

INTERNATIONAL STANDARD

ISO/IEC 13817-1

First edition
1996-12-15

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

Part 1: Base language

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

Partie 1: Langage de base



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

AVA Documents
Document Sharing Hub

Contents

	Page
Foreword	xi
Introduction	xii
1 Scope	1
2 Normative References	2
3 Definitions	3
3.1 Structure of Formal Definition	4
3.2 Conventions	4
3.2.1 Informative text	4
4 Conformity	5
4.1 Specifications	5
5 Basic Mathematical Notation	6
5.1 Logic Notation	6
5.2 Basic Set Theory	6
5.3 Cartesian Products	7
5.4 Binary Relations and Functions	8
5.5 Finite Sequences	9
5.6 Finite Mappings	10
5.7 Ordinal Numbers	11
5.8 Definition by Transfinite Induction	12
5.9 Cardinality and Cardinal Numbers	12
5.10 Structured Expressions	13
5.11 Semantic Function and Predicate Definitions	15
5.12 Use of Recursion	16
6 Core Abstract Syntax	17
6.1 Document	17
6.2 Definitions	17
6.2.1 Type Definitions	17
6.2.2 State Definition	19
6.2.3 Value Definitions	19
6.2.4 Function Definitions	19
6.2.5 Operation Definitions	20

6.3	Expressions	21
6.3.1	Local Binding Expressions	22
6.3.2	Conditional Expressions	22
6.3.3	Unary Expressions	22
6.3.4	Binary Expressions	23
6.3.5	Quantified Expressions	23
6.3.6	Iota Expression	24
6.3.7	Set Expressions	24
6.3.8	Sequence Expressions	24
6.3.9	Map Expressions	25
6.3.10	Tuple Constructor	25
6.3.11	Record Expressions	25
6.3.12	Apply Expressions	25
6.3.13	Lambda Expression	26
6.3.14	Is Expression	26
6.3.15	Literals	26
6.3.16	Identifiers	27
6.4	State Designators	28
6.5	Statements	28
6.5.1	Local Binding Statements	28
6.5.2	Block and Assignment Statements	29
6.5.3	Conditional Statements	29
6.5.4	Loop Statements	29
6.5.5	Non-Deterministic Sequences	30
6.5.6	Call and Return Statements	30
6.5.7	Exception Handling Statements	31
6.6	Patterns and Bindings	31
7	Dynamic Semantic Domains	34
7.1	The Domain Universe	34
7.1.1	Basic Definitions	34
7.1.1.1	Complete Partial Orders and Fixed Point Definitions	34
7.1.1.2	Operators on Complete Partial Orders	35
7.1.2	A Universe of Complete Partial Orders	37
7.1.3	A Universe of Domains	38
7.2	The Semantic Domains	40
7.2.1	Basic Semantic Domains	40
7.2.2	Extended Semantic Domains	42
7.2.3	Semantic Domains for Evaluation Functions	43
7.2.4	Auxiliary Semantic Domains	45
8	The Dynamic Semantics	46
8.1	Document	46
8.2	Definitions	48
8.2.1	Type Definitions	48
8.2.1.1	The Verification Predicate for Types	49
8.2.1.2	Evaluation Functions for Types	50
8.2.2	State Definition	57
8.2.3	Value Definitions	57
8.2.3.1	The Verification Predicate for Value Definitions	58
8.2.3.2	The Evaluation Function for Value Definitions	61
8.2.4	Function Definitions	63
8.2.4.1	Verification Predicates for Function Definitions	63
8.2.4.2	Evaluation Functions for Polymorphic Functions	67
8.2.5	Operation Definitions	69
8.2.5.1	Verification Predicates for Operation Definitions	69
8.2.5.2	Evaluation Functions for Operation Definitions	70

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

8.7.12	General Functions and Predicates	171
9	The Mathematical Concrete Syntax	178
9.1	Document	178
9.2	Definitions	178
9.2.1	Type Definitions	178
9.2.2	State Definition	179
9.2.3	Value Definitions	179
9.2.4	Function Definitions	179
9.2.5	Operation Definitions	179
9.3	Expressions	180
9.3.1	Bracketed Expressions	180
9.3.2	Local Binding Expressions	180
9.3.3	Conditional Expressions	180
9.3.4	Unary Expressions	181
9.3.5	Binary Expressions	181
9.3.6	Quantified Expressions	182
9.3.7	Iota Expression	183
9.3.8	Set Expressions	183
9.3.9	Sequence Expressions	183
9.3.10	Map Expressions	183
9.3.11	Tuple Constructor Expression	183
9.3.12	Record Expressions	183
9.3.13	Apply Expressions	183
9.3.14	Lambda Expression	183
9.3.15	Is Expressions	183
9.3.16	Names	183
9.4	State Designators	183
9.5	Statements	184
9.5.1	Local Binding Statements	184
9.5.2	Block and Assignment Statements	184
9.5.3	Conditional Statements	184
9.5.4	Loop Statements	184
9.5.5	Nondeterministic Statement	184
9.5.6	Call and Return Statements	185
9.5.7	Exception Handling Statements	185
9.5.8	Identity Statement	185
9.6	Patterns and Bindings	185
9.6.1	Patterns	185
9.6.2	Bindings	185
9.7	Lexical Specification	185
9.7.1	General	185
9.7.2	Characters	186
9.7.3	Symbols	188
9.8	Operator Precedence	188
9.8.1	The Family of Combinators	189
9.8.2	The Family of Applicators	189
9.8.3	The Family of Evaluators	189
9.8.4	The Family of Relations	190
9.8.5	The Family of Connectives	190
9.8.6	The Family of Constructors	190
9.8.7	Grouping	190
9.8.8	The Type Operators	191
10	The Interchange Concrete Syntax	192
10.1	Introduction	192
10.2	Lexis	192

10.3 Symbols 192

11 The Outer Abstract Syntax 196

11.1 Document 196

11.2 Definitions 196

11.2.1 Type Definitions 196

11.2.2 State Definition 197

11.2.3 Value Definitions 198

11.2.4 Function Definitions 198

11.2.5 Operation Definitions 198

11.3 Expressions 199

11.3.1 Bracketed Expressions 200

11.3.2 Local Binding Expressions 200

11.3.3 Conditional Expressions 200

11.3.4 Unary Expressions 201

11.3.5 Binary Expressions 201

11.3.6 Quantified Expressions 202

11.3.7 Iota Expression 202

11.3.8 Set Expressions 202

11.3.9 Sequence Expressions 203

11.3.10 Map Expressions 203

11.3.11 Tuple Constructor Expression 203

11.3.12 Record Expressions 203

11.3.13 Apply Expressions 203

11.3.14 Lambda Expression 204

11.3.15 Is Expressions 204

11.3.16 Names 204

11.4 State Designators 204

11.5 Statements 204

11.5.1 Local Binding Statements 205

11.5.2 Block and Assignment Statements 205

11.5.3 Conditional Statements 205

11.5.4 Loop Statements 206

11.5.5 Nondeterministic Statement 206

11.5.6 Call and Return Statements 206

11.5.7 Exception Handling Statements 206

11.5.8 Identity Statement 206

11.6 Patterns and Bindings 207

11.6.1 Patterns 207

11.6.2 Bindings 207

11.7 Lexical Specification 208

12 The Syntax Mapping 210

12.1 Structure and Style of the Definition 210

12.1.1 Division into Modules 210

12.1.1.1 Module "OAS2CAS" 210

12.1.1.2 Module "OAS" 210

12.1.1.3 Module "CAS" 210

12.1.1.4 Module "GetUnusedId" 210

12.1.2 Pre-conditions in the VDM-SL Definition of the Syntax Mapping 211

12.1.3 Transformation of a Document to CAS.Definitions 211

12.1.3.1 The Introduction of Additional Identifiers 211

12.1.3.2 The Generation of Quoted Pre- and Post-conditions for Functions 212

12.1.3.3 The Generation of Quoted Pre- and Post-conditions for Operations 212

12.1.3.4 The Transformation of Expressions 213

12.1.3.5 Pre-conditions of Operations 213

12.1.3.6 Guards of Error Handlers 214

12.1.3.7	Quoted Post-conditions of Implicit Operations	214
12.1.3.8	The Transformation of Type Definitions	214
12.1.3.9	The Transformation of Value Definitions	214
12.1.3.10	The Transformation of the State	214
12.1.4	Notational Conventions	215
12.2	Syntaxes and Auxiliary Functions	215
12.2.1	Module "OAS"	215
12.2.2	Module "CAS"	215
12.2.3	Module "GetUnusedId"	216
12.3	The Syntax Mapping Functions	216
12.3.1	Document	217
12.3.2	Definitions	220
12.3.2.1	Type Definitions	220
12.3.2.2	State Definition	224
12.3.2.3	Value Definitions	225
12.3.2.4	Function Definitions	225
12.3.2.5	Operation Definitions	232
12.3.3	Expressions	237
12.3.3.1	Bracketed Expressions	239
12.3.3.2	Local Binding Expressions	239
12.3.3.3	Conditional Expressions	240
12.3.3.4	Unary Expressions	241
12.3.3.5	Binary Expressions	241
12.3.3.6	Quantified Expressions	241
12.3.3.7	Iota Expression	241
12.3.3.8	Set Expressions	242
12.3.3.9	Sequence Expressions	242
12.3.3.10	Map Expressions	242
12.3.3.11	Tuple Constructor Expression	243
12.3.3.12	Record Expressions	243
12.3.3.13	Apply Expressions	244
12.3.3.14	Lambda Expression	244
12.3.3.15	Is Expressions	245
12.3.3.16	Names	245
12.3.4	State Designators	246
12.3.5	Statements	246
12.3.5.1	Local Binding Statements	247
12.3.5.2	Block and Assignment Statements	248
12.3.5.3	Conditional Statements	248
12.3.5.4	Loop Statements	249
12.3.5.5	NonDeterministic Statement	250
12.3.5.6	Call and Return Statements	250
12.3.5.7	Exception Handling Statements	250
12.3.5.8	Identity Statement	251
12.3.6	Patterns and Bindings	251
12.3.6.1	Patterns	251
12.3.6.2	Bindings	252
12.3.7	Lexical Specification	253
12.3.7.1	General	253
12.3.7.2	Characters	253
12.3.7.3	Symbols	253
13	The Static Semantic Domains	255
13.1	Type Representations	255
13.1.1	Special Subclasses of Type Representations	256
13.2	Environments	257
13.2.1	Accessing Environments	257

- 13.2.2 Updating Environments 258
 - 13.3 Well-formedness Classifications 258
 - 13.4 Type Relations 259
 - 13.4.1 Subtypes 259
 - 13.4.2 Overlapping Subtypes, Disjoint Types and Overlapping Types 264
 - 13.4.3 Auxiliary Type Relations and Functions 267
 - 13.5 Extended Abstract Syntax 273
- 14 The Static Semantics 273
 - 14.1 Documents 274
 - 14.1.1 Auxiliary Well-formedness Requirements 275
 - 14.2 Definitions 276
 - 14.2.1 Type and State Definitions 276
 - 14.2.1.1 Types and Type maps 277
 - 14.2.1.2 Extraction of Type Representations 279
 - 14.2.2 Value, Function, and Operation Definitions 280
 - 14.2.2.1 Simultaneous Definitions 280
 - 14.2.2.2 Ordered Definition Groups and Definition Sequences 281
 - 14.2.2.3 Definitions 282
 - 14.2.2.4 Definition Groups and their Ordering 285
 - 14.2.2.5 Ordering of Definitions 286
 - 14.2.2.6 Asserted Visible Environments 287
 - 14.2.2.7 Pre-Conditions 289
 - 14.3 Expressions 289
 - 14.3.1 Expression Characteristic Predicates 290
 - 14.3.2 Relaxations and Restriction of Predicates 291
 - 14.3.2.1 The Subsumption Rule 291
 - 14.3.2.2 The InType Rule 291
 - 14.3.2.3 Union Close 292
 - 14.3.2.4 Sub-Environment Based Checks 292
 - 14.3.3 Well-Formedness of Expressions 294
 - 14.3.4 Bracketed Expression 295
 - 14.3.5 Local Binding Expressions 295
 - 14.3.5.1 Let Expression 295
 - 14.3.5.2 Let Be ST Expression 295
 - 14.3.5.3 Def Expression 296
 - 14.3.6 Conditional Expressions 296
 - 14.3.6.1 If Expression 296
 - 14.3.6.2 Cases Expression 297
 - 14.3.7 Unary Expressions 298
 - 14.3.7.1 Numeric Operations 299
 - 14.3.7.2 Logical Operation 300
 - 14.3.7.3 Set Operations 301
 - 14.3.7.4 Sequence Operations 302
 - 14.3.7.5 Map Operations 303
 - 14.3.7.6 Map Inverse Expression 304
 - 14.3.8 Binary Expressions 304
 - 14.3.8.1 Numeric Operations 305
 - 14.3.8.2 Numeric Comparison Operators 307
 - 14.3.8.3 Equality operations 308
 - 14.3.8.4 Logical Operations 308
 - 14.3.8.5 Logical Set Operations 310
 - 14.3.8.6 Set Operations 311
 - 14.3.8.7 Sequence Operation 311
 - 14.3.8.8 Sequence and Map Modification Operation 312
 - 14.3.8.9 Map Operations 313
 - 14.3.8.10 Compose and Iterate 315

14.3.9	Quantified Expressions	316
14.3.10	Iota Expression	317
14.3.11	Set Expressions	318
14.3.12	Sequence Expressions	319
14.3.13	MapExpressions	320
14.3.14	Tuple Constructor	321
14.3.15	Record Expressions	322
14.3.16	Apply Expressions	323
14.3.17	Lambda Expressions	324
14.3.18	Is Expressions	326
14.3.19	Names	326
14.3.20	Literals	326
14.3.21	Auxiliary Well-formedness Predicates	327
14.4	State Designators	327
14.5	Statements	328
14.5.1	Local Binding Statements	329
14.5.2	Block and Assignment Statements	330
14.5.3	Conditional Statements	332
14.5.4	Loop Statements	333
14.5.5	Non-Deterministic Statement	334
14.5.6	Call and Return Statements	335
14.5.7	Exception Handling Statements	335
14.5.8	Identity Statements	337
14.6	Patterns and Bindings	337
14.6.1	Patterns	338
14.6.1.1	Pattern Characteristic Predicates	338
14.6.1.2	Relaxation of Pattern Characteristic Predicates	340
14.6.1.3	Well-formedness of Patterns	340
14.6.2	Bindings	343
14.6.3	Value Environment	346
14.7	Auxiliary Functions	347
14.7.1	Dependency Relations	347
14.7.1.1	Free and Defined Names	348
14.7.1.2	Syntactical Sub-Components	356
14.7.1.3	The Definitional Basis of Types	361
14.7.2	Syntax Transformations	361
14.7.3	Substitutions	363
14.7.4	Indirectly Defined Functions	368
14.7.5	Classification Functions	371
A	Extensions	372
B	Tool Conformity	373
B.1	Semantic conformity of tools	373
C	Modules	375
C.1	Overview	375
C.2	Requirements of Modularization	375
C.2.1	Language Facilities	375
C.2.1.0.1	Syntactic Separation	375
C.2.1.0.2	Explicit Import	375
C.2.1.0.3	Explicit Export	375
C.2.1.0.4	Parameterization	376
C.2.2	Semantics	376
C.3	Extant Approaches to Structuring Specifications	376
C.3.1	The Syntactic Approach	376
C.3.2	The Z Approach	376

- C.3.3 The Manchester Approach (Fitzgerald & Jones) 376
 - C.3.4 The VVSL Approach (Middelburg) 377
 - C.3.5 The RAISE Approach 377
- D Cross-references 379
 - D.1 Cross References for the Dynamic Semantics 379
 - D.1.1 Naming and Typesetting Conventions Used 379
 - D.1.2 Listing of Functions/Predicates: Alphabetic (uses) 380
 - D.2 Cross References for the Concrete Syntax 385
 - D.3 Cross References for the Abstract Syntax 388
 - D.4 Cross References for the Syntax Mapping 390
 - D.5 Cross References for the Static Semantics 393
- E Bibliography 397

Figures

- 1 Structure of the syntax mapping 210

Tables

- 1 Character set 187
- 2 Interchange syntax: representation of symbols 192

Foreword

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

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

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

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

— *Part 1: Base language*

Additional parts will specify modules and the development method.

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

Introduction

Historical background of the Vienna Development Method (VDM)

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

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

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

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

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

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

During the period from 1971 to 1973, the Vienna group was diverted into other activities not really related to formal description. Cliff Jones at this time went back to the Hursley Laboratory and worked on a *functional* language description (ACJ72) and other aspects of what has become known as VDM. In particular he published a development of Earley's recogniser (Jon72) which is one of the first reports to use data reification. In late 1972 and throughout '73 and '74 the Vienna group (Cliff Jones returned and Dines Bjørner was recruited) had the opportunity to work on a PL/I compiler for what was then a very novel machine architecture. They of course decided to base their development for the compiler on a formal description of the programming language. PL/I was then undergoing ECMA/ANSI standardization (ANS76). The Vienna group chose to write a denotational semantics for PL/I (BBH⁺74); this is the origin of the VDM¹⁾ work on language description techniques.

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

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

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

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

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

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

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

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

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

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

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

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

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

Part 1: Base language

1 Scope

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

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

It does not specify:

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

2 Normative References

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

ISO 8859-1:1987¹⁾, *Information processing — 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*.

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

1) Currently under revision.

3 Definitions

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

- 3.1 conforming specification:** A specification which is written in the language defined by this part of ISO/IEC 13817 and which obeys the conformity clauses for specifications in this part of ISO/IEC 13817.
- 3.2 conforming tool:** A tool which processes conforming specifications and which obeys the conformity clauses for tools in this part of ISO/IEC 13817.
- 3.3 consistent specification:** A specification that satisfies either the mathematical or interchange concrete syntax and can be translated by the syntax mapping into a representation that uses the core abstract syntax and for which the dynamic semantics defines at least one model.
- 3.4 core abstract syntax:** The core abstract syntax gives a second abstract version of the underlying structure of the language that uses a reduced number of constructs. Translation between the outer abstract syntax and the core abstract syntax is, (in general) explicitly defined by a mapping function (see clause 12). In a few cases, where the core abstract syntax is not clearly related to the outer abstract syntax, translation or technical notes (annotations) are used to explain the translation.
- 3.5 dynamic semantics:** Dynamic semantics are the rules for the derivation of the models of a specification that satisfies either the mathematical or interchange concrete syntax and can be translated by the mapping function into a representation that uses the core abstract syntax. The dynamic semantics are expressed using a mathematical notation that is defined in clause 5. The meaning of a consistent specification is given by a set of models that the specification defines.
- 3.6 error:** A fault in a specification which means the specification could be a meaningless specification or the specification is inconsistent.
- 3.7 extension:** An addition to the requirements of this part of ISO/IEC 13817 that does not invalidate any specification conforming with this standard except by prohibiting the use of one or more particular spellings of identifiers. This implies that any new semantic definitions that are introduced must be consistent with those in this part of ISO/IEC 13817.
- 3.8 inconsistent specification:** A specification that is not consistent — the set of models defined for the specification as defined by the dynamic semantics is empty.
- 3.9 interchange concrete syntax:** The interchange concrete syntax is an alternative set of lexical rules from the Lexis (see clause 9.7) the representation of which uses standard symbols.
- 3.10 mathematical concrete syntax:** The mathematical concrete syntax is a set of grammatical rules defining the syntactic structure of the language in terms of other concrete syntax rules, or lexical rules from the Lexis (see clause 9.7).
- 3.11 meaningless specification:** A specification which cannot be translated by the syntax mapping to the core abstract syntax and thus for which no semantics can be given.
- 3.12 model:** A mathematical construction designated as the meaning of a specification (or part of such specification) as defined by the dynamic semantics.
- 3.13 outer abstract syntax:** The outer abstract syntax gives an abstract version of the underlying structure of the language. Translation between the concrete and abstract syntax is implicit. In a few cases, where the abstract syntax is not clearly related to the concrete syntax, translation or technical notes (annotations) are added to explain any apparent divergence.
- 3.14 proof obligations:** The set of proofs that are necessary to show that a specification is semantically correct.
- 3.15 reification rules:** A set of rules by which it is possible to express that one specification satisfy another more abstract specification. The first specification can then be considered an implementation of the latter.

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

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

3.18 system: Software for controlling the execution of tools.

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

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

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

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

3.1 Structure of Formal Definition

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

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

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

3.2 Conventions

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

3.2.1 Informative text

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

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

4 Conformity

4.1 Specifications

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

clause 9 or

clause 10

and is syntax-conformant or consistent with respect to

clause 6,

clause 7,

clause 8,

clause 11 and

clause 12.

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

5 Basic Mathematical Notation

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

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

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

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

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

5.1 Logic Notation

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

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

5.2 Basic Set Theory

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

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

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

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

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

For any set A , $\mathcal{P}(A)$ denotes the powerset of A , i.e. the set of all subsets of A . Similarly, $\mathcal{IF}(A)$ 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, $\text{card}(A)$, is used to get the cardinality of a set A . Normally this will be used for finite sets, however in a few places it is also used for possibly infinite sets. This will be further explained in a clause about cardinal numbers (see 5.9).

Finally, some basic domains (such as Booleans (B), the natural numbers (N), integers (Z), the rational numbers (Q), reals (R)) 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).

5.3 Cartesian Products

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

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

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

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

The (generalized) Cartesian product of n sets $A_1, \dots, A_n$ 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:

$$\begin{aligned} &\forall a_1, a_1' \in A_1, \dots, \forall a_n, a_n' \in A_n. \\ &(a_1, \dots, a_n) = (a_1', \dots, a_n') \Leftrightarrow (a_1 = a_1') \wedge \dots \wedge (a_n = a_n') \end{aligned}$$

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.

5.4 Binary Relations and Functions

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

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

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

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

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

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

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

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

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

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

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

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

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

5.5 Finite Sequences

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

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

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

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

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

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 $\text{len}(s)$ such that $\{1, \dots, \text{len}(s)\} = \delta_0(s)$. This gives a function

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

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 $\{1, \dots, \text{len}(s)\}$. 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 $1 \leq n \leq \text{len}(s)$.

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

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

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

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

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

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

$$\text{hd} : \mathbb{L}_1(A) \rightarrow A$$

$$\text{hd}(s) = s(1)$$

$$\text{tl} : \mathbb{L}_1(A) \rightarrow \mathbb{L}(A)$$

$$\text{tl}(s) = [s(i) \mid i \in \{2, \dots, \text{len}(s)\}]$$

$$\text{inds} : \mathbb{L}(A) \rightarrow \mathbb{IF}(\omega)$$

$$\text{inds}(s) = \{1, \dots, \text{len}(s)\}$$

$$\text{elems} : \mathbb{L}(A) \rightarrow \mathbb{IF}(A)$$

$$\text{elems}(s) = \{s(i) \mid i \in \text{inds}(s)\}$$

$$\text{join} : \mathbb{L}(A) \times \mathbb{L}(A) \rightarrow \mathbb{L}(A)$$

$$\text{join}([a_1, \dots, a_n], [b_1, \dots, b_m]) = [a_1, \dots, a_n, b_1, \dots, b_m]$$

$$\text{Join} : \mathbb{L}(\mathbb{L}(A)) \rightarrow \mathbb{L}(A)$$

$$\text{Join}(l_1, \dots, l_n) = \text{join}(l_1, \text{join}(l_2, \dots, \text{join}(l_{n-1}, l_n) \dots)).$$

5.6 Finite Mappings

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

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

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 $\mathbb{IM}(A, B)$ such that

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 $TypeVarId$).

5.7 Ordinal Numbers

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

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

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 (End72) the natural numbers are represented as follows:

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

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

The usual ordering on natural numbers can be defined by:

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

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.

5.8 Definition by Transfinite Induction

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

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

Here the factorial function $! : \omega \rightarrow \omega$ 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:

- a) $H(0) = a$.
- b) $H(S(\beta)) = F(H(\beta))$ for every β such that $S(\beta) < \alpha$.
- c) $H(\beta) = G(H \upharpoonright \beta)$ for every limit ordinal $\beta < \alpha$.

5.9 Cardinality and Cardinal Numbers

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

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 $\text{card}(S)$, called the *cardinality* of S , such that S and $\text{card}(S)$ 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 $\text{card}(S) < \omega$; *countable* if $\text{card}(S) = \omega$; and *uncountable* if $\omega < \text{card}(S)$.

5.10 Structured Expressions

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

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

```

if  $T$  then  $e_1$  else  $e_2$  =  $e_1$ 
if  $F$  then  $e_1$  else  $e_2$  =  $e_2$ .

```

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

```

if  $P \wedge Q$ 
then cons
else altn

```

it will correspond to:

```

if  $P$ 
then if  $Q$ 
      then cons
      else altn
else altn

```

For disjunction the corresponding explanation is:

```

if  $P \vee Q$ 
then cons
else altn

```

it will correspond to:

```

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

```

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

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

The **let** construct

```

let  $x = e$  in  $e'$ 

```

can be read as a shorthand for:

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

The **cases** construct:

```

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

```

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 (clause 8 contains the formal definitions for *MkTag*, *StripTag*, and *ShowTag* if they are required for the understanding of the following). The expression:

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

corresponds to:

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

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

Similarly the expression:

```

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

```

corresponds to:

```

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

```

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

$t :: some_selector : x$

5.11 Semantic Function and Predicate Definitions

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

5. *FunctionName* : *Type*
 - 5.1 *FunctionName*(*arguments*) $\triangleq$
 - .2 *Function-definition*
- Annotations to *FunctionName*:**
5. Function-annotations
- End of annotations**

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

- FunctionName* : *Type*
 6. *FunctionName* must satisfy :
 - (1) *Axiom(s)*
- Annotations to *FunctionName*:**
6. Function-annotations
- End of annotations**

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

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

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

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

```

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

```

instead of

```

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

```

5.12 Use of Recursion

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

6 Core Abstract Syntax

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

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

6.1 Document

Document = *Definitions*

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

6.2 Definitions

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.

6.2.1 Type Definitions

TypeDef :: *shape* : *Type*
 inv : *InvId* × *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.

Type = *BasicType* | *CompositeType* | *ProductType* | *UnionType* | *OptionalType* |
 SetType | *SeqType* | *MapType* | *FnType* | *TypeId* | *TypeVar* | *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.

BasicType :: *bt* : NATONE | NAT | INTEGER | RAT | REAL | BOOLEAN | CHAR | TOKEN | 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 clause 7.2.1).

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

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

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

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

ProductType :: *fields* : *Type**

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

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

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

OptionalType :: *tp* : *Type*

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

SetType :: *elemtp* : *Type*

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

SeqType = *Seq0Type* | *Seq1Type*

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

Seq0Type :: *elemtp* : *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*.

Seq1Type :: *elemtp* : *Type*

Seq1Type 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*

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

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

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

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

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

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

TypeId :: *id* : *ValueId*

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

TypeVar :: *var* : *TypeVarId*

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

QuoteType :: *lit* : *QuoteLit*

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

6.2.2 State Definition

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

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

6.2.3 Value Definitions

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

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

6.2.4 Function Definitions

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

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* : Type Var⁺
 heading : ImplFnHeading
 pre : Expr
 post : Expr

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

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

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

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

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

Par :: *id* : ValueId
 type : Type

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

Observe that implicitly defined functions cannot be Curried.

6.2.5 Operation Definitions

Observe that:

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

ExplOpDef :: *heading* : OpHeading
 body : Stmt

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

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

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

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

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

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

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

ReadWriteMode = READ | READWRITE

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

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

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

6.3 Expressions

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

Expr is an expression.

6.3.1 Local Binding Expressions

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

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

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

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

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

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

6.3.2 Conditional Expressions

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

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

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

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

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

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

6.3.3 Unary Expressions

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

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

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

$UnaryOp$ is an ‘unary operator’.

6.3.4 Binary Expressions

$BinaryExpr :: left : Expr$
 $opr : BinaryOp$
 $right : Expr$

$BinaryExpr$ is a ‘binary expression’. It consists of a left hand side expression, $left$, a binary operator, opr , and a right hand side expression, $right$.

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

$BinaryOp$ is a ‘binary operator’.

6.3.5 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.

$AllOrExistsExpr :: quant : Quantifier$
 $bind : Bind\text{-}set$
 $pred : Expr$

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

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

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

$ExistsUniqueExpr :: bind : Bind$
 $pred : Expr$

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

6.3.6 Iota Expression

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

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

6.3.7 Set Expressions

SetEnumeration :: *els* : *Expr**

SetEnumeration is a set enumeration expression which simply consists of a sequence¹⁾ of element expressions, *els*.

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

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

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

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

6.3.8 Sequence Expressions

SeqEnumeration :: *els* : *Expr**

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

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

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

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

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

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

6.3.9 Map Expressions

MapEnumeration :: *els* : *Maplet**

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

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

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

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

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

6.3.10 Tuple Constructor

TupleConstructor :: *fields* : *Expr*⁺

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

6.3.11 Record Expressions

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

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

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

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

6.3.12 Apply Expressions

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

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

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

¹⁾In Jon86 these are called μ -functions.

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

$$\begin{array}{ll} \text{FctTypeInst} :: id & : Id \\ & tpinst : Type^+ \end{array}$$

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

6.3.13 Lambda Expression

$$\begin{array}{ll} \text{Lambda} :: parm & : Par \\ & body : Expr \end{array}$$

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

6.3.14 Is Expression

$$\begin{array}{ll} \text{IsExpr} :: tag & : ValueId \mid BasicType \\ & arg : Expr \end{array}$$

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

6.3.15 Literals

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

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

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

NumLit is a ‘numeric literal’. It is any numeric value.

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

BoolLit is a ‘Boolean literal’. It is either TRUE or FALSE.

$$CharLit :: val : CHAR$$

CharLit is a ‘character literal’. It is a character.

$$TextLit :: val : CharLit^*$$

TextLit is a ‘text literal’. It is a sequence of ‘character literals’.

$$QuoteLit :: val : CharLit^*$$

QuoteLit is a ‘quote literal’. It is represented as a token.

$$NilLit :: val :$$

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

6.3.16 Identifiers

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

$$Id = ValueId \mid PreId \mid PostId \mid InvId \mid InitId \mid MkId \mid IsId \mid \\ StateTypeId \mid StateConstId \mid RecSelId \mid RecModId \mid RecConsId \mid TypeEnvId \mid TagEnvId$$

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).

6.4 State Designators

$StateDesignator = ValueId \mid FieldRef \mid MapOrSeqRef$

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

$FieldRef :: sd : StateDesignator$
 $sel : ValueId$

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

$MapOrSeqRef :: sd : StateDesignator$
 $sel : Expr$

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

6.5 Statements

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

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

6.5.1 Local Binding Statements

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

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

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

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

$DefStmt :: lhs : Pattern$
 $rhs : Expr \mid Call$
 $in : Stmt$

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*.

6.5.2 Block and Assignment Statements

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

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

Sequence :: *stmts* : *Stmt**

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

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

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

6.5.3 Conditional Statements

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

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

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

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

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

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

6.5.4 Loop Statements

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

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

LoopDirection = FORWARDS | BACKWARDS

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

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

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

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

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

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

6.5.5 Non-Deterministic Sequences

NonDetStmt :: *stmts* : *Stmt*⁺

NonDetStmt is a ‘non-deterministic statement’. It consists of a non-empty sequence of statements¹⁾, *stmts*. It denotes an arbitrary ordering of execution of these statements.

6.5.6 Call and Return Statements

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

Call is a statement calling an operation. It consists of an identifier (the name of the operation), *oprt*, an argument expression, *arg*, and an optional state designator indicating which part of the state it is acting upon, *state* (it will only be 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.

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

6.5.7 Exception Handling Statements

$HandlerStmt = Always \mid TrapStmt \mid RecTrapStmt$

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

$Always :: post : Stmt$
 $body : Stmt$

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

$TrapStmt :: pat : Pattern$
 $post : Stmt$
 $body : Stmt$

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

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

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

$Exit :: expr : [Expr]$

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

6.6 Patterns and Bindings

$Pattern = PatternId \mid MatchVal \mid SeqPattern \mid TuplePattern \mid$
 $RecordPattern \mid SetPattern \mid ConstrPattern$

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

$PatternId :: id : [ValueId]$

PatternId is a ‘pattern identifier’. It is either simply an identifier or a “don’t-care”.

$MatchVal :: val : Expr$

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

$SeqPattern = SeqEnumPattern \mid SeqConcPattern$

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

$SeqEnumPattern :: els : Pattern^*$

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

$$\begin{array}{l} \textit{SeqConcPattern} :: lp : \textit{Pattern} \\ \phantom{\textit{SeqConcPattern} :: } rp : \textit{Pattern} \end{array}$$

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

$$\textit{TuplePattern} :: fields : \textit{Pattern}^+$$

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

$$\begin{array}{l} \textit{RecordPattern} :: tag : \textit{MkId} \\ \phantom{\textit{RecordPattern} :: } fields : \textit{Pattern}^* \end{array}$$

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).

$$\textit{SetPattern} = \textit{SetEnumPattern} \mid \textit{SetUnionPattern}$$

SetPattern is a ‘set pattern’. It is either a ‘set enumeration pattern’ or a ‘set union pattern’.

$$\textit{SetEnumPattern} :: els : \textit{Pattern}^+$$

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

$$\begin{array}{l} \textit{SetUnionPattern} :: lp : \textit{Pattern} \\ \phantom{\textit{SetUnionPattern} :: } rp : \textit{Pattern} \end{array}$$

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

$$\textit{ConstrPattern} = \textit{SetConstrPattern} \mid \textit{TypeConstrPattern}$$

ConstrPattern is a ‘constrained pattern’. It is either a ‘set constrained pattern’ or a ‘type constrained pattern’.

$$\begin{array}{l} \textit{SetConstrPattern} :: pat : \textit{Pattern} \\ \phantom{\textit{SetConstrPattern} :: } set : \textit{Expr} \end{array}$$

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

$$\begin{array}{l} \textit{TypeConstrPattern} :: pat : \textit{Pattern} \\ \phantom{\textit{TypeConstrPattern} :: } type : \textit{Type} \end{array}$$

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

$$\textit{Bind} = \textit{SetBind} \mid \textit{TypeBind}$$

Bind is a ‘binding’. It is either a ‘set binding’ or a ‘type binding’.

$$\begin{array}{l} \textit{SetBind} :: pat : \textit{Pattern} \\ \phantom{\textit{SetBind} :: } set : \textit{Expr} \end{array}$$

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.

7 Dynamic Semantic Domains

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

7.1 The Domain Universe

The domain universe which is introduced in this clause provides denotations for all values expressible in VDM-SL. This clause will first introduce the basic theory behind cpo's, and then a number of cpo operators will be presented. Then a universe of cpo's is built using transfinite induction and the defined cpo operators. In order to be able to interpret invariants semantically a universe of domains is finally constructed. This last clause also defines new domain operators²⁾ — operators which will be used in the definition of the dynamic semantics.

7.1.1 Basic Definitions

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

7.1.1.1 Complete Partial Orders and Fixed Point Definitions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 Yf where Y is a least fixed point operator yielding the least solution.

7.1.1.2 Operators on Complete Partial Orders

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

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

Definition 7.10 (Lifting) For any set S , the result of *lifting* S is a cpo $S_\perp$ 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 7.11 (Union-compatible cpo's) Let $\mathcal{A}$ be a family of cpo's. The family $\mathcal{A}$ is *union-compatible* if:

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

is a cpo.

Definition 7.12 (Union) Let $\mathcal{A}$ be a union-compatible family of cpo's with for all A in $\mathcal{A}$ $\perp_A = \perp$. 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 7.13 (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 7.14 (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 ... and $a_n \sqsubseteq_{A_n} a'_n$.

Definition 7.15 (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 $\text{len}(l_1) = \text{len}(l_2)$ and
for $i \in \{1, \dots, \text{len}(l_1)\}$, $l_1(i) \sqsubseteq_A l_2(i)$.

Definition 7.16 (Record space) Let $id \in Id$ be a VDM-SL identifier, and let $A_1, \dots, A_n$ 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 ... and $a_n \sqsubseteq_{A_n} a'_n$.

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

Definition 7.17 (Mapping space) Let A be a flat cpo and B be a cpo with $\perp = \perp_A = \perp_B$. 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 7.18 (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:

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