{body}, \text{nil})];$

461.0 $\text{ExtractOpPostDefFromImplDef} :$
 .1 $\text{ImplOprtDef} \xrightarrow{t} \text{TypeR}^* \times \text{Pattern}^* \times (\text{OldName} \xrightarrow{m} \text{Name}) \rightarrow \text{ExplFnDef}^*$
 .2 $\text{ExtractOpPostDefFromImplDef}$
 .3 $(\text{mk-ImplOprtDef}(id, \text{partps}, \text{residtype}, -, -, \text{oppost}, -))(sttypes, stpats, subst) \triangleq$
 .4 let $\text{idpost} = \text{mk-PostId}(\text{GetToken}(id))$,
 .5 $\text{mk-(rpats, rtypes)} = \text{if } \text{residtype} = \text{nil}$
 .6 $\text{then } \text{mk-}([], [])$
 .7 $\text{else } \text{mk-}([\text{mk-PatternId}(\text{residtype.id})], [\text{residtype.type}])$,
 .8 $\text{tp} = \text{mk-PartialFnType}(\text{Types2Type}(\text{ParameterTypes2Types}(\text{partps}) \curvearrowright sttypes \curvearrowright rtypes), \text{BOOLEAN})$,
 .9 $\text{parms} = \text{ParameterTypes2Pats}(\text{partps}) \curvearrowright stpats \curvearrowright rpats$,
 .10 $\text{body} = \text{ONInExprSubst}(\text{subst})(\text{oppost}) \text{ in}$
 .11 $[\text{mk-ExplFnDef}(\text{idpost}, [], \text{tp}, \text{idpost}, [\text{parms}], \text{body}, \text{nil})];$

462.0 $\text{ExtractInvDefs} : (\text{TypeDef} \mid \text{StateDef})^* \xrightarrow{t} \text{ExplFnDef}^*$
 .1 $\text{ExtractInvDefs}(\text{defs}) \triangleq$
 .2 $\text{if } \text{defs} = [] \text{ then } []$
 .3 $\text{else let } \text{invfns} = \text{if Is[TypeDef, TypeDef | StateDef]}(\text{hd } \text{defs})$
 .4 $\text{then ExtractInvFromTpDef}(\text{hd } \text{defs})$
 .5 $\text{else ExtractInvFromStDef}(\text{hd } \text{defs}) \text{ in}$
 .6 $\text{invfns} \curvearrowright \text{ExtractInvDefs}(\text{tl } \text{defs});$

463.0 $\text{ExtractInvFromTpDef} : \text{TypeDef} \xrightarrow{t} \text{ExplFnDef}^*$
 .1 $\text{ExtractInvFromTpDef}(\text{td}) \triangleq$
 .2 $\text{if } \text{td.typeinv} = \text{nil} \text{ then } []$
 .3 $\text{else let } id = \text{td.id}$,
 .4 $\text{idinv} = \text{mk-InvId}(\text{GetToken}(id))$,
 .5 $\text{basetype} = \text{if } \text{is-UnTaggedTypeDef}(\text{td}) \text{ then } \text{td.shape}$
 .6 $\text{else } \text{mk-CompositeType}(\text{id}, \text{td.fields})$,
 .7 $\text{tp} = \text{mk-TotalFnType}(\text{basetype}, \text{BOOLEAN})$,
 .8 $\text{mk-InvInitFn}(p, e) = \text{td.typeinv} \text{ in}$
 .9 $[\text{mk-ExplFnDef}(\text{idinv}, [], \text{tp}, \text{idinv}, [[p]], e, \text{nil})];$

464.0 $\text{ExtractInvFromStDef} : \text{StateDef} \xrightarrow{t} \text{ExplFnDef}^*$
 .1 $\text{ExtractInvFromStDef}(\text{mk-StateDef}(id, fs, stinv, -)) \triangleq$
 .2 $\text{ExtractInvFromTpDef}(\text{mk-TaggedTypeDef}(id, fs, stinv));$

465.0 $\text{ExtractInitDefs} : \text{StateDef}^* \xrightarrow{t} \text{ExplFnDef}^*$
 .1 $\text{ExtractInitDefs}(\text{stdefs}) \triangleq$
 .2 $\text{if } \text{stdefs} = []$
 .3 $\text{then } []$
 .4 $\text{else ExtractInitDef}(\text{hd } \text{stdefs});$

466.0 $\text{ExtractInitDef} : \text{StateDef} \xrightarrow{t} \text{ExplFnDef}^*$

- .1 $\text{ExtractInitDef}(\text{mk-StateDef}(id, -, -, stinit)) \triangleq$
- .2 $\text{if } stinit = \text{nil} \text{ then } []$
- .3 $\text{else let } idinit = \text{mk-InitId}(\text{GetToken}(id)),$
- .4 $\text{basetype} = \text{mk-Name}(id),$
- .5 $\text{tp} = \text{mk-TotalFnType}(\text{basetype}, \text{BOOLEAN}),$
- .6 $\text{mk-InvInitFn}(p, e) = stinit \text{ in}$
- .7 $[\text{mk-ExplFnDef}(idinit, [], tp, idinit, [[p]], e, \text{nil})];$

467.0 $\text{UsedIds} : \text{TypeDef}^* \times \text{StateDef}^* \times \text{ValueDef}^* \times \text{FunctionDef}^* \times \text{OperationDef}^* \xrightarrow{t} \text{Id-set}$

- .1 $\text{UsedIds}(ts, ss, vs, fs, os) \triangleq$
- .2 $\text{let } tm = \text{ExtractTypeMap}(ts),$
- .3 $\text{stm} = \text{StateTypeMap}(ss) \text{ in}$
- .4 $\bigcup \{\{n.name\} \cup \text{SynComps}[Id](tm(n)) \mid n \in \text{dom } tm\} \cup$
- .5 $\bigcup \{\{n.name\} \cup \text{SynComps}[Id](stm(n)) \mid n \in \text{dom } stm\} \cup$
- .6 $\bigcup \{\text{SynComps}[Id](d) \mid d \in \text{elems } (vs \curvearrowright fs \curvearrowright os)\};$

468.0 $\text{StatePats} : \text{StateDef}^* \times \text{Id-set} \xrightarrow{t} \text{Pattern}^* \times (\text{OldName} \xrightarrow{m} \text{Name})$

- .1 $\text{StatePats}(ss, used) \triangleq$
- .2 $\text{if } ss = [] \text{ then } \text{mk-}([\text{}], \{\mapsto\})$
- .3 $\text{else let } \text{mk-StateDef}(i, fs, -, -) = \text{hd } ss,$
- .4 $\text{on} = \text{OldNewIds}(fs, used) \text{ in}$
- .5 $\text{mk-}([\text{mk-RecordPattern}(i, [\text{let } \text{mk-}(o, -) = \text{on}(i) \text{ in } \text{mk-PatternId}(o) \mid i \in \text{inds on}]),$
- .6 $\text{mk-RecordPattern}(i, [\text{let } \text{mk-}(-, n) = \text{on}(i) \text{ in } \text{mk-PatternId}(n) \mid i \in \text{inds on}]),$
- .7 $\{\text{mk-OldName}(n) \mapsto \text{mk-Name}(o) \mid \text{mk-}(o, n) \in \text{elems on}\});$

469.0 $\text{OldNewIds} : \text{Field}^* \times \text{Id-set} \xrightarrow{t} (\text{Id} \times \text{Id})^*$

- .1 $\text{OldNewIds}(fs, used) \triangleq$
- .2 $\text{if } fs = [] \text{ then } []$
- .3 $\text{else let } \text{mk-Field}(i, -) = \text{hd } fs \text{ in}$
- .4 $\text{let } \text{mk-}(o, n) : \text{Id} \times \text{Id} \text{ be st}$
- .5 $o \neq n \wedge$
- .6 $\text{if } i \neq \text{nil} \text{ then } n = i \text{ else } n \notin \text{used} \wedge$
- .7 $o \notin \text{used} \text{ in}$
- .8 $[\text{mk-}(o, n)] \curvearrowright \text{OldNewIds}(\text{tl } fs, \text{used} \cup \{o, n\});$

470.0 $\text{GetToken} : \text{Id} \xrightarrow{t} \text{token}$

- .1 $\text{GetToken}(id) \triangleq$
- .2 $\text{cases } id :$
- .3 $\text{mk-ValueId}(t), \text{PreId}(t), \text{PostId}(t), \text{InvId}(t), \text{InitId}(t), \text{MkId}(t), \text{IsId}(t) \rightarrow t$
- .4 end

11.7.5 Classification Functions

471.0 $\text{Is}[@T_1, @T_2] : @T_2 \rightarrow \mathbb{B}$

- .1 $\text{Is}(x) \triangleq$
- .2 $\exists y : @T_1 \cdot x = y;$

472.0 $\text{IsFunctionDef} : \text{ValFnOpDef} \rightarrow \mathbb{B}$

- .1 $\text{IsFunctionDef}(d) \triangleq$
- .2 $\exists fd : \text{FunctionDef} \cdot fd = d;$

473.0 $\text{IsPattern} : \text{PatternBind} \rightarrow \mathbb{B}$

- .1 $\text{IsPattern}(x) \triangleq$
- .2 $\exists p : \text{Pattern} \cdot p = x;$

474.0 $\text{Is-Expr} : \text{Expr} \mid \text{Call} \rightarrow \mathbb{B}$

- .1 $\text{Is-Expr}(x) \triangleq$
- .2 $\exists e : \text{Expr} \cdot e = x;$

475.0 $\text{IsType} : \text{Syn} \rightarrow \mathbb{B}$

- .1 $\text{IsType}(x) \triangleq$
- .2 $\exists t : \text{Type} \cdot t = x$

Bibliography

- [ABH⁺92] Derek Andrews, Hans Bruun, Bo Stig Hansen, Graeme Parkin, and Nico Plat. VDM Specification Language: Mathematical concrete syntax and corresponding outer abstract syntax. Technical report, Technical University of Denmark, November 1992.
- [AH82] Derek Andrews and Wolfgang Henhapl. Pascal. In D. Bjørner and C. B. Jones, editors, *Formal Specification & Software Development*, pages 175–251. Prentice Hall International, 1982.
- [AHW⁺96] Derek Andrews, Roger Henry, Don Ward, Kees Pronk, et al. *Information Technology — Programming Languages, Modula-2*. Number ISO/IEC 10514-1. June 1996.
- [AI91] Derek Andrews and Darell Ince. *Practical Formal Methods with VDM*. McGraw Hill, 1991.
- [Air87a] Mícheál Mac an Aircinnigh. Mathematical Structures and their Morphisms in Meta-IV. In D. Bjørner and C. B. Jones, editors, *VDM'87; VDM – A Formal Method at Work*, pages 287–320. Springer-Verlag, March 1987. LNCS 252.
- [Air87b] Mícheál Mac an Aircinnigh. Specification by Data Types. In D. Bjørner and C. B. Jones, editors, *VDM'87; VDM – A Formal Method at Work*, pages 362–388. Springer-Verlag, March 1987. LNCS 252.
- [And88] Derek Andrews. Report from the BSI Panel for the Standardization of VDM. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88; VDM – The Way Ahead*, pages 74–78. Springer-Verlag, September 1988. LNCS 328.
- [Be82] D. Bjørner and C. B. Jones (eds.). *Formal Specification and Software Development*. Prentice-Hall International, 1982.
- [Bea88] Stephen Bear. Structuring for the VDM Specification Language. In *VDM'88 VDM – The Way Ahead*, pages 2–25. VDM-Europe, Springer-Verlag, September 1988. LNCS 328.
- [BFM89] Robin Bloomfield, Peter Froome, and Brian Monahan. SpecBox: A toolkit for BSI-VDM. *SafetyNet*, (5):4–7, 1989.
- [BFPLR94] Juan Bicarregui, John Fitzgerald, Richard Moore Peter Lindsay, and Brian Ritchie. *Proof in VDM: A Practitioner's Guide*. FACIT. Springer-Verlag, 1994. ISBN 3-540-19813-X.
- [BHD91] Hans Bruun, Bo Stig Hansen, and Flemming Damm. An Approach to the Static Semantics of VDM-SL. In S. Prehn and W. J. Toetenel, editors, *VDM'91; Formal Software Development Methods*, pages 220–253. Springer-Verlag, 1991. LNCS 551.
- [BHD⁺95] Hans Bruun, Bo Stig Hansen, Flemming Damm, et al. The Static Semantics of VDM-SL. Technical Report ID/DTH 1/1, Technical University of Denmark, 1995.
- [BJ78] Dines Bjørner and Cliff B. Jones, editors. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, 1978. LNCS 61.
- [BJ82] D. Bjørner and C. B. Jones. *Formal Specification & Software Development*. Prentice Hall International, 1982.

- [BKB88] Andrzej Tarlecki Beata Konikowska and Andrzej Blikle. A three-valued logic for software specification and validation. In L. Marshall R. Bloomfield and R. Jones, editors, *VDM'88 VDM – The Way Ahead*, pages 218–242. VDM-Europe, Springer-Verlag, September 1988.
- [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.
- [BT83] Andrzej Blikle and Andrzej Tarlecki. Naive denotational semantics. *Information Processing 83, R.E.A. Mason (ed.)*, pages 345–355, 1983.
- [Che86] J. H. Cheng. *A Logic for Partial Functions*. PhD thesis, Department of Computer Science, University of Manchester, 1986. UMCS-86-7-1.
- [CJ90] J. H. Cheng and C. B. Jones. On the usability of logics which handle partial functions. In C. Morgan and J. Woodcock, editors, *Proceedings of the Third refinement Workshop*. Springer-Verlag, 1990.
- [CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.
- [Daw91] John Dawes. *The VDM-SL Reference Guide*. Pitman, 1991.
- [DBH91] Flemming Damm, Hans Bruun, and Bo Stig Hansen. On Type Checking in VDM and Related Consistency Issues. In *VDM'91; Formal Software Development Methods*, pages 45–62. Springer-Verlag, 1991. LNCS 551.
- [Din89] Dines Bjørner. Specification and Transformation, Towards a Meaning of ‘M’ in VDM. Lecture notes to IFIP TC2/WG2.2 South America Tutorial: Formal Description of Programming Concepts, April 1989, February 1989. A revised and expanded version of a TAPSOFT article.
- [ELL94] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. *ACM Sigplan Notices*, 29(9):77–80, September 1994.
- [End77] H. B. Enderton. *Elements of Set Theory*. Academic Press, London, 1977.
- [Fit94] John Fitzgerald. The VDM Forum – A new e-mail list for researchers, practitioners and teachers. *FACS Europe*, 1(2):17, Spring 1994.
- [HJ78] Wolfgang Henhapl and Cliff B. Jones. A Formal Definition of Algol 60 as described in the 1975 Modified Report. In Dines Bjørner and Cliff B. Jones, editors, *The Vienna Development Method: The Meta-Language*, pages 305–336. Springer-Verlag, 1978. LNCS 61.
- [HJ89] I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.
- [Hoa85] C. A. R. Hoare. *Communication Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1985.
- [Jon80] C. B. Jones. *Software Development — A Rigorous Approach*. Prentice Hall International, 1980.
- [Jon86] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.
- [Jon90] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, Englewood Cliffs, New Jersey, second edition, 1990.
- [JS90] C. B. Jones and R. Shaw, editors. *Case Studies in VDM*. Prentice Hall International, 1990.

- [JSS91] Niel D. Jones, Peter Sestoft, and Harald Søndergaard. Mix. a self-applicable partial evaluator for experiments in compiler generation. Rapport issn 0107-8283, Department of Computer Science, DIKU. University of Copenhagen, December 1991.
- [Kle38] S. C. Kleene. A notation for ordinal numbers. *Journal of Symbolic Logic*, 3:150–155, 1938.
- [Kle52] S. C. Kleene. *Introduction to Mathematics*. North Holland, 1952. Republished in 1957, 59, 62, 64, 71.
- [LAMB89] Peter Gorm Larsen, Michael Meincke Arentoft, Brian Monahan, and Stephen Bear. Towards a formal semantics of the BSI/VDM Specification Language. In Ritter, editor, *Information Processing 89*, pages 95–100. IFIP, North-Holland, August 1989.
- [Lar92] Peter Gorm Larsen. The Dynamic Semantics of the BSI/VDM Specification Language. Technical report, The Institute of Applied Computer Science, February 1992.
- [Lar94] Peter Gorm Larsen. The VDM Bibliography. Technical report, The Institute of Applied Computer Science, Forskerparken 10, DK-5230 Odense M, Denmark, January 1994.
- [LBC90] John T. Latham, Vicky J. Bush, and Ian D. Cottam. *The Programming Process. An Introduction Using VDM and Pascal*. Addison Wesley, 1990.
- [LHP⁺93] P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information Technology Programming Languages – VDM-SL. Technical report, First Committee Draft Standard: CD 13817-1, November 1993. ISO/IEC JTC1/SC22/WG19 N-20.
- [LHP⁺95] P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. Information Technology Programming Languages – VDM-SL. Technical report, Draft International Standard: DIS 13817-1, September 1995. ISO/IEC JTC1/SC22/WG19.
- [LHP⁺96] P. G. Larsen, B. S. Hansen, H. Brunn N. Plat, H. Toetenel, D. J. Andrews, J. Dawes, G. Parkin, et al. *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method — Specification Language — Part 1: Base language*. Number ISO/IEC 13817-1. December 1996.
- [M⁺62] J. McCarthy et al. *LISP 1.5 Programmers Manual*. MIT Press, Cambridge, Massachusetts, 1962.
- [McC61] J. McCarthy. A basis for a mathematical theory of computation. In *Western Joint Computer Conference*, 1961.
- [Mon85] Brian Q. Monahan. A Semantic Definition of the STC VDM Reference Language. BSI/IST/5/-/19 N-9, November 1985.
- [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 Bjørner, Jones, Airchinnigh, and Neuhold, editors, *VDM '87 VDM – A Formal Method at Work*, pages 210–236. VDM-Europe, Springer-Verlag LNCS 252, 1987.
- [Par79] David Park. On the semantics of fair parallelism. In Dines Bjørner, editor, *Abstract Software Specifications*, pages 504–526. Springer-Verlag LNCS86, 1979.
- [Par94] Graeme I. Parkin. Vienna development method specification language (VDM-SL). *Computer Standard & Interfaces*, 16:527–530, 1994. Special issue with the general title: The programming language standards scene, ten years on.

- [PL92] Nico Plat and Peter Gorm Larsen. An overview of the ISO/VDM-SL standard. *Sigplan Notices*, 27(8):76–82, August 1992.
- [Plot76] G. D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5:452–487, September 1976.
- [PT92] Nico Plat and Hans Toetenel. A formal transformation from the BSI/VDM-SL concrete syntax to the core abstract syntax. Technical Report 92-07, Delft University, March 1992.
- [Rey84] John C. Reynolds. Polymorphism is not set-theoretic. In D. B. MacQueen G. Kahn and G. D. Plotkin, editors, *Semantics of Data Types*, pages 145–156, 1984.
- [Sch86] David A. Schmidt. *Denotational Semantics – A Methodology for Language Development*. Allyn and Bacon, Inc. 1986.
- [Sco76] D. S. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3), September 1976.
- [Sen87] Dev Sen. Objectives of the British Standardization of a Language to Support the VDM. In D. Bjørner and C. B. Jones, editors, *VDM'87; VDM – A Formal Method at Work*, pages 321–323. Springer-Verlag, 1987. LNCS 252.
- [Spi88] Mike Spivey. *Understanding Z – A Specification Language and its formal semantics*. Cambridge University, 1988.
- [Sto77] Joseph E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.
- [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.
- [Wie89] Morten Wieth. Loose specification and its semantics. In G. X. Ritter, editor, *Information Processing 89*, pages 1115–1120. IFIP, North-Holland, August 1989.

Index A

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.

A.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 section, 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 variables¹:

Abbreviation	Explanation
<i>ev</i>	Evaluator
<i>l</i>	Loose
<i>lev</i>	Loose evaluator
<i>p</i>	Pattern
<i>pev</i>	Pattern evaluator
<i>tf</i>	Transformer
<i>ob</i>	Object
<i>den</i>	Denotation
<i>d</i>	Domain
<i>env</i>	Environment
<i>_s</i>	Set of ...
<i>_m</i>	Map of ...
<i>_l</i>	List of ...
<i>_p</i>	Pair of ...
<i>mul</i>	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 chapter 2.

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.

A.1.1 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

¹The first group contains general abbreviations of different concepts, the second group contains suffixes and the last group contains prefixes.

²This convention is also used in languages like Hope, ML and Prolog.

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
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
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: ExpandImplRecTypeDfs	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
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
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

Index B

The Concrete and Abstract Syntax

all expression, 194
AllExpr, 194
always statement, 202
AlwaysStmt, 202
and, 193
applicator, 211
Apply, 196
apply, 196, 211
arithmetic abs, 190
arithmetic divide, 192
arithmetic infix operator, 211
arithmetic integer division, 192
arithmetic minus, 192
arithmetic mod, 192
arithmetic multiplication, 192
arithmetic plus, 192
arithmetic prefix operator, 211
arithmetic rem, 192
assign statement, 200
AssignDef, 200
assignment definition, 200
AssignStmt, 200

basic type, 180
BasicType, 180
binary expression, 190
binary operator, 190
BinaryExpr, 190
BinaryOp, 192
Bind, 205
bind, 204
bind list, 205
BindList, 205
block statement, 200
BlockStmt, 200
boolean literal, 209
BoolLit, 209
bracketed expression, 187
bracketed type, 179
BracketedExpr, 187
BracketedType, 179

Call, 202
call statement, 202
CaseAltn, 188
CaseExprAlternatives, 188
cases expression, 188
cases expression alternative, 188
cases expression alternatives, 188
cases statement, 200
cases statement alternative, 201
cases statement alternatives, 201
CasesExpr, 188
CasesStmt, 200
CasesStmtAlternatives, 201
CaseStmtAltn, 201
character, 206
character literal, 210
CharLit, 210
combinator, 211
comment, 210
composite type, 180
CompositeType, 180
composition, 193, 211
compound delimiter, 208
connective, 212

dcl statement, 200
DclStmt, 200
def bind, 188
def expression, 188
def statement, 199
DefBind, 188
DefExpr, 188
definition block, 178
DefinitionBlock, 178
DefStmt, 199
delimiter, 208
delimiter character, 206
digit, 206
discretionary type, 182
DiscretionaryType, 182
distinguished letter, 206
distributed map merge, 190

distributed sequence concatenation, 190
 distributed set intersection, 190
 distributed set union, 190
Document, 178
 document, 178

 elseif expression, 188
 elseif statement, 200
ElsifExpr, 188
ElsifStmt, 200
EqDef, 199
 equal, 193
 equals definition, 199
Error, 186
 error, 186
 error list, 186
 evaluator, 211
 exceptions, 186
 exists expression, 194
 exists unique expression, 194
ExistsExpr, 194
ExistsUniqueExpr, 194
 exit statement, 203
ExitStmt, 203
ExplFnDef, 183
 explicit function definition, 183
 explicit operation definition, 185
ExplOprtDef, 185
 exponent, 209
Expr, 187
 expression, 186
 expression list, 186
ExprList, 186
 externals, 185

FctTypeInst, 196
Field, 180
 field, 180
 field list, 180
 field reference, 198
 field select, 196, 211
FieldList, 180
FieldRef, 198
FieldSelect, 196
 finite power set, 190
 floor, 190
FnType, 181
 function definition, 183
 function definitions, 183
 function type, 181
 function type instantiation, 196, 211
FunctionDef, 183
FunctionDefinitions, 183

 general map type, 181

GeneralMapType, 181
 greater than, 192
 greater than or equal, 192
 Greek letter, 206

Id, 208
 identifier, 208
 identifier type pair, 184
 identity statement, 203
IdentStmt, 203
IdType, 184
 if expression, 188
 if statement, 200
IfExpr, 188
IfStmt, 200
ImplFnDef, 183
 implicit function definition, 183
 implicit operation definition, 185
ImplOprtDef, 185
 imply, 193
 in set, 193
 index for loop, 201
IndexForLoop, 201
Initialization, 182
 initialization, 182
InitId, 209
 injective map type, 181
InjectiveMapType, 181
Invariant, 182
 invariant, 182
 invariant initial function, 182
InvId, 209
InvInitFn, 182
 iota expression, 194
IotaExpr, 194
 is basic type, 197
 is basic type expression, 197
 is defined type expression, 197
 is expression, 197
IsBasicType, 197
IsBasicTypeExpr, 197
IsDefTypeExpr, 197
IsExpr, 197
IsId, 209
 iterate, 193, 211

 keyword, 208
 keyword letter, 206

Lambda, 196
 lambda expression, 196
 less than, 192
 less than or equal, 192
 let be expression, 187
 let be statement, 199

let expression, 187
 let statement, 199
LetBeSTEexpr, 187
LetBeSTStmt, 199
LetExpr, 187
LetStmt, 199
Literal, 209
 local definition, 199
LocalDef, 199
 logical equivalence, 193
 logical infix operator, 212
 logical prefix operator, 212

 map comprehension, 195
 map domain, 190
 map domain restrict by, 193
 map domain restrict to, 193
 map enumeration, 195
 map infix operator, 212
 map inverse, 211
 map inverse expression, 190
 map merge, 193
 map or sequence modify, 193
 map or sequence reference, 198
 map prefix operator, 211
 map range, 190
 map range restrict by, 193
 map range restrict to, 193
 map type, 181
MapComprehension, 195
MapEnumeration, 195
MapInverseExpr, 190
Maplet, 195
 maplet, 195
MapOrSeqRef, 198
MapType, 181
 match value, 203
MatchVal, 203
MkId, 209
Mode, 185
 mode, 185
MultBind, 205
 multiple bind, 205
 multiple set bind, 205
 multiple type bind, 205
MultSetBind, 205
MultTypeBind, 205

Name, 197
 name, 197
 name list, 197
NameList, 197
 nil literal, 209
NilLit, 210
 nondeterministic statement, 202

NonDetStmt, 202
 not, 190
 not equal, 193
 not in set, 193
 numeral, 209
 numeric literal, 209
NumLit, 209

OExceptions, 186
OExprList, 186
OExternals, 185
 old name, 197
OldName, 197
OPatList, 204
OPatTypePairList, 184
 operation definition, 184
 operation definitions, 184
 operation type, 185
OperationDef, 184
OperationDefinitions, 184
 optional type, 180
OptionalType, 180
OpType, 185
 or, 193
 other character, 206
 others expression, 188
 others statement, 201
OthersExpr, 188
OthersStmt, 201
OTypeVarList, 184

 parameter types, 184
Parameters, 184
 parameters, 184
 parameters list, 184
ParametersList, 184
ParameterTypes, 184
 partial function type, 181
PartialFnType, 181
PatList, 204
Pattern, 203
 pattern, 203
 pattern bind, 204
 pattern identifier, 203
 pattern list, 204
 pattern type pair, 184
 pattern type pair list, 184
PatternBind, 204
PatternId, 203
PatTypePair, 184
 plain letter, 206
PostId, 209
 prefix expression, 189
PrefixExpr, 189
PreId, 209

product type, 180
ProductType, 180
 proper subset, 193

QuantExpr, 193
 quantified expression, 193
 quote literal, 210
 quote type, 180
QuoteLit, 210
QuoteType, 180

 record constructor, 196
 record modification, 196
 record modifier, 196
 record pattern, 204
RecordConstructor, 196
RecordModification, 196
RecordModifier, 196
RecordPattern, 204
RecTrapStmt, 202
 recursive trap statement, 202
 relation, 212
 relational infix operator, 212
 return statement, 202
ReturnStmt, 202

 separator, 208
 seq conc pattern, 204
 seq enum pattern, 204
 seq type, 181
 seq0 type, 181
Seq0Type, 181
 seq1 type, 181
Seq1Type, 181
SeqComprehension, 195
SeqConcPattern, 204
SeqEnumeration, 195
SeqEnumPattern, 204
SeqForLoop, 201
SeqType, 181
 sequence comprehension, 195
 sequence concatenate, 193
 sequence elements, 190
 sequence enumeration, 195
 sequence for loop, 201
 sequence head, 190
 sequence indices, 190
 sequence infix operator, 212
 sequence length, 190
 sequence prefix operator, 211
 sequence tail, 190
 set bind, 205
 set cardinality, 190
 set comprehension, 194
 set difference, 193

 set enum pattern, 203
 set enumeration, 194
 set for loop, 201
 set infix operator, 212
 set intersection, 193
 set prefix operator, 211
 set range expression, 194
 set relational operator, 212
 set type, 180
 set union, 193
 set union pattern, 204
SetBind, 205
SetComprehension, 194
SetEnumeration, 194
SetEnumPattern, 204
SetForLoop, 201
SetRange, 194
SetType, 181
SetUnionPattern, 204
 state definition, 182
 state designator, 198
StateDef, 182
StateDesignator, 198
 statement, 198
Stmt, 199
SubSequence, 195
 subsequence, 195, 211
 subset, 193
 symbolic literal, 209

 tagged type definition, 179
TaggedTypeDef, 179
 text literal, 210
TextLit, 210
 total function type, 181
TotalFnType, 182
Trap, 203
 trap, 202
 trap statement, 202
Traps, 202
 traps, 202
TrapStmt, 202
 tuple constructor, 195
 tuple pattern, 204
TupleConstructor, 195
TuplePattern, 204
Type, 179
 type, 179
 type bind, 205
 type bind list, 205
 type definition, 178
 type definitions, 178
 type name, 182
 type variable, 182

type variable identifier, 182
type variable list, 184
TypeBind, 205
TypeBindList, 205
TypeDef, 179
TypeDefinitions, 178
TypeName, 182
TypeVar, 182
TypeVarId, 182

unary expression, 189
unary minus, 190
unary operator, 189
unary plus, 189
UnaryExpr, 189
UnaryOp, 189
union type, 180
UnionType, 180
untagged type definition, 179
UnTaggedTypeDef, 179

value definition, 183
value definitions, 183
ValueDef, 183
ValueDefinitions, 183
ValueId, 209
var information, 185
VarInf, 185

while loop, 201
WhileLoop, 201

Index C

The Syntax Mapping

- AllExpr2AllOrExistsExpr, 247, 250
AlwaysStmt2Always, 256, 260
Apply2Apply, 247, 253
AssignStmt2Assign, 256, 258

BinaryExpr2BinaryExpr, 247, 250
Bind2Bind, 249–251, 257, 262, 263
BindList2BindSet, 250–252, 262
BlockStmt2BlockOrSequence, 256, 257, 257, 260

Call2Call, 256–258, 260
CaseAltn2CaseAltn, 249, 249
CasesExpr2CasesExpr, 247, 249
CasesStmt2CasesStmt, 256, 258
CaseStmtAltn2CaseStmtAltn, 258, 258
CompositeType2CompositeType, 230, 230

DefExpr2DefExpr, 247, 249, 249
DefinitionBlocks2ExplFnDefMap, 225, 226
DefinitionBlocks2ExplOpDefMap, 225, 227
DefinitionBlocks2ExplPolyFnDefMap, 225, 226
DefinitionBlocks2ImplFnDefMap, 225, 227
DefinitionBlocks2ImplOpDefMap, 225, 228
DefinitionBlocks2ImplPolyFnDefMap, 225, 226
DefinitionBlocks2StateDef, 225, 228
DefinitionBlocks2TypeDefMap, 225, 225
DefinitionBlocks2ValDefMap, 225, 227
DefStmt2DefStmt, 256, 257, 257
Document2Definitions, 225

ExistsExpr2AllOrExistsExpr, 247, 250
ExistsUniqueExpr2ExistsUniqueExpr, 247, 250
ExitStmt2ExitStmt, 256, 261
ExplFnDef2ExplFnDef, 234, 236, 237, 238, 240, 242, 244
ExplOprtDef2ExplOpDef, 242, 242
Expr2Expr, 233, 235, 245, 246, 246, 247–255, 257–259, 261, 262

FctTypeInst2FctTypeInst, 247, 253
Field2Field, 230, 230
FieldSelect2FieldSelect, 247, 253
FnType2FnType, 230, 231, 237

FunctionDef2ExplFnDefMap, 236, 236, 248, 256
FunctionDef2ExplPolyFnDefMap, 233, 234
FunctionDef2ImplFnDefMap, 240, 240, 248, 256
FunctionDef2ImplPolyFnDefMap, 237, 237
FunctionDefinitions2ExplFnDefMap, 226, 236
FunctionDefinitions2ExplPolyFnDefMap, 226, 233
FunctionDefinitions2ImplFnDefMap, 227, 240
FunctionDefinitions2ImplPolyFnDefMap, 227, 237

GeneralMapType2GeneralMapType, 230, 231
GetUnusedId, 224, 229, 235, 239, 243, 244, 254

Id2Id, 229, 230, 232, 234, 236, 238–240, 242–245, 247, 252–255, 257, 259–262, 263
IfExpr2IfExpr, 247, 249, 249
IfStmt2IfStmt, 256, 258, 258
ImplFnDef2ImplFnDef, 238, 238, 240
ImplOprtDef2ImplOpDef, 244, 245
IndexForLoop2IndexForLoop, 256, 259
InjectiveMapType2InjectiveMapType, 230, 231
InvInitFn2Lambda, 229, 229, 232
IotaExpr2IotaExpr, 247, 251
IsBasicTypeExpr2IsExpr, 247, 254
IsDefTypeExpr2IsExpr, 247, 254
IsExpr, 247
IsPolymorphic, 233, 234, 236, 237, 240, 248, 256

Lambda2Lambda, 247, 254
LetBeSTE Expr2LetBeSTE Expr, 247, 248
LetBeST Stmt2LetBeST Stmt, 256, 257
LetExpr2LetExpr, 230, 247, 248
LetStmt2LetStmt, 256, 256

MakeDisjExcps, 245, 246, 246
MakeLambda, 234, 235, 237
MakeLetExpr, 229, 229, 239, 254
MakeOpHeading, 243, 243, 245
MakePostDisjExcps, 245, 245
MapComprehension2MapComprehension, 247, 252
MapEnumeration2MapEnumeration, 247, 252
MapInverseExpr2UnaryExpr, 247, 250
Maplet2Maplet, 252, 252, 252

Name2Id, 253, 255
NonDetStmt2NonDetStmt, 256, 259
NumberOfParsMatchesFnType, 235, 235
NumberOfParsMatchesOpType, 242, 242

OExpr2TrueOrExpr, 237, 243, 246, 249, 251, 252, 257
OldName2OldId, 247, 255
OperationDef2ExplOpDefMap, 241, 241
OperationDef2ImplOpDefMap, 243, 244, 244
OperationDefinitions2ExplOpDefMap, 227, 228, 241
OperationDefinitions2ImplOpDefMap, 228, 243
OptionalType2OptionalType, 230, 231
OTypeVarList2TypeVarList, 236, 237, 239

Pattern2Pattern, 233, 235, 245, 250, 254, 259, 261, 261, 262
PatternBind2Pattern, 249, 257, 259, 260, 262
PatternId2PatternId, 261, 261
PrefixExpr2UnaryExpr, 247, 250
ProductType2ProductType, 230, 231, 239, 254

QuoteType2QuoteType, 230, 230

RecordConstructor2RecordConstructor, 247, 252
RecordModifier2RecordModifier, 247, 253
RecordPattern2RecordPattern, 261, 262
RecTrapStmt2RecTrapStmt, 256, 260
ReturnStmt2ReturnStmt, 256, 260

Seq0Type2Seq0Type, 230, 231
Seq1Type2Seq1Type, 230, 231
SeqComprehension2SeqComprehension, 247, 251
SeqConcPattern2SeqConcPattern, 261, 262
SeqEnumeration2SeqEnumeration, 247, 251
SeqEnumPattern2SeqEnumPattern, 261, 261
SeqForLoop2SeqForLoop, 256, 259
SetComprehension2SetComprehension, 247, 251
SetEnumeration2SetEnumeration, 247, 251
SetEnumPattern2SetEnumPattern, 261, 261
SetForLoop2SetForLoop, 256, 259
SetRange2SetRange, 247, 251
SetType2SetType, 230, 231
SetUnionPattern2SetUnionPattern, 261, 261
SplitOPatTypePairList, 238, 239, 239, 240, 244, 245
StateDef2StateDef, 228, 232
StateDesignator2StateDesignator, 255, 255, 255, 258, 260
Stmt2Stmt, 243, 256, 256, 257–260
SubSequence2SubSequence, 247, 251

TrapStmt2TrapStmt, 256, 260
TupleConstructor2TupleConstructor, 247, 252
TuplePattern2TuplePattern, 245, 254, 261, 262

Index D

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**
 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**
 OTypr2Types, **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**
 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**
 TypeRTTypeRel, 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**
 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-SqComprehension, 182, **262**
wf-SEQCONC, 214, **240**
wf-SqConcPattern, 325, **335**
wf-SEQDISTRCONC, 193, **208**
wf-SEQELEMS, 193, **206**
wf-SqEnumeration, 182, **261**
wf-SqEnumPattern, 325, **334**
wf-SqForLoop, 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**

