

XVSM: A Narrative and a Formalisation

A Technical Note, Version 5

Dines Bjørner
Fredsvej 11, DK-2840 Holte, Denmark
bjorner@gmail.com – www.imm.dtu.dk/~db

Started Mon. 19 April, 2010: Compiled: August 2, 2010: 14:56 ECT¹

¹This document presents work in progress. The document constitutes a technical note. It reports on an attempt to formalise XVSM: the Extensible Virtual Shared Memory as reported in the Dipl.Ing. thesis by Stefan Craß: *A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell – Design and Specification*. Technische Universität Wien, 05.02.2010.

Document History

- I began this formalisation “exercise” around April 19, 2010.
- A first draft (Version 1) sketch was issued April 23, 2010.
- It was commented upon by the originators of the *XVSM* concepts, Eva Kühn, Gernot Salzer and Stefan Craß on April 29.
- Then I left Vienna to return home on April 30.
- My next, corrected and slightly extended formalisation was communicated May 8 (Version 2)
- and commented upon by the “originators” on May 31.
- (Between May 8 and June 15 I was otherwise occupied: a two week lecture tour of Ukraine and a four day seminar trip to Budapest.)
- A June 24, 2010 version (Version 3) attempts to interpret the comments of May 31 while extending the formalisation. To this author’s great surprise, he had not, till late May, considered whether a type system, even a dynamic one, might be useful. It seems that the co-creators of *XVSM* also had not documented any such thinking. So Version 3 of this technical note started to investigate a type concept for *XVSM*.
- Version 4, with minor corrections and even minor extensions was circulated July 22, 2010.
- Changes of Version 5 wrt. Version 4 are marked with double thick change bars.
- Among these changes are:
 - ★ Book instead of article format.
 - ★ Subdivision into more chapters.
 - ★ Entries made for chapters on Core Application Programmer’s Interfaces (CAPI-1–CAPI-4).

Contents

Document History	3
1 Introduction	9
1.1 On Targets of Formal Specification	9
1.2 Why Specify Software Concepts Formally	10
1.3 An XVSM Type System	10
1.3.1 Type Systems	10
1.3.2 Static and Dynamic Type Systems	11
1.3.3 Why Type Systems	11
1.3.4 Words of Caution	11
2 XVSM Trees	13
2.1 XTree Syntax	13
2.1.1 XTree Rules	13
2.1.2 XTree Types	13
2.1.3 XTree Type Designator Wellformedness	14
2.1.4 XTree Type Functions	14
2.1.5 XTree Wellformedness	14
2.1.6 XTree Subtypes	15
3 XTree Operations	17
3.1 XTree Multiset Union	17
3.1.1 Commensurate Multiset Arguments	17
3.1.2 Type “Prediction”	18
3.1.3 A Theorem: Correctness of Type “Prediction”	18
3.2 XTree Multiset Equality	18
3.3 XTree Multiset Subset	18
3.4 Property Multiset Membership	19
3.5 XTree Multiset Membership	19
3.6 XTree Multiset Cardinality	19
3.7 Arbitrary Selection of XTrees or Properties from Multisets	19
3.8 XTree Multiset Difference	20
3.9 XTree List Concatenation	20
3.10 XTree List Equality	20
3.11 XTree List Property Membership	20
3.12 XTree List XTree Membership	21
3.13 XTree List Length	21
3.14 XTree List Head	21
3.15 XTree List Tail	21
3.16 XTree List N th Element	21
4 Indexing	23
4.1 Paths and Indexes	23
4.2 Proper Index	23
4.3 Index Selecting	24
4.4 Path Indexing	24
5 Queries	25
5.1 Generally on Semantics	25
5.2 Syntax: Simple XVSM Queries	25
5.2.1 Syntax: Predefined Selector Queries	26
5.2.2 Semantics: Predefined Selector Queries	26
Count	26

	Sort Up	27
	MUCH MORE TO COME	27
6	CAPI-1: Basic Operations	29
6.1	A “Storage” Model	30
6.2	The CAPI-1 Operations	31
6.2.1	writeL1	32
6.2.2	write1	33
6.2.3	writeBulk1	34
6.2.4	read1	35
6.2.5	take1	36
6.3	CAPI-1: A Discussion	37
7	CAPI-2: Transactions	39
7.1	Transaction Model	39
7.2	Locking	39
7.3	Logging	39
7.4	Operations	40
7.4.1	Transactional Management Operations	40
	createTransaction2	40
	commitTransaction2	40
	rollbackTransaction2	40
	startTransaction2	40
	finishTransaction2	40
	cancelTransaction2	40
7.4.2	Transactional Space Operations	41
	writel2	41
	write2	41
	writeBulk2	41
	read2	41
	take2	41
7.4.3	Explicit Locking Operations	42
	setExclusiveLock2	42
	getReadable2	42
	getTakeable2	42
7.5	CAPI-2: A Discussion	42
8	CAPI-3: Coordination	43
8.1	Coordinator Interface	44
8.1.1	Init Function	45
8.1.2	Accountant Functions	46
	onInsert Function	47
	onRead Function	48
	onRemove Function	49
	onDataReturn Function	50
8.1.3	Query Function	51
8.2	Container Operations	52
8.2.1	createContainer3 Operation	53
8.2.2	destroyContainer3 Operation	54
8.2.3	lookupContainer3 Operation	55
8.2.4	setContainer3 Operation	56
8.2.5	write3 Operation	57
8.2.6	take3 Operation	58
8.2.7	read3 Operation	59

8.2.8	delete4 Operation	60
8.3	Predefined Coordinators	61
8.3.1	SystemCoordinator	62
8.3.2	QueryCoordinator	63
8.3.3	FifoCoordinator	64
8.3.4	KeyCoordinator	65
8.3.5	LabelCoordinator	66
8.3.6	LindaCoordinator	67
8.3.7	VectorCoordinator	68
8.4	Custom Coordination	69
8.5	CAPI-3: A Discussion	70
9	CAPI-4: Runtime Model	71
9.1	Request Scheduling	72
9.2	Aspects	73
9.3	CAPI-4 Operations	74
9.3.1	Transaction Management	75
	createTransaction	76
	commitTransaction	77
	rollbackTransaction	78
9.3.2	Aspect Management	79
	addAspect	80
	removeAspect	81
9.3.3	Container Operations	82
	createContainer	83
	destroyContainer	84
	lookupContainer	85
	setContainerLock	86
	write	87
	take	88
	read	89
	delete	90
9.3.4	Management API	91
9.4	CAPI-4: A Discussion	92
10	Closing	93
10.1	What Have We Covered ?	93
10.2	What Have We Not Covered ?	93
10.3	What To Do Next ?	93
10.4	Acknowledgement	93
10.5	Bibliographical Notes	93
10.5.1	Description Languages	93
10.5.2	References	94
A	RSL: The Raise Specification Language — A Primer	97
A.1	Type Expressions	97
A.1.1	Atomic Types	97
	Basic Types:	97
	Concrete Composite Types	97
	Composite Type Expressions:	97
A.1.2	Concrete Types	99
	Type Definition:	99
	Variety of Type Definitions:	99
	Record Types:	99

	Subtypes:	100
	Sorts:	100
A.2	Propositional Expressions	100
	Propositional Expressions:	100
	Simple Predicate Expressions:	100
	Quantified Expressions:	100
A.3	Arithmetic	101
	Arithmetic:	101
A.3.1	Set Enumerations	101
	Set Enumerations:	101
	Set Comprehension:	101
A.3.2	Cartesian Enumerations	101
	Cartesian Enumerations:	102
A.3.3	List Enumerations	102
	List Enumerations:	102
	List Comprehension:	102
A.3.4	Map Enumerations	102
	Map Enumerations:	102
	Map Comprehension:	103
A.3.5	Set Operator Signatures	103
	Set Operations:	103
	Set Examples:	103
	Set Operation Definitions:	104
	Cartesian Operations:	104
A.3.6	List Operator Signatures	105
	List Operations:	105
	List Examples:	105
A.3.7	Map Operator Signatures and Map Operation Examples	106
	Map Operation Redefinitions:	107
A.4	The λ-Calculus Syntax	108
	λ -Calculus Syntax:	108
	Free and Bound Variables:	108
	Substitution:	108
	α and β Conversions:	109
	Sorts and Function Signatures:	109
	Explicit Function Definitions:	109
	Implicit Function Definitions:	109
A.5	Simple let Expressions	110
	Let Expressions:	110
	Recursive let Expressions:	110
	Predicative let Expressions:	110
	Patterns:	110
	Conditionals:	111
	Operator/Operand Expressions:	111
A.6	Statements and State Changes	111
	Statements and State Change:	111
	Variables and Assignment:	112
A.7	Statement Sequences and skip	112
	Statement Sequences and skip :	112
A.8	Imperative Conditionals	112
	Imperative Conditionals:	112
	Iterative Conditionals:	112
A.9	Iterative Sequencing	112
	Iterative Sequencing:	112

A.10 Process Channels	112
Process Channels:	113
Process Composition:	113
Input/Output Events:	113
Process Definitions:	113
Simple RSL Specifications:	114

Chapter 1

Introduction

XVSM, the Extensible Virtual Shared Memory concept, has been described in a number of conference proceeding publications: [33, 34, 20, 8]. The MSc Thesis [19] claims to present a formal model, but what is presented is not a proper formal model. To be a proper formal model there must be an abstract presentation in some formal, that is, mathematically well defined specification language and there must be a formal proof system for that language. Usually a formal semantics is also an abstract specification. `Haskell`, although a commendable programming language, is not suited for the specification of a semantics of *XVSM*, and [19] presents a `Haskell` implementation of *XVSM* and not an abstraction. A reasonably precise, even readable (and also executable), definition of *XVSM* could have been done in `Haskell`. Such a definition would carefully build up definitions, in `Haskell`, of the syntax of *XVSM* `XTrees`, of *XVSM* queries, etc. We shall present a formal definition of *XVSM* in `RSL`, the Raise Specification Language [23, 24].

1.1 On Targets of Formal Specification

Formalisation of software concepts started in the 1960s. The most notable example was that of the formal (operational semantics) description of the PL/I programming language [5, 6, 7]. The ULD notation emerged (ULD I, ULD II, ULD III -- ULD for Universal Language Description). This name of notation was later renamed into VDL (Vienna Language Description) by J.A.N. Lee [36]. Peter Lucas (sometimes with Kurt Walk) reviewed the [43, 38, 37, 39, 40, 41, 42] semantics descriptions of notably ULD III and the background for VDM (the Vienna Development Method).

As a result of the VDL (research and experimental development) work the IBM Vienna Laboratory undertook, in 1973, to develop, for the IBM market a new PL/I compiler for a new IBM computer (code named FSM: Future Systems machine). The US and European IBM laboratories' development of this computer was, eventually, curtailed, in February 1974. Nevertheless, the IBM Vienna Laboratory, was able to complete the work on a formal (denotational semantics-like) description of PL/I [4]. This work led to VDM [16, 30, 17, 31, 32, 21] – which later led to RAISE [23, 24] (1990). All the other now available formal specification languages came after VDM: Alloy [29] (2000), B, Event B [2] (1990, 2000) and Z [47] (1980).

First with VDM and now, as here, with `RSL`, formal specification has been used – other than for the semantics description of programming languages – first for formalising software designs, then for formalising requirements for general software, and for formalising (their) domain descriptions.

In this technical note we apply, not for the first time, formal specification to what the proposers of *XVSM* refers to as *middleware*: computer software that connects software components or applications. The software consists of a set of services that allows multiple processes running on one or more machines to interact (including sharing data). *Middleware* technology evolved to provide for interoperability in support of the move to coherent distributed architectures, which are most often used to support and simplify complex distributed applications. It includes web servers, application servers, and similar tools that support application development and delivery.

Middleware is especially integral to modern information technology based on XML, SOAP, Web services, and service-oriented architecture.

1.2 Why Specify Software Concepts Formally

A number of independent reasons can be given for why one might wish to formally specify a software concept¹. We itemize some of these:

- **As a design aid:** In researching and experimentally developing the design of a software concept, experiments with formal models of the software concept, or just some of its sub-concepts, have shown to help clarify and simplify many design issues².
- **As a communication document:** A suitably narrated and formalised specification, such as the present technical note lays a ground for (but is not yet), can be used as a, or the, ‘semantics’ specification for XVSM. It can serve as a standards document.
- **As a basis for implementation:** A suitably narrated and formalised specification, such as the present technical note, can serve as a basis for (thus provably) correct implementations of proper XVSM middleware.
- **As a basis for teaching & training:** An XVSM communication document can serve as the basis for instruction in the use (i.e., ‘programming’) of XVSM-dependent applications.
- **As a basis for proving properties of XVSM:** The formal specification of XVSM, such as attempted, or at least begun, with the present technical note, can be referred to in formal proofs of properties of XVSM and its applications.

1.3 An XVSM Type System

One of the great contributions of computing science to mathematics has been the studies made of type systems. And one of the great advances of software engineering from the middle 1950s till today has been the use of suitable, usually static, type systems.

The current author has (therefore) been quite surprised when discovering, that a language such as the XVSM query language and the *Core Application Programming Interface Languages*³ (such as CAPI-1, CAPI-2, and CAPI-3) has not been endowed by a type system. Instead of erroneous query and transaction results (here modelled by **chaos**) an XVSM programme could use these type testing facilities to secure provably correct uses of XVSM.

We shall, here and there, ‘divert’ from a straight line reformulation of [19], and present components of an XVSM Type System (XVSM/TS).

1.3.1 Type Systems

Many kinds of type systems can be proposed for XVSM. Defining a type system may imply that only correctly typed data, i.e., **XTrees**, and only arguments to operations: queries and actions, that, in some weak or strong sense, satisfy the signature (that is, the type) of the operation are allowed.

¹By a software concept we mean such concepts as (the semantics of) programming languages, database models or database management systems, operating systems, specific application systems [such as air traffic, banking, manufacturing, transportation, or other], their requirements, their underlying domains, etc.

²The current author offers the following observation (i) and advice (ii): (i) it seems that formalisation was not used in the conceptualisation of XVSM; and (ii) further extensions of XVSM should preferably be based on the present – or similar, reworked – formalisation and should itself use formal modelling. In reading publications about XVSM an experienced reader of precise descriptions too easily resolves that there are simply far too many ambiguous, incinsistent and incomplete description points: they may not be so, but the current english texts leaves such an experienced reader of precise descriptions to resolve so.

³An application programming interface (API) is an interface implemented by a software program which enables it to interact with other software.

(We then speak of a weakly, respectively strongly type language.) We shall, through the judicious use of concepts of sub, commensurate- and super-types, suggest one (of several possible) *XVSM* type systems. It is important to emphasize this: that either one of several *XVSM* type systems are possible. The one presented here may not be the best for a number of contemplated applications of *XVSM*, but it is probably a sensible one! Recommendable monographs cum textbooks on type systems and programming languages are [46, 44]. Further foundational studies of type systems are provided in the monographs [25, 1].

1.3.2 Static and Dynamic Type Systems

A programming language is said to use static typing when type checking can be performed during compile-time as opposed to run-time.

A programming language is said to be dynamically typed when the majority of its type checking can only be performed at run-time as opposed to at compile-time. In dynamic typing, values have types but variables do not (necessarily); that is, a variable can refer to a value of any type. Whether one can speak of *XVSM* variables is not known.

We shall anyway think of the type system that we shall put forward for *XVSM* as being a dynamic type system.

1.3.3 Why Type Systems

Reasons for endowing *XVSM* with a type system can be itemized:

- **Safety:** Checking, before execution, that an operation, with the types of its argument and the types of the space-based data, that is, *XTrees*, satisfy the type rules helps avoid otherwise meaningless operations.
- **Optimisation:** Static type-checking may provide useful compile-time information. Dynamic type-checking may provide useful run-time information.
- **Documentation:** In expressive type systems, types can serve as a form of documentation, since they can illustrate the intent of the programmer.
- **Abstraction (Modularity):** Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation.

Any one chosen type system will have been devised so as to satisfy at least one of the above reasons.

1.3.4 Words of Caution

The type system proposed here for *XVSM* is just an example. I am not quite sure that my particular design choices are the right ones for a system like *XVSM*. A perhaps more proper *XVSM* type system should evolve as the result of close, concentrated discussions and work, in Vienna, not over the Internet, between the leading authors of [33, 34, 20, 8, 19] and Dines Bjørner. But what I am rather sure of is that for *XVSM* to be considered a serious contender for so-called space-based computing *XVSM* must be endowed with a type system and with a suitable set of type system (run-time) operations.

Chapter 2

XVSM Trees

2.1 XTree Syntax

2.1.1 XTree Rules

1. There are labels and labels are further unspecified quantities.
2. Properties are pairs of labels and XTrees, that is, a property is such a pair.
3. An XTree is either an XTree value or an XTree multiset or an XTree sequence (an XTree list).
 - a) An XTree value is either some XTree text or is some XTree integer.
 - b) An XTree multiset consists of a multiset of properties.
 - c) An XTree sequence consists of a list of properties.

type

```
1 L
2 P = L × XT
3 XT = XV | XL | XS
3a XV == mk_ST(sel_txt:Text) | mk_IN(sel_i:Int)
3b XS == mk_XS(sel_xs:(P  $\overrightarrow{m}$  Nat))
3c XL == mk_XL(sel_xl:P*)
```

2.1.2 XTree Types

4. An XTree type is either
 - a) an integer type, or
 - b) a text type, or
 - c) a multiset type which maps its entry labels into corresponding XTree type, or
 - d) a sequence type which is a sequence of labelled XTree types.

type

```
4 XTTy = IntTy | TxtTy | MulTy | SeqTy
4a IntTy == mkITy
4b TxtTy == mkTTy
4c MulTy == mk_MTy(m:(L  $\overrightarrow{m}$  XTTy))
4d SeqTy == mk_STy(m:(L × XTTy)*)
```

XTTy are type designators.

2.1.3 XTree Type Designator Wellformedness

5. A type designator, i.e., any XTTy is wellformed if it satisfies the following conditions:
- Integer and text type designators are wellformed.
 - Multiset type designators are wellformed if the type designators for any label are wellformed.
 - Sequence type designators are wellformed if all labelled type designators are wellformed and if the type designators for identically labelled entries are the same type.¹

value

5. $\text{wf_XTTy}: \text{XTTy} \rightarrow \mathbf{Bool}$
 5. $\text{wf_XTTy}(t) \equiv$
 5. **case** t **of**
 5a. $\text{mkITy} \rightarrow \mathbf{true}$,
 5a. $\text{mkTTy} \rightarrow \mathbf{true}$,
 5b. $\text{mk_MTy}(\text{tym}) \rightarrow \forall t': \text{XTTy} \bullet t' \in \mathbf{rng} \text{ tym} \Rightarrow \text{wf_XTTy}(t')$
 5c. $\text{mk_STy}(\text{tyl}) \rightarrow$
 5c. $\quad \forall (l', t'): (\mathbf{L} \times \text{XTTy}) \bullet (l', t') \in \mathbf{elems} \text{ tyl} \Rightarrow \text{wf_XTTy}(t') \wedge$
 5c. $\quad \forall (l'', t''): (\mathbf{L} \times \text{XTTy}) \bullet (l'', t'') \in \mathbf{elems} \text{ tyl} \Rightarrow \text{xtr_type}(t') = \text{xtr_type}(t'')$ **end**

2.1.4 XTree Type Functions

6. Given an XTree one can “extract” its type:
- The type of an integer value is mkITy .
 - The type of a text value is mkTTy .
 - The type of an XTree multiset, ms , is $\text{mk_MTy}(\text{tym})$ where tym is a mapping from the labels of ms to the XTree type of the corresponding values.²
 - The type of an XTree sequence, sq , is $\text{mk_STy}(\text{tys})$ where tys is a sequence of labelled XTree types of the indexed (and labelled) XTree values of the sequence.

value

6. $\text{xtr_type}: \text{XT} \xrightarrow{\sim} \text{XTTy}$
 6. $\text{xtr_type}(xt) \equiv$
 6. **case** xt **of**
 6a. $\text{mk_IN}(\text{intg}) \rightarrow \text{mkITy}$,
 6b. $\text{mk_ST}(\text{text}) \rightarrow \text{mkTTy}$,
 6c. $\text{mk_XS}(xs) \rightarrow \text{mk_MTy}([\!|l \mapsto \text{xtr_type}(xt') \!| \!|: \mathbf{L} \bullet l \in \mathbf{dom} \text{ xs} \wedge xt' \in \text{xs}(l)])$,
 6d. $\text{mk_XL}(xl) \rightarrow \text{mk_STy}([\!|i: \mathbf{Nat} \bullet i \in \mathbf{inds} \text{ xl} \wedge \text{xl}(i) = (l, xt') \!| \!|)$
 6. **end**
 6. **pre:** $\text{type_conform}(xt)$

2.1.5 XTree Wellformedness

7. An XTree is type_conformant if
- it is an integer, or
 - it is a text, or

¹**Note:** This constraints is in line with the constraint of Item 4c on the preceding page.

²**Note:** Thus we constrain two or more properties with the same label to be of the same type – or, as we shall see, subtypes of such a type. This is a consequence of Item 4c on the previous page.

- c) it is a multiset all of whose **XTrees** are `type_conformant` and all identically labelled **XTrees** have the same type, or
- d) it is a sequence all of whose **XTrees** are `type_conformant` and all of whose identically labelled **XTrees** have the same type.

value

```

7. type_conform: XT → Bool
7a. type_conform(xt) ≡
7.   case xt of
7a.     mk_IN(intg) → true,
7b.     mk_ST(text) → true,
7c.     mk_XS(xs) →
7c.       ∀ (l',xt'),(l'',xt''):(L×XT)•{(l',xt'),(l'',xt'')} ⊆ dom xs ∧
7c.       type_conform(xt') ∧
7c.a      l'=l'' ⇒ xtr_type(xt') = xtr_type(xt''),
7d.     mk_XL(xl) →
7d.       ∀ (l',xt'),(l'',xt''):(L×XT)•{(l',xt'),(l'',xt'')} ⊆ elems xl ∧
7d.       type_conform(xt') ∧
7d.a      l'=l'' ⇒ xtr_type(xt')=xtr_type(xt'')
7.   end

```

Discussion: Whether, in formula lines 7c.a and 7d.a, to insist on equality of types or to allow one type to be a subtype of the other (whichever way) is a question to be considered.

2.1.6 XTree Subtypes

8. We define a subtype relation as a relation between a pair of type designators:

- a) The **XTree** integer type is (i.e., designates) a subtype of itself.
- b) The **XTree** text type is (i.e., designates) a subtype of itself.
- c) Let two multiset type designators be `mk_MTy(tym')` and `mk_MTy(tym'')`.
`mk_MTy(tym')` is (i.e., designates) a subtype of `mk_MTy(tym'')`
 - i. if the definition set of labels of `mk_MTy(tym')` is a subset of the definition set of labels of `mk_MTy(tym'')`,
 - ii. and, if for identical labels, ℓ , in `mk_MTy(tym')` and `mk_MTy(tym''(\ell))` is (i.e., designates) a subtype of `mk_MTy(tym''(\ell))`.
- d) Let two sequence type designators be `mk_STy(tyl')` and `mk_STy(tyl'')`.
`mk_STy(tyl')` is (i.e., designates) a subtype of `mk_STy(tyl'')`
 - i. if the length of `tyl'` is less than or equal to the length of `tyl''`,³
 - ii. if for index positions, i , of `tyl'` the labels of the indexed properties `tyl'(i)` (= (l',t')) and `tyl''(i)` (= (l'',t'')) are the same ($l'=l''$) and
 - iii. type designator t' is (i.e., designates) a subtype of t'' .
- e) Only such pairs of types as implied by the above can possibly enjoy a subtype relation.

value

```

8. is_subtype: XXTy × XXTy → Bool
8. is_subtype(ta,tb) ≡
8.   case (ta,tb) of
8a.     (mkITy,mkITy) → true,
8b.     (mkTTy,mkTTy) → true,

```

³We could, instead of this “prefix” subtype property, have defined an “embedded” subtype property: that `tyl'` is a subtype of a properly embedded sequence of `tyl''`

```

8c. (mk_MTy(tym'),mk_MTy(tym'')) →
8(c)i.   dom tym' ⊆ tym'' ∧
8(c)ii.  ∀ l:L•l ∈ dom tym' ⇒ is_subtype(tym'(l),tym''(l)),
8d. (mk_STy(tyl'),mk_STy(tyl'')) →
8(d)i.   len tyl' ≤ tyl'' ∧
8(d)ii.  ∀ i:Nat • 1 ≤ i ≤ len tyl' ⇒
8(d)ii.   let ((l',t'),(l'',t''))=(tyl'(i),tyl''(i)) in
8(d)ii.   l'=l'' ∧
8(d)iii.  is_subtype(t',t'') end,
8e.   _ → false
8.   end

```

Please note that if td' and td'' are type designators, then either td' denotes a subtype of td'' or td'' denotes a subtype of td' or neither denotes a subtype of the other.

Chapter 3

XTree Operations

3.1 XTree Multiset Union

9. By the union of two multisets we understand their bag (i.e., multiset) union.
 - a) For any property which is common to both multisets the multiset union maps the property into the sum of its number of occurrences in the two argument multisets.
 - b) For any property which is only in one of the multisets the multiset union contains that property with the number of occurrences designated by that multiset.
 - c) Shared label values must be of comparable_types.

value

- ```
9 XSunion: XS × XS → XS
9a XSunion(mk_XS(xs1),mk_XS(xs2)) ≡
9a mk_XS([p↦xs1(p)+xs2(p)|p ∈ dom xs1 ∩ dom xs2]
9b ∪ xs1\dom xs2 ∪ xs2\dom xs1)
9c pre: comparable_types(xtr_type(mk_XS(xs1)),xtr_type(mk_XS(xs2)))
```

#### 3.1.1 Commensurate Multiset Arguments

10. Two multiset values (types) are comparable
11. if for identical (i.e., shared) labels have identical types (are equal);
12. or maybe we should just ask for an appropriate subtype relation.

**value**

- ```
10. comparable_values: XS × XS → Bool
10. comparable_values(mk_XS(lm'),mk_XS(lm'')) ≡
11.   ∀ l:L • l ∈ dom lm' ∩ lm'' ⇒
11.     (xtr_type(lm'(l)) = xtr_type(lm''(l))) ∨
12.     is_subtype(xtr_type(lm'(l)),xtr_type(lm''(l))) ∨
12.     is_subtype(xtr_type(lm''(l)),xtr_type(lm'(l)))
```

value

- ```
10. comparable_types: XTty × XTty → Bool
10. comparable_types(mk_XT(lmt'),mk_XT(lmt'')) ≡
11. ∀ l:L • l ∈ dom lmt' ∩ lmt'' ⇒
11. (lmt'(l) = lmt''(l)) ∨
12. is_subtype(lmt'(l),lmt''(l)) ∨ is_subtype(lmt''(l),lmt'(l))
```

### 3.1.2 Type “Prediction”

13. One can calculate the type of the result of a multiset union from its two arguments:

- a)
- b)
- c)
- d)

13.

13a.

13b.

13c.

13d.

### 3.1.3 A Theorem: Correctness of Type “Prediction”

14. One can prove the following theorem:

- a)
- b)
- c)
- d)
- e)

14.

14a.

14b.

14c.

14d.

14e.

## 3.2 XTree Multiset Equality

15. Multiset equality is bag equality of the multisets.

**value**

15 XSequal:  $XS \times XS \rightarrow \mathbf{Bool}$

15 XSequal(mk\_XS(xs1),mk\_XS(xs2))  $\equiv$  xs1 = xs2

## 3.3 XTree Multiset Subset

16. One multiset is a subset of another multiset

- a) if the first has a subset of the properties of the latter and
- b) and, for each property of the first its number of occurrences in the former is equal to or smaller than its number of occurrences in the latter.

**value**

16 XSsubset:  $XS \times XS \rightarrow \mathbf{Bool}$

16 XSsubset(mk\_XS(xs1),mk\_XS(xs2))  $\equiv$

16a  $\mathbf{dom} \text{ xs1} \subseteq \mathbf{dom} \text{ xs2} \wedge$

16b  $\forall p:P \cdot p \in \mathbf{dom} \text{ xs1} \Rightarrow \text{xs1}(p) \leq \text{xs2}(p)$

### 3.4 Property Multiset Membership

17. A property,  $p=(l,xt)$ , is in a multiset if it occurs in the multiset with a cardinality higher than 0.

**value**

```
17 XSmember: P × XS → Bool
17 XSmember(p,mk_XS(xs)) ≡ p ∈ dom xs ∧ xs(p)>0
```

### 3.5 XTree Multiset Membership

18. An **XTree**,  $xt$ , is a member of a multiset,  $xs$ , if there exists a label,  $\ell$  such that the property  $(\ell,xt)$  is a member of  $xs$ .

**value**

```
18 XSmember: XT × XS → Bool
18 XSmember(xt,mk_XS(xs)) ≡ ∃ l:L • XSmember((l,xt),mk_XS(xs))
```

### 3.6 XTree Multiset Cardinality

19. The cardinality of a multiset is the sum total of all the **XTrees** of distinct properties of that multiset.

**value**

```
19 XScard(mk_XS(xs)) ≡
19 if xs = [] then 0
19 else
19 let (l,xt):P•(l,xt) ∈ dom xs in
19 xs(l,xt) + XScard(mk_XS(xs \ {(l,xt)})) end end
```

### 3.7 Arbitrary Selection of XTrees or Properties from Multisets

20. To select an **XTree** of a multiset
- a) is undefined if the multiset is empty.
  - b) If it is not empty then an arbitrary property is chosen from the (definition set of the) multiset and the **XTree** of that property is yielded.
  - c) To select a property of a multiset basically follows the above description.

**value**

```
20 XSselectXT: XS ↪ XT
20 XSselectXT(mk_XS(xs)) ≡
20a if xs=[]
20a then chaos
20b else let (l,xt):P•(l,xt) ∈ dom xs in xt end
20b end
```

```
20 XSselectP: XS ↪ P
20 XSselectP(mk_XS(xs)) ≡
20a if xs=[]
```

```

20a then chaos
20c else let p:P•p ∈ dom xs in p end
20c end

```

### 3.8 XTree Multiset Difference

21. The multiset difference of two multisets,  $xs1$  and  $xs2$ ,
- is the multiset where properties that are in both  $xs1$  and  $xs2$  occur in the result with their number of occurrences being their difference, if larger than 0,
  - to which is joined the multiset of  $xs1$  whose properties are not in  $xs2$ .

**value**

```

21 XTreeDiff: XS × XS → XS
21 XTreeDiff(mk_XS(xs1),mk_XS(xs2)) ≡
21a mkXS(rm0([p→xs1(p)−xs2(p)|p:P•p ∈ dom xs1 ∩ dom xs2]))
21b ∪ xs1 \ dom xs2

```

```

rm0: (P \overline{m} Int) → (P \overline{m} Nat)
rm0(pmn) ≡ [p→pmn(p)|p:P•p ∈ dom pmn ∧ pmn(p)>0]

```

### 3.9 XTree List Concatenation

22. The concatenation of two XTree lists is the usual concatenation of lists.
23. Labels,  $\ell$ , common to the two XTree lists must designate XTree,  $xt1$  and  $xt2$  (i.e., properties  $(\ell,xt1)$  and  $(\ell,xt2)$ ) where one is a subtype of the other (i.e., including “vice versa”).

**value**

```

22 XTreeListConc: XL × XL → XL
22 XTreeListConc(mk_XL(xl1),mk_XL(xl2)) ≡ mk_XL(xl1^xl2)
23 pre ∀ (l1,xtt),(l2,xt2):P•(l1,xtt) ∈ elems xl1 ∧ (l2,xt2) ∈ elems xl2 ∧ l1=l2 ⇒
23 subtype(xt1,xt2) ∨ subtype(xt2,xt1)

```

### 3.10 XTree List Equality

24. The equality of two XTree lists is the usual equality of lists.

**value**

```

24 XTreeListEqual: XL × XL → Bool
24 XTreeListEqual(mk_XL(xl1),mk_XL(xl2)) ≡ xl1=xl2

```

### 3.11 XTree List Property Membership

25. A property is a member of an XTree list
26. if there is an index into the list which identifies that property.

**value**

```

25 XMemTreeList: P × XL → Bool
26 XMemTreeList(p,mk_XL(xl)) ≡ ∃ i:Nat • i ∈ inds xl ∧ p=xl(i)

```

### 3.12 XTree List XTree Membership

27. An `XTree` is a member of an `XTree` list
28. if there is an index into the list which identifies that `XTree`.

**value**

27 `XMbrTreeList`: `XT × XL → Bool`

28 `XMbrTreeList`(`xt,mk_XL(xl)`)  $\equiv \exists i:\mathbf{Nat},l:\mathbf{Label} \bullet i \in \mathbf{inds} \text{ xl} \wedge (l,xt)=xl(i)$

### 3.13 XTree List Length

29. The length of an `XTree` list
- a) is the length of the list it contains.

**value**

29 `XTreeListLength`: `XL → Nat`

29a `XTreeListLength`(`mk_XL(xl)`)  $\equiv \mathbf{len} \text{ xl}$

### 3.14 XTree List Head

30. The head, or first, element of an `XTree` list
- a) is the head property of the list it contains.

**value**

30 `XTreeListHead`: `XL → P`

30a `XTreeListHead`(`mk_XL(xl)`)  $\equiv \mathbf{if} \text{ xl}=\langle \rangle \mathbf{then} \mathbf{chaos} \mathbf{else} \mathbf{hd} \text{ xl} \mathbf{end}$

### 3.15 XTree List Tail

31. The tail, or rest, of an `XTree` list
- a) is the tail of the list it contains.

**value**

31 `XTreeListTail`: `XL → XL`

31a `XTreeListTail`(`mk_XL(xl)`)  $\equiv \mathbf{if} \text{ xl}=\langle \rangle \mathbf{then} \mathbf{chaos} \mathbf{else} \mathbf{mk\_XL}(\mathbf{tl} \text{ xl}) \mathbf{end}$

### 3.16 XTree List Nth Element

32. The  $n$ th element of a list
- a) if  $n$  is an index of the list then it is the property indexed by  $n$  else it is undefined.

**value**

32 `NthXTreeListElem`: `Nat × XL  $\overset{\sim}{\rightarrow}$  P`

32a `NthXTreeListElem`(`n,mk_XL(xl)`)  $\equiv \mathbf{if} \ 0 < n \leq \mathbf{len} \ \text{xl} \ \mathbf{then} \ \text{xl}(n) \ \mathbf{else} \ \mathbf{chaos} \ \mathbf{end}$



# Chapter 4

## Indexing

### 4.1 Paths and Indexes

- 33. An index is either a label or a wildcard or a ???
- 34. non-zero natural number.
- 35. A path is a finite sequence of zero, one or more indexes.

**type**

- 33 `Index == mk_L(l:L) | mk_WldCrd | mk_Nat(i:Nat1)`
- 34 `Nat1 = { |n:Nat•n>0| }`
- 35 `Path = Index*`

### 4.2 Proper Index

- 36. We define an `is_Index` predicate over indexes and `Xtrees`.
  - a) If there is a property,  $(\ell, xt)$ , which is in a multiset `mk_XS(xs)` then  $\ell$  is an index of that `mk_XS(xs)`.
  - b) If there is an index,  $j$ , into the list, `xl`, of an `XTree` list, `mk_XL(xl)`, then  $j$  is an index of that `mk_XL(xl)`;
  - c) if, furthermore, there is the property,  $(\ell, xt)$  at list `xl` position  $j$ , then  $\ell$  is an index into `mk_XL(xl)`; and
  - d) `mk_WldCrd` is (always) an index.

**value**

- `is_Index: Index × XT → Bool`
- `is_Index(i,xt) ≡`
  - case** `(i,xt)` **of**
  - 36a `(mk_L(l),mk_XS(xs)) → ≡ ∃ xt':XT•(l,xt') ∈ dom xs,`
  - 36b `(mk_Nat(j),mk_XL(xl)) → j ∈ inds xl,`
  - 36c `(mk_L(l),mk_XL(xl)) → ∃ j:Nat1,xt':XT•j ∈ inds xl∧xl(j)=(l,xt'),`
  - 36d `(mk_WldCrd,_) → true,`
  - `__ → false`
- end**

### 4.3 Index Selecting

37. Given an index  $i$  it thus may or may not identify an  $XTree$ ,  $xt'$ , or a property,  $p:P$ , of an argument  $XTree$ ,  $xt$ . The definition follows those of Items 36a–36c.

**value**

```

37 Identify: Index × XT \rightsquigarrow (XT|P)
37 Identify(i,xt) \equiv
37 case (i,xt) of
36a (mk_L(l),mk_XS(xs)) \rightarrow let $xt':XT \bullet (l,xt') \in \text{dom } xs$ in xt' end,
36b (mk_Nat1(i),mk_XL(xl)) $\rightarrow xl(i)$,
36c (mk_L(l),mk_XL(xl)) \rightarrow let $i:Nat1, xt':XT \bullet i \in \text{inds } xl \wedge xl(i) = (l,xt')$ in xt' end,
36d (mk_WldCrd, mk_XS(xs)) \rightarrow let $p:P \bullet p \in \text{dom } xs$ in p end,
36d (mk_WldCrd, mk_XL(xl)) \rightarrow hd xl
37 end
37 pre is_Index(i,xt)

```

### 4.4 Path Indexing

38. Given an  $XTree$ ,  $xt$ , a path,  $pth$ , may or may not identify an  $XTree$ ,  $xtr'$ , of  $xt$ . The selection function,  $Select$  is defined recursively:

- a) If the path is empty then the argument  $XTree$ ,  $xt$ , is yielded.
- b) If the head of the path is an index of the  $XTree$ ,  $xt$ , then the so indexed  $XTree$ ,  $xt_x$ , is selected.
- c) Otherwise the path,  $pth$ , is ill-defined.

**value**

```

38 Select: XT × Path \rightsquigarrow XT | P
38a Select(xtop,⟨⟩) \equiv xtop
38b Select(xt,⟨i⟩^pth) \equiv
38b if is_Index(i,xt)
38b then
38b let $e = \text{Identify}(i,xt)$ in
38b if $e:P \wedge pth \neq \langle \rangle$ then chaos end
38b Select(e,pth) end
38c else chaos end

```

# Chapter 5

## Queries

39. An *XVSM* query is a [piped] sequence of simple *XVSM* queries.

**type**

39  $Q = SQ^*$

### 5.1 Generally on Semantics

40. The idea is the following:

- a) The meaning of a simple *XVSM* query,  $sq:SQ$ , as applied to an *XTree*,  $xt:XT$ , is expressed as  $MSQ(sq)(xt)$ , and is to be an *XTree* multiset or an *XTree* list. **Not an *XTree* value ?**
- b) The meaning of an *XVSM* query,  $q:Q$ , as applied to an *XTree*,  $xt:XT$ , is expressed as  $MQ(q)(xt)$ , and is to be an *XTree* multiset or an *XTree* list.
- c) The meaning function,  $MQ$ , when applied to an empty query,  $\langle \rangle$ , is  $MQ(\langle \rangle)(xt)$ , that is,  $xt$ .
- d) The meaning function,  $MQ$ , when applied to a non-empty query,  $\langle sq \rangle^q$ , is  $MQ(q)(MSQ(sq)(xt))$ .
- e) Both  $MSQ$  and  $MQ$  may be undefined for some combinations of queries and *Xtrees*.

**value**

40a  $MSQ: SQ \rightarrow XT \rightsquigarrow XT$

40b  $MQ: Q \rightarrow XT \rightsquigarrow XT$

40b  $MSQ(sq)$  **as**  $xt$

40c  $MQ(\langle \rangle)(xt) \equiv xt$

40d  $MQ(\langle sq \rangle^q)(xt) \equiv MQ(q)(MSQ(sq)(xt))$

40e  $MQ(\langle sq \rangle^q)(xt) \equiv$

40e   **if**  $IS\_UNDEFINED(MSQ(sq)(xt))$

40e    **then**  $IS\_UNDEFINED(MQ(\langle sq \rangle^q)(xt))$

40e    **else** ... to be defined ...

40e    **end**

### 5.2 Syntax: Simple *XVSM* Queries

41. A simple *XVSM* query is either a selector query or a matchmaker query.

42. A [simple] selector [XVSM] query is either a predefined selector query or ...

**type**

41 SQ = SelQ | MatchQ

42 SelQ = PreSelQ | ...

### 5.2.1 Syntax: Predefined Selector Queries

43. A predefined selector query is either a count, a sort<sub>up</sub>, a sort<sub>down</sub>, a reverse, an identity, or a unique (selector) query.

- a) A count query states a non-zero natural number.
- b) A sort up query states a path.
- c) A sort down query states a path.
- d) A reverse query does not present an argument.
- e) An identity query does not present an argument.
- f) A unique (selector) query states a path.

43 PreSelQ = Cnt | SrtUp | SrtDo | Rev | Id | Uniq | ...

43a Cnt == mk\_Cnt(sel\_n: Nat)

43b SrtUp == mk\_SrtUp(sel\_p: Path)

43c SrtDo == mk\_SrtDo(sel\_p: Path)

43d Rev == mk\_Rev

43e Id == mk\_Id

43f Uniq == mk\_Uniq(sel\_p: Path)

### 5.2.2 Semantics: Predefined Selector Queries

#### Count

44. The mk\_Cnt(n) selector query applies to an XTree, xt, and,

- a) if it is an XTree list and if the list is of length n or more, yields the XTree list mk\_XL(xl') of the first n properties of xt = mk\_XL(xl), else it yields **chaos**; or
- b) if it is an XTree multiset and if the multiset has at least n properties, yields an XTree multiset, mk\_XS(xs'), of n arbitrarily chosen properties of xt = mk\_XS(xs), else it yields **chaos**.

44 MPreSelQ: PreSelQ → XT  $\rightsquigarrow$  XT

44 MPreSelQ(mk\_Cnt(n))(xt)  $\equiv$

44 **case** xt **of**

44a mk\_XL(xl) →

44a **if** len xl ≥ n **then** mk\_XL(⟨xl(i)|i: Nat•i:[1..n]⟩) **else** chaos **end**,

44b mk\_XS(xs) →

44b **if** card dom xs ≥ n

44b **then** let ps: P-set•card ps=n ∧ ps ⊆ dom xs **in**

44b mk\_XS([p ↦ xs(p)|p: P•p ∈ ps]) **end**

44b **else** chaos

44b **end**,

44b **—** → chaos

44b **end**

**Sort Up**

45. The `mk_SrtUp` selector query applies to a (relative) path,  $\text{pth}^\ell$ , and an `XTree`, `xt`.
- First we `Select` from `xt` the `XTree`, `xt''`, identified by the path `pth`.
  - The selected `XTree`, `xt''`, is either a list or a multiset.
  - The result of `MPreSelQ(mk_SrtUp(pthℓ))(xt)` is the `XTree` list `xt'` which has all the entries that `xt` has except that these are now ordered with respect to the ordering of the  $\ell$  values of `xt''`.

**value**

```

45 MPreSelQ: SrtUp → XT → XL
45 MPreSelQ(mk_SrtUp(pthℓ))(xt) ≡
45a let xt'' = Select(xt)(pth) in
45b let vl =

45c end end

```

MUCH MORE TO COME



## Chapter 6

# CAPI-1: Basic Operations



## 6.1 A “Storage” Model

## 6.2 The CAPI-1 Operations

**6.2.1 writeL1**

## 6.2.2 **write1**

### 6.2.3 **writeBulk1**

## 6.2.4 read1

**6.2.5 take1**

## 6.3 CAPI-1: A Discussion



## Chapter 7

# CAPI-2: Transactions



7.1 **Transaction Model**

7.2 **Locking**

7.3 **Logging**

## 7.4 Operations

### 7.4.1 Transactional Management Operations

**createTransaction2**

**commitTransaction2**

**rollbackTransaction2**

**startTransaction2**

**finishTransaction2**

**cancelTransaction2**

## 7.4.2 Transactional Space Operations

**writeL2**

**write2**

**writeBulk2**

**read2**

**take2**

### 7.4.3 Explicit Locking Operations

`setExclusiveLock2`

`getReadable2`

`getTakeable2`

## 7.5 CAPI-2: A Discussion

## Chapter 8

### **CAP1-3: Coordination**

## 8.1 Coordinator Interface

### 8.1.1 Init Function

## 8.1.2 Accountant Functions

**onInsert Function**

**onRead Function**

## onRemove Function

**onDataReturn Function**

### 8.1.3 Query Function

## 8.2 Container Operations

## 8.2.1 createContainer3 Operation

## 8.2.2 **destroyContainer3** Operation

### 8.2.3 lookupContainer3 Operation

## 8.2.4 **setContainer3 Operation**

## 8.2.5 **write3** Operation

## 8.2.6 **take3** Operation

## 8.2.7 read3 Operation

## 8.2.8 delete4 Operation

## 8.3 Predefined Coordinators

### 8.3.1 **SystemCoordinator**

### 8.3.2 QueryCoordinator

### 8.3.3 **FifoCoordinator**

### 8.3.4 KeyCoordinator

### 8.3.5 **LabelCoordinator**

### 8.3.6 LindaCoordinator

### 8.3.7 **VectorCoordinator**

## 8.4 Custom Coordination

## 8.5 CAPI-3: A Discussion

## Chapter 9

# **CABI-4: Runtime Model**

## 9.1 Request Scheduling

## 9.2 Aspects

### **9.3 CAPI-4 Operations**

### 9.3.1 Transaction Management

**createTransaction**

**commitTransaction**

**rollbackTransaction**



### 9.3.2 Aspect Management

**addAspect**

**removeAspect**

### **9.3.3 Container Operations**

**createContainer**

**destroyContainer**

**lookupContainer**

**setContainerLock**

**write**

**take**

**read**

**delete**



### 9.3.4 Management API

## 9.4 CAPI-4: A Discussion

# Chapter 10

## Closing

### 10.1 What Have We Covered ?

### 10.2 What Have We Not Covered ?

### 10.3 What To Do Next ?

### 10.4 Acknowledgement

### 10.5 Bibliographical Notes

#### 10.5.1 Description Languages

Besides using as precise a subset of a national language, as here English, as possible, and in enumerated expressions and statements, we have “paired” such narrative elements with corresponding enumerated clauses of a formal specification language. We have been using the RAISE Specification Language, RSL, [24], in our formal texts. But any of the model-oriented approaches and languages offered by

- Alloy [29],
- **B, Event B** [2],
- VDM [22] and
- Z [47],

should work as well.

No single one of the above-mentioned formal specification languages, however, suffices. Often one has to carefully combine the above with elements of

- Petri Nets [45],
- CSP: Communicating Sequential Processes [27],
- MSC: Message Sequence Charts [28],
- Statecharts [26],
- and some temporal logic, for example
  - ★ DC: Duration Calculus [48]

★ or TLA+ [35].

Research into how such diverse textual and diagrammatic languages can be meaningfully and proof-theoretically combined is ongoing [3]. And even then !

### 10.5.2 References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, NY, USA, August 1996.
- [2] J.-R. Abrial. *The B Book: Assigning Programs to Meanings and Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, England, 1996 and 2009.
- [3] K. Araki et al., editors. *IFM 1999–2009: Integrated Formal Methods*, volume 1945, 2335, 2999, 3771, 4591, 5423 (only some are listed) of *Lecture Notes in Computer Science*. Springer, 1999–2009.
- [4] H. Bekič, D. Bjørner, W. Henhapl, C. B. Jones, and P. Lucas. A Formal Definition of a PL/I Subset. Technical Report 25.139, IBM Laboratory, Vienna, December 1974.
- [5] H. Bekič, P. Lucas, K. Walk, and M. Others. Formal Definition of PL/I, ULD Version I. Technical report, IBM Laboratory, Vienna, 1966.
- [6] H. Bekič, P. Lucas, K. Walk, and M. Others. Formal Definition of PL/I, ULD Version II. Technical report, IBM Laboratory, Vienna, 1968.
- [7] H. Bekič, P. Lucas, K. Walk, and M. Others. Formal Definition of PL/I, ULD Version III. IBM Laboratory, Vienna, 1969.
- [8] S. Bessler, E. Kühn, R. Mordinyi, and S. Tomic. Using tuple-spaces to manage the storage and dissemination of spatial-temporal content. *Journal of Computer and System Sciences*, page 10, February 2010. Link: <http://dx.doi.org/10.1016/j.jcss.2010.01.010>.
- [9] D. Bjørner. Programming in the Meta-Language: A Tutorial. In D. Bjørner and C. B. Jones, editors, *The Vienna Development Method: The Meta-Language*, [16], LNCS, pages 24–217. Springer-Verlag, 1978.
- [10] D. Bjørner. Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a PL/I-like On-Condition Language Definition. In D. Bjørner and C. B. Jones, editors, *The Vienna Development Method: The Meta-Language*, [16], LNCS, pages 337–374. Springer-Verlag, 1978.
- [11] D. Bjørner. The Vienna Development Method: Software Abstraction and Program Synthesis. In *Mathematical Studies of Information Processing*, volume 75 of LNCS. Springer-Verlag, 1979. Proceedings of Conference at Research Institute for Mathematical Sciences (RIMS), University of Kyoto, August 1978.
- [12] D. Bjørner. Stepwise Transformation of Software Architectures. In [17], chapter 11, pages 353–378. Prentice-Hall, 1982.
- [13] D. Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science, the EATCS Series. Springer, 2006. Chapters 12–14 are primarily authored by Christian Krog Madsen. See [14, 15].
- [14] D. Bjørner. *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press, 2008.
- [15] D. Bjørner. **Chinese:** *Software Engineering, Vol. 2: Specification of Systems and Languages*. Qinghua University Press. Translated by Dr Liu Bo Chao et al., 2010.

- [16] D. Bjørner and C. B. Jones, editors. *The Vienna Development Method: The Meta-Language*, volume 61 of *LNCS*. Springer, 1978. This was the first monograph on *Meta-IV*. [9, 10, 11].
- [17] D. Bjørner and C. B. Jones, editors. *Formal Specification and Software Development*. Prentice-Hall, 1982.
- [18] D. Bjørner, C. B. Jones, M. M. an Airchinnigh, and E. J. Neuhold, editors. *VDM – A Formal Method at Work*. Proc. VDM-Europe Symposium 1987, Brussels, Belgium, Springer, Lecture Notes in Computer Science, Vol. 252, March 1987.
- [19] S. Craß. A Formal Model of the Extensible Virtual Shared Memory (XVSM) and its Implementation in Haskell – Design and Specification. M.sc., Technische Universität Wien, A-1040 Wien, Karlsplatz 13, Austria, February 5 2010.
- [20] S. Craß, E. Kühn, and G. Salzer. Algebraic Foundation of a Data Model for an Extensible Space-based Collaboration Protocol. In B. C. Desai, editor, *IDEAS 2009*, pages 301–306, Cetraro, Calabria, Italy, September 16–18 2009.
- [21] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
- [22] J. Fitzgerald and P. G. Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development*. Cambridge University Press, Cambridge, UK, Second edition, 2009.
- [23] C. W. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1992.
- [24] C. W. George, A. E. Haxthausen, S. Hughes, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method*. The BCS Practitioner Series. Prentice-Hall, Hemel Hempstead, England, 1995.
- [25] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7. Cambridge Univ. Press, Cambridge, UK, Cambridge Tracts in Theoretical Computer Science edition, 1989.
- [26] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [27] C. Hoare. *Communicating Sequential Processes*. C.A.R. Hoare Series in Computer Science. Prentice-Hall International, 1985. Published electronically: <http://www.usingcsp.com/-cspbook.pdf> (2004).
- [28] ITU-T. CCITT Recommendation Z.120: Message Sequence Chart (MSC), 1992, 1996, 1999.
- [29] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, Mass., USA, April 2006. ISBN 0-262-10114-9.
- [30] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice-Hall, 1980.
- [31] C. B. Jones. *Systematic Software Development — Using VDM*. Prentice-Hall, 1986.
- [32] C. B. Jones. *Systematic Software Development — Using VDM, 2nd Edition*. Prentice-Hall, 1989.
- [33] E. Kühn, R. Mordinyi, L. Keszthelyi, and C. Schreiber. Introducing the Concept of Customizable Structured Space for Agent Coordination in the Production of Automation Domain. In S. Decker, Sichman and Castelfranchi, editors, *8<sup>th</sup> Intl. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS 2009)*, volume 625–632 of *Proceedings of Autonomous Agents and Multi-Agent Systems*, Budapest, Hungary, May 10–15 2009. 8.

- [34] E. Kühn, R. Mordinyi, L. Keszthelyi, C. Schreiber, S. Bessler, and S. Tomic. Aspect-oriented Space Containers for Efficient Publish/Subscribe Scenarios in Intelligent Transportation Systems. In T. D. and P. H. Meersmann, editors, *OTM 2009, Part I*, volume 5870 of *LNCS*, pages 432–448. Springer, 2009.
- [35] L. Lamport. *Specifying Systems*. Addison–Wesley, Boston, Mass., USA, 2002.
- [36] J. Lee and W. Delmore. The Vienna Definition Language, a generalization of instruction definitions. In *SIGPLAN Symp. on Programming Language Definitions, San Francisco*, Aug. 1969.
- [37] P. Lucas. Formal Definition of Programming Languages and Systems. In *Proc. IFIP’71*. IFIP World Congress Proceedings, Springer, 1971.
- [38] P. Lucas. On the Semantics of Programming Languages and Software Devices. In Rustin, editor, *Formal Semantics of Programming Languages*. Prentice-Hall, 1972.
- [39] P. Lucas. On the formalization of programming languages: Early history and main approaches. In D. Bjørner and C. B. Jones, editors, [16]. Springer, 1978.
- [40] P. Lucas. Formal Semantics of Programming Languages: VDL. *IBM Journal of Devt. and Res.*, 25(5):549–561, 1981.
- [41] P. Lucas. Main approaches to formal specification. In [12], chapter 1, pages 3–24. Prentice-Hall, 1982.
- [42] P. Lucas. Origins, hopes, and achievements. In [18], pages 1–18. Springer, 1987.
- [43] P. Lucas and K. Walk. On the Formal Description of PL/I. *Annual Review Automatic Programming Part 3*, 6(3), 1969.
- [44] B. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [45] W. Reisig. *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Leitfäden der Informatik. Vieweg+Teubner, 1st edition, 15 June 2010. 248 pages; ISBN 978-3-8348-1290-2.
- [46] D. A. Schmidt. *The Structure of Typed Programming Languages*. MIT Press, 1994. ISBN 0262193493.
- [47] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.
- [48] C. C. Zhou and M. R. Hansen. *Duration Calculus: A Formal Approach to Real-time Systems*. Monographs in Theoretical Computer Science. An EATCS Series. Springer–Verlag, 2004.

# Appendix A

## RSL: The Raise Specification Language — A Primer

### A.1 Type Expressions

Type expressions are expressions whose value are type, that is, possibly infinite sets of values (of “that” type).

#### A.1.1 Atomic Types

Atomic types have (atomic) values. That is, values which we consider to have no proper constituent (sub-)values, i.e., cannot, to us, be meaningfully “taken apart”.

RSL has a number of *built-in* atomic types. There are the Booleans, integers, natural numbers, reals, characters, and texts.

**Basic Types:**

type

|     |             |                                                                    |
|-----|-------------|--------------------------------------------------------------------|
| [1] | <b>Bool</b> | true, false                                                        |
| [2] | <b>Int</b>  | ... , -2, -2, 0, 1, 2, ...                                         |
| [3] | <b>Nat</b>  | 0, 1, 2, ...                                                       |
| [4] | <b>Real</b> | ..., -5.43, -1.0, 0.0, 1.23... , 2,7182... , 3,1415... , 4.56, ... |
| [5] | <b>Char</b> | "a", "b", ..., "0", ...                                            |
| [6] | <b>Text</b> | "abracadabra"                                                      |

Composite Types

Composite types have composite values. That is, values which we consider to have proper constituent (sub-)values, i.e., can be meaningfully “taken apart”. There are two ways of expressing composite types: either explicitly, using concrete type expressions, or implicitly, using sorts (i.e., abstract types) and observer functions.

**Concrete Composite Types** From these one can form type expressions: finite sets, infinite sets, Cartesian products, lists, maps, etc.

Let A, B and C be any type names or type expressions, then:

**Composite Type Expressions:**

|     |          |
|-----|----------|
| [7] | A-set    |
| [8] | A-infset |

- [9]  $A \times B \times \dots \times C$
- [10]  $A^*$
- [11]  $A^\omega$
- [12]  $A \xrightarrow{\vec{m}} B$
- [13]  $A \rightarrow B$
- [14]  $A \xrightarrow{\sim} B$
- [15]  $(A)$
- [16]  $A \mid B \mid \dots \mid C$
- [17]  $\text{mk\_id}(\text{sel\_a}:A, \dots, \text{sel\_b}:B)$
- [18]  $\text{sel\_a}:A \dots \text{sel\_b}:B$

The following are generic type expressions:

1. The Boolean type of truth values **false** and **true**.
2. The integer type on integers  $\dots, -2, -1, 0, 1, 2, \dots$ .
3. The natural number type of positive integer values  $0, 1, 2, \dots$ .
4. The real number type of real values, i.e., values whose numerals can be written as an integer, followed by a period ("."), followed by a natural number (the fraction).
5. The character type of character values "a", "b", ...
6. The text type of character string values "aa", "aaa", ..., "abc", ...
7. The set type of finite cardinality set values.
8. The set type of infinite and finite cardinality set values.
9. The Cartesian type of Cartesian values.
10. The list type of finite length list values.
11. The list type of infinite and finite length list values.
12. The map type of finite definition set map values.
13. The function type of total function values.
14. The function type of partial function values.
15. In  $(A)$   $A$  is constrained to be:
  - either a Cartesian  $B \times C \times \dots \times D$ , in which case it is identical to type expression kind 9,
  - or not to be the name of a built-in type (cf., 1–6) or of a type, in which case the parentheses serve as simple delimiters, e.g.,  $(A \xrightarrow{\vec{m}} B)$ , or  $(A^*)\text{-set}$ , or  $(A\text{-set})\text{list}$ , or  $(A \mid B) \xrightarrow{\vec{m}}$   $(C \mid D \mid (E \xrightarrow{\vec{m}} F))$ , etc.
16. The postulated disjoint union of types  $A, B, \dots$ , and  $C$ .
17. The record type of  $\text{mk\_id}$ -named record values  $\text{mk\_id}(av, \dots, bv)$ , where  $av, \dots, bv$ , are values of respective types. The distinct identifiers  $\text{sel\_a}$ , etc., designate selector functions.
18. The record type of unnamed record values  $(av, \dots, bv)$ , where  $av, \dots, bv$ , are values of respective types. The distinct identifiers  $\text{sel\_a}$ , etc., designate selector functions.

Sorts and Observer Functions

**type**

A, B, C, ..., D

**value**

obs\_B: A → B, obs\_C: A → C, ..., obs\_D: A → D

The above expresses that values of type A are composed from at least three values — and these are of type B, C, ..., and D. A concrete type definition corresponding to the above presupposing material of the next section

**type**

B, C, ..., D

A = B × C × ... × D

Type Definitions

**A.1.2 Concrete Types**

Types can be concrete in which case the structure of the type is specified by type expressions:

**Type Definition:****type**

A = Type\_expr

Some schematic type definitions are:

**Variety of Type Definitions:**

- [1] Type\_name = Type\_expr /\* without |s or subtypes \*/
- [2] Type\_name = Type\_expr\_1 | Type\_expr\_2 | ... | Type\_expr\_n
- [3] Type\_name ==  
         mk\_id\_1(s\_a1:Type\_name\_a1,...,s\_ai:Type\_name\_ai) |  
         ... |  
         mk\_id\_n(s\_z1:Type\_name\_z1,...,s\_zk:Type\_name\_zk)
- [4] Type\_name :: sel\_a:Type\_name\_a ... sel\_z:Type\_name\_z
- [5] Type\_name = { | v:Type\_name' • P(v) | }

where a form of [2–3] is provided by combining the types:

**Record Types:**

```
Type_name = A | B | ... | Z
A == mk_id_1(s_a1:A_1,...,s_ai:A_i)
B == mk_id_2(s_b1:B_1,...,s_bj:B_j)
...
Z == mk_id_n(s_z1:Z_1,...,s_zk:Z_k)
```

Types A, B, ..., Z are disjoint, i.e., shares no values, provided all mk\_id\_k are distinct and due to the use of the disjoint record type constructor ==.

**axiom**

```
∀ a1:A_1, a2:A_2, ..., ai:Ai •
 s_a1(mk_id_1(a1,a2,...,ai))=a1 ∧ s_a2(mk_id_1(a1,a2,...,ai))=a2 ∧
 ... ∧ s_ai(mk_id_1(a1,a2,...,ai))=ai ∧
 ∀ a:A • let mk_id_1(a1',a2',...,ai') = a in
 a1' = s_a1(a) ∧ a2' = s_a2(a) ∧ ... ∧ ai' = s_ai(a) end
```

**Subtypes**

In RSL, each type represents a set of values. Such a set can be delimited by means of predicates. The set of values  $b$  which have type  $B$  and which satisfy the predicate  $\mathcal{P}$ , constitute the subtype  $A$ :

**Subtypes:****type**

$$A = \{ | b:B \cdot \mathcal{P}(b) | \}$$

**Sorts — Abstract Types**

Types can be (abstract) sorts in which case their structure is not specified:

**Sorts:****type**

$A, B, \dots, C$

The RSL Predicate Calculus

## A.2 Propositional Expressions

Let identifiers (or propositional expressions)  $a, b, \dots, c$  designate Boolean values (**true** or **false** [or **chaos**]). Then:

**Propositional Expressions:**

**false, true**

$a, b, \dots, c \sim a, a \wedge b, a \vee b, a \Rightarrow b, a = b, a \neq b$

are propositional expressions having Boolean values.  $\sim, \wedge, \vee, \Rightarrow, =$  and  $\neq$  are Boolean connectives (i.e., operators). They can be read as: *not, and, or, if then (or implies), equal and not equal*.

**Simple Predicate Expressions**

Let identifiers (or propositional expressions)  $a, b, \dots, c$  designate Boolean values, let  $x, y, \dots, z$  (or term expressions) designate non-Boolean values and let  $i, j, \dots, k$  designate number values, then:

**Simple Predicate Expressions:**

**false, true**

$a, b, \dots, c$

$\sim a, a \wedge b, a \vee b, a \Rightarrow b, a = b, a \neq b$

$x = y, x \neq y,$

$i < j, i \leq j, i \geq j, i \neq j, i \geq j, i > j$

are simple predicate expressions.

**Quantified Expressions**

Let  $X, Y, \dots, C$  be type names or type expressions, and let  $\mathcal{P}(x), \mathcal{Q}(y)$  and  $\mathcal{R}(z)$  designate predicate expressions in which  $x, y$  and  $z$  are free. Then:

**Quantified Expressions:**

$\forall x:X \cdot \mathcal{P}(x)$

$\exists y:Y \cdot \mathcal{Q}(y)$

$\exists ! z:Z \cdot \mathcal{R}(z)$

are quantified expressions — also being predicate expressions.

They are “read” as: For all  $x$  (values in type  $X$ ) the predicate  $\mathcal{P}(x)$  holds; there exists (at least) one  $y$  (value in type  $Y$ ) such that the predicate  $\mathcal{Q}(y)$  holds; and there exists a unique  $z$  (value in type  $Z$ ) such that the predicate  $\mathcal{R}(z)$  holds.

Concrete RSL Types: Values and Operations

## A.3 Arithmetic

**Arithmetic:**

**type**

$\mathbf{Nat}, \mathbf{Int}, \mathbf{Real}$

**value**

$+, -, *: \mathbf{Nat} \times \mathbf{Nat} \rightarrow \mathbf{Nat} \mid \mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int} \mid \mathbf{Real} \times \mathbf{Real} \rightarrow \mathbf{Real}$

$/: \mathbf{Nat} \times \mathbf{Nat} \rightsquigarrow \mathbf{Nat} \mid \mathbf{Int} \times \mathbf{Int} \rightsquigarrow \mathbf{Int} \mid \mathbf{Real} \times \mathbf{Real} \rightsquigarrow \mathbf{Real}$

$<, \leq, =, \neq, \geq, > (\mathbf{Nat} \mid \mathbf{Int} \mid \mathbf{Real}) \rightarrow (\mathbf{Nat} \mid \mathbf{Int} \mid \mathbf{Real})$

Set Expressions

### A.3.1 Set Enumerations

Let the below  $a$ 's denote values of type  $A$ , then the below designate simple set enumerations:

**Set Enumerations:**

$\{\{\}, \{a\}, \{e_1, e_2, \dots, e_n\}, \dots\} \in \mathbf{A-set}$

$\{\{\}, \{a\}, \{e_1, e_2, \dots, e_n\}, \dots, \{e_1, e_2, \dots\}\} \in \mathbf{A-infset}$

Set Comprehension

The expression, last line below, to the right of the  $\equiv$ , expresses set comprehension. The expression “builds” the set of values satisfying the given predicate. It is abstract in the sense that it does not do so by following a concrete algorithm.

**Set Comprehension:**

**type**

$A, B$

$P = A \rightarrow \mathbf{Bool}$

$Q = A \rightsquigarrow B$

**value**

$\text{comprehend}: \mathbf{A-infset} \times P \times Q \rightarrow \mathbf{B-infset}$

$\text{comprehend}(s, P, Q) \equiv \{ Q(a) \mid a: A \bullet a \in s \wedge P(a) \}$

Cartesian Expressions

### A.3.2 Cartesian Enumerations

Let  $e$  range over values of Cartesian types involving  $A, B, \dots, C$ , then the below expressions are simple Cartesian enumerations:

**Cartesian Enumerations:****type**

A, B, ..., C  
 $A \times B \times \dots \times C$

**value**

$(e_1, e_2, \dots, e_n)$

List Expressions

**A.3.3 List Enumerations**

Let  $a$  range over values of type  $A$ , then the below expressions are simple list enumerations:

**List Enumerations:**

$\{\langle \rangle, \langle e \rangle, \dots, \langle e_1, e_2, \dots, e_n \rangle, \dots\} \in A^*$   
 $\{\langle \rangle, \langle e \rangle, \dots, \langle e_1, e_2, \dots, e_n \rangle, \dots, \langle e_1, e_2, \dots, e_n, \dots \rangle, \dots\} \in A^\omega$

$\langle a_i \dots a_j \rangle$

The last line above assumes  $a_i$  and  $a_j$  to be integer-valued expressions. It then expresses the set of integers from the value of  $e_i$  to and including the value of  $e_j$ . If the latter is smaller than the former, then the list is empty.

List Comprehension

The last line below expresses list comprehension.

**List Comprehension:****type**

A, B,  $P = A \rightarrow \text{Bool}$ ,  $Q = A \rightsquigarrow B$

**value**

comprehend:  $A^\omega \times P \times Q \rightsquigarrow B^\omega$   
 comprehend( $l, P, Q$ )  $\equiv$   
 $\langle Q(l(i)) \mid i \text{ in } \langle 1..len\ l \rangle \bullet P(l(i)) \rangle$

Map Expressions

**A.3.4 Map Enumerations**

Let (possibly indexed)  $u$  and  $v$  range over values of type  $T1$  and  $T2$ , respectively, then the below expressions are simple map enumerations:

**Map Enumerations:****type**

$T1, T2$   
 $M = T1 \xrightarrow{m} T2$

**value**

$u, u_1, u_2, \dots, u_n: T1, v, v_1, v_2, \dots, v_n: T2$   
 $[], [u \mapsto v], \dots, [u_1 \mapsto v_1, u_2 \mapsto v_2, \dots, u_n \mapsto v_n] \forall \in M$

Map Comprehension

The last line below expresses map comprehension:

**Map Comprehension:****type**

$U, V, X, Y$   
 $M = U \xrightarrow{m} V$   
 $F = U \xrightarrow{\sim} X$   
 $G = V \xrightarrow{\sim} Y$   
 $P = U \rightarrow \mathbf{Bool}$

**value**

$\text{comprehend}: M \times F \times G \times P \rightarrow (X \xrightarrow{m} Y)$   
 $\text{comprehend}(m, F, G, P) \equiv$   
 $[ F(u) \mapsto G(m(u)) \mid u:U \bullet u \in \text{dom } m \wedge P(u) ]$

Set Operations

**A.3.5 Set Operator Signatures****Set Operations:****value**

19  $\in: A \times \mathbf{A}\text{-infset} \rightarrow \mathbf{Bool}$   
20  $\notin: A \times \mathbf{A}\text{-infset} \rightarrow \mathbf{Bool}$   
21  $U: \mathbf{A}\text{-infset} \times \mathbf{A}\text{-infset} \rightarrow \mathbf{A}\text{-infset}$   
22  $U: (\mathbf{A}\text{-infset})\text{-infset} \rightarrow \mathbf{A}\text{-infset}$   
23  $\cap: \mathbf{A}\text{-infset} \times \mathbf{A}\text{-infset} \rightarrow \mathbf{A}\text{-infset}$   
24  $\cap: (\mathbf{A}\text{-infset})\text{-infset} \rightarrow \mathbf{A}\text{-infset}$   
25  $\setminus: \mathbf{A}\text{-infset} \times \mathbf{A}\text{-infset} \rightarrow \mathbf{A}\text{-infset}$   
26  $\subset: \mathbf{A}\text{-infset} \times \mathbf{A}\text{-infset} \rightarrow \mathbf{Bool}$   
27  $\subseteq: \mathbf{A}\text{-infset} \times \mathbf{A}\text{-infset} \rightarrow \mathbf{Bool}$   
28  $=: \mathbf{A}\text{-infset} \times \mathbf{A}\text{-infset} \rightarrow \mathbf{Bool}$   
29  $\neq: \mathbf{A}\text{-infset} \times \mathbf{A}\text{-infset} \rightarrow \mathbf{Bool}$   
30  $\text{card}: \mathbf{A}\text{-infset} \xrightarrow{\sim} \mathbf{Nat}$

Set Examples

**Set Examples:****examples**

$a \in \{a, b, c\}$   
 $a \notin \{\}, a \notin \{b, c\}$   
 $\{a, b, c\} \cup \{a, b, d, e\} = \{a, b, c, d, e\}$   
 $U\{\{a\}, \{a, b\}, \{a, d\}\} = \{a, b, d\}$   
 $\{a, b, c\} \cap \{c, d, e\} = \{c\}$   
 $\cap\{\{a\}, \{a, b\}, \{a, d\}\} = \{a\}$   
 $\{a, b, c\} \setminus \{c, d\} = \{a, b\}$   
 $\{a, b\} \subset \{a, b, c\}$   
 $\{a, b, c\} \subseteq \{a, b, c\}$   
 $\{a, b, c\} = \{a, b, c\}$   
 $\{a, b, c\} \neq \{a, b\}$   
 $\text{card } \{\} = 0, \text{card } \{a, b, c\} = 3$

Informal Explication

19.  $\in$ : The membership operator expresses that an element is a member of a set.

20.  $\notin$ : The nonmembership operator expresses that an element is not a member of a set.

21.  $\cup$ : The infix union operator. When applied to two sets, the operator gives the set whose members are in either or both of the two operand sets.
22.  $\cup$ : The distributed prefix union operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
23.  $\cap$ : The infix intersection operator. When applied to two sets, the operator gives the set whose members are in both of the two operand sets.
24.  $\cap$ : The prefix distributed intersection operator. When applied to a set of sets, the operator gives the set whose members are in some of the operand sets.
25.  $\setminus$ : The set complement (or set subtraction) operator. When applied to two sets, the operator gives the set whose members are those of the left operand set which are not in the right operand set.
26.  $\subseteq$ : The proper subset operator expresses that all members of the left operand set are also in the right operand set.
27.  $\subset$ : The proper subset operator expresses that all members of the left operand set are also in the right operand set, and that the two sets are not identical.
28.  $=$ : The equal operator expresses that the two operand sets are identical.
29.  $\neq$ : The nonequal operator expresses that the two operand sets are *not* identical.
30. **card**: The cardinality operator gives the number of elements in a finite set.

#### Set Operator Definitions

The operations can be defined as follows ( $\equiv$  is the definition symbol):

#### Set Operation Definitions:

value

$$s' \cup s'' \equiv \{ a \mid a:A \bullet a \in s' \vee a \in s'' \}$$

$$s' \cap s'' \equiv \{ a \mid a:A \bullet a \in s' \wedge a \in s'' \}$$

$$s' \setminus s'' \equiv \{ a \mid a:A \bullet a \in s' \wedge a \notin s'' \}$$

$$s' \subseteq s'' \equiv \forall a:A \bullet a \in s' \Rightarrow a \in s''$$

$$s' \subset s'' \equiv s' \subseteq s'' \wedge \exists a:A \bullet a \in s'' \wedge a \notin s'$$

$$s' = s'' \equiv \forall a:A \bullet a \in s' \equiv a \in s'' \equiv s' \subseteq s'' \wedge s'' \subseteq s'$$

$$s' \neq s'' \equiv s' \cap s'' \neq \{\}$$

**card**  $s \equiv$   
 if  $s = \{\}$  then 0 else  
 let  $a:A \bullet a \in s$  in  $1 + \text{card}(s \setminus \{a\})$  end end  
 pre  $s$  /\* is a finite set \*/  
**card**  $s \equiv \text{chaos}$  /\* tests for infinity of  $s$  \*/

#### Cartesian Operations

#### Cartesian Operations:

type

A, B, C  
 $g0: G0 = A \times B \times C$   
 $g1: G1 = (A \times B \times C)$   
 $g2: G2 = (A \times B) \times C$   
 $g3: G3 = A \times (B \times C)$

value

$va:A, vb:B, vc:C, vd:D$   
 $(va,vb,vc):G0,$   
 $(va,vb,vc):G1$   
 $((va,vb),vc):G2$   
 $(va3,(vb3,vc3)):G3$

**decomposition expressions**

$$\text{let } (a1,b1,c1) = g0,$$

$$\begin{aligned} & (a1',b1',c1') = g1 \text{ in .. end} \\ \text{let } ((a2,b2),c2) = g2 \text{ in .. end} \\ \text{let } (a3,(b3,c3)) = g3 \text{ in .. end} \end{aligned}$$

List Operations

**A.3.6 List Operator Signatures****List Operations:**

value

$$\text{hd}: A^\omega \rightsquigarrow A$$

$$\text{tl}: A^\omega \rightsquigarrow A^\omega$$

$$\text{len}: A^\omega \rightsquigarrow \text{Nat}$$

$$\text{inds}: A^\omega \rightarrow \text{Nat-infset}$$

$$\text{elems}: A^\omega \rightarrow A\text{-infset}$$

$$\text{.}(\cdot): A^\omega \times \text{Nat} \rightsquigarrow A$$

$$\hat{\cdot}: A^* \times A^\omega \rightarrow A^\omega$$

$$=: A^\omega \times A^\omega \rightarrow \text{Bool}$$

$$\neq: A^\omega \times A^\omega \rightarrow \text{Bool}$$

List Operation Examples

**List Examples:**

examples

$$\text{hd}\langle a1,a2,\dots,a_m \rangle = a1$$

$$\text{tl}\langle a1,a2,\dots,a_m \rangle = \langle a2,\dots,a_m \rangle$$

$$\text{len}\langle a1,a2,\dots,a_m \rangle = m$$

$$\text{inds}\langle a1,a2,\dots,a_m \rangle = \{1,2,\dots,m\}$$

$$\text{elems}\langle a1,a2,\dots,a_m \rangle = \{a1,a2,\dots,a_m\}$$

$$\langle a1,a2,\dots,a_m \rangle(i) = a_i$$

$$\langle a,b,c \rangle \hat{\cdot} \langle a,b,d \rangle = \langle a,b,c,a,b,d \rangle$$

$$\langle a,b,c \rangle = \langle a,b,c \rangle$$

$$\langle a,b,c \rangle \neq \langle a,b,d \rangle$$

Informal Explication

- **hd**: Head gives the first element in a nonempty list.
- **tl**: Tail gives the remaining list of a nonempty list when Head is removed.
- **len**: Length gives the number of elements in a finite list.
- **inds**: Indices give the set of indices from 1 to the length of a nonempty list. For empty lists, this set is the empty set as well.
- **elems**: Elements gives the possibly infinite set of all distinct elements in a list.
- $\ell(i)$ : Indexing with a natural number,  $i$  larger than 0, into a list  $\ell$  having a number of elements larger than or equal to  $i$ , gives the  $i$ th element of the list.
- $\hat{\cdot}$ : Concatenates two operand lists into one. The elements of the left operand list are followed by the elements of the right. The order with respect to each list is maintained.
- $=$ : The equal operator expresses that the two operand lists are identical.

- $\neq$ : The nonequal operator expresses that the two operand lists are *not* identical.

The operations can also be defined as follows:

List Operator Definitions

value

$\text{is\_finite\_list}: A^\omega \rightarrow \mathbf{Bool}$

$\text{len } q \equiv$   
 case  $\text{is\_finite\_list}(q)$  of  
 true  $\rightarrow$  if  $q = \langle \rangle$  then 0 else  $1 + \text{len } \text{tl } q$  end,  
 false  $\rightarrow$  chaos end

$\text{inds } q \equiv$   
 case  $\text{is\_finite\_list}(q)$  of  
 true  $\rightarrow \{ i \mid i:\mathbf{Nat} \bullet 1 \leq i \leq \text{len } q \}$ ,  
 false  $\rightarrow \{ i \mid i:\mathbf{Nat} \bullet i \neq 0 \}$  end

$\text{elems } q \equiv \{ q(i) \mid i:\mathbf{Nat} \bullet i \in \text{inds } q \}$

$q(i) \equiv$   
 if  $i=1$   
 then  
 if  $q \neq \langle \rangle$   
 then let  $a:A, q':Q \bullet q = \langle a \rangle \wedge q'$  in  $a$  end  
 else chaos end  
 else  $q(i-1)$  end

$\text{fq} \wedge \text{iq} \equiv$   
 $\langle$  if  $1 \leq i \leq \text{len } \text{fq}$  then  $\text{fq}(i)$  else  $\text{iq}(i - \text{len } \text{fq})$  end  
 $\mid i:\mathbf{Nat} \bullet$  if  $\text{len } \text{iq} \neq \text{chaos}$  then  $i \leq \text{len } \text{fq} + \text{len } \text{iq}$  end  
 $\text{pre } \text{is\_finite\_list}(\text{fq})$

$\text{iq}' = \text{iq}'' \equiv$   
 $\text{inds } \text{iq}' = \text{inds } \text{iq}'' \wedge \forall i:\mathbf{Nat} \bullet i \in \text{inds } \text{iq}' \Rightarrow \text{iq}'(i) = \text{iq}''(i)$

$\text{iq}' \neq \text{iq}'' \equiv \sim(\text{iq}' = \text{iq}'')$

Map Operations

### A.3.7 Map Operator Signatures and Map Operation Examples

value

$\text{m}(a): M \rightarrow A \xrightarrow{\sim} B, \text{m}(a) = b$

$\text{dom}: M \rightarrow \mathbf{A-infset}$  [domain of map]  
 $\text{dom } [a_1 \mapsto b_1, a_2 \mapsto b_2, \dots, a_n \mapsto b_n] = \{a_1, a_2, \dots, a_n\}$

$\text{rng}: M \rightarrow \mathbf{B-infset}$  [range of map]  
 $\text{rng } [a_1 \mapsto b_1, a_2 \mapsto b_2, \dots, a_n \mapsto b_n] = \{b_1, b_2, \dots, b_n\}$

$\dagger: M \times M \rightarrow M$  [override extension]  
 $[a \mapsto b, a' \mapsto b', a'' \mapsto b''] \dagger [a' \mapsto b'', a'' \mapsto b'] = [a \mapsto b, a' \mapsto b'', a'' \mapsto b']$

$$\cup: M \times M \rightarrow M \text{ [merge } \cup \text{]} \\ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] \cup [a''' \mapsto b'''] = [a \mapsto b, a' \mapsto b', a'' \mapsto b'', a''' \mapsto b''']$$

$$\setminus: M \times \mathbf{A\text{-infset}} \rightarrow M \text{ [restriction by]} \\ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] \setminus \{a\} = [a' \mapsto b', a'' \mapsto b'']$$

$$/: M \times \mathbf{A\text{-infset}} \rightarrow M \text{ [restriction to]} \\ [a \mapsto b, a' \mapsto b', a'' \mapsto b''] / \{a', a''\} = [a' \mapsto b', a'' \mapsto b'']$$

$$=, \neq: M \times M \rightarrow \mathbf{Bool}$$

$$\circ: (A \xrightarrow{m} B) \times (B \xrightarrow{m} C) \rightarrow (A \xrightarrow{m} C) \text{ [composition]} \\ [a \mapsto b, a' \mapsto b'] \circ [b \mapsto c, b' \mapsto c'] = [a \mapsto c, a' \mapsto c']$$

### Map Operation Explication

- $m(a)$ : Application gives the element that  $a$  maps to in the map  $m$ .
- **dom**: Domain/Definition Set gives the set of values which *maps to* in a map.
- **rng**: Range/Image Set gives the set of values which *are mapped to* in a map.
- $\dagger$ : Override/Extend. When applied to two operand maps, it gives the map which is like an override of the left operand map by all or some “pairings” of the right operand map.
- $\cup$ : Merge. When applied to two operand maps, it gives a merge of these maps.
- $\setminus$ : Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements that are not in the right operand set.
- $/$ : Restriction. When applied to two operand maps, it gives the map which is a restriction of the left operand map to the elements of the right operand set.
- $=$ : The equal operator expresses that the two operand maps are identical.
- $\neq$ : The nonequal operator expresses that the two operand maps are *not* identical.
- $\circ$ : Composition. When applied to two operand maps, it gives the map from definition set elements of the left operand map,  $m_1$ , to the range elements of the right operand map,  $m_2$ , such that if  $a$  is in the definition set of  $m_1$  and maps into  $b$ , and if  $b$  is in the definition set of  $m_2$  and maps into  $c$ , then  $a$ , in the composition, maps into  $c$ .

### Map Operation Redefinitions

The map operations can also be defined as follows:

### Map Operation Redefinitions:

value

$$\text{rng } m \equiv \{ m(a) \mid a:A \bullet a \in \text{dom } m \}$$

$$m1 \dagger m2 \equiv \\ [ a \mapsto b \mid a:A, b:B \bullet \\ a \in \text{dom } m1 \setminus \text{dom } m2 \wedge b=m1(a) \vee a \in \text{dom } m2 \wedge b=m2(a) ]$$

$$m1 \cup m2 \equiv [ a \mapsto b \mid a:A, b:B \bullet \\ a \in \text{dom } m1 \wedge b=m1(a) \vee a \in \text{dom } m2 \wedge b=m2(a) ]$$

$$m \setminus s \equiv [ a \mapsto m(a) \mid a:A \bullet a \in \mathbf{dom} m \setminus s ]$$

$$m / s \equiv [ a \mapsto m(a) \mid a:A \bullet a \in \mathbf{dom} m \cap s ]$$

$$m1 = m2 \equiv$$

$$\mathbf{dom} m1 = \mathbf{dom} m2 \wedge \forall a:A \bullet a \in \mathbf{dom} m1 \Rightarrow m1(a) = m2(a)$$

$$m1 \neq m2 \equiv \sim(m1 = m2)$$

$$m^\circ n \equiv$$

$$[ a \mapsto c \mid a:A, c:C \bullet a \in \mathbf{dom} m \wedge c = n(m(a)) ]$$

$$\mathbf{pre\ rng} m \subseteq \mathbf{dom} n$$

$\lambda$ -Calculus + Functions

## A.4 The $\lambda$ -Calculus Syntax

$\lambda$ -Calculus Syntax:

**type** /\* A BNF Syntax: \*/

$$\langle L \rangle ::= \langle V \rangle \mid \langle F \rangle \mid \langle A \rangle \mid ( \langle A \rangle )$$

$$\langle V \rangle ::= /* \text{variables, i.e. identifiers} */$$

$$\langle F \rangle ::= \lambda \langle V \rangle \bullet \langle L \rangle$$

$$\langle A \rangle ::= ( \langle L \rangle \langle L \rangle )$$

**value** /\* Examples \*/

$$\langle L \rangle: e, f, a, \dots$$

$$\langle V \rangle: x, \dots$$

$$\langle F \rangle: \lambda x \bullet e, \dots$$

$$\langle A \rangle: f a, (f a), f(a), (f)(a), \dots$$

Free and Bound Variables

**Free and Bound Variables:** Let  $x, y$  be variable names and  $e, f$  be  $\lambda$ -expressions.

- $\langle V \rangle$ : Variable  $x$  is free in  $x$ .
- $\langle F \rangle$ :  $x$  is free in  $\lambda y \bullet e$  if  $x \neq y$  and  $x$  is free in  $e$ .
- $\langle A \rangle$ :  $x$  is free in  $f(e)$  if it is free in either  $f$  or  $e$  (i.e., also in both).

Substitution In RSL, the following rules for substitution apply:

**Substitution:**

- $\mathbf{subst}([N/x]x) \equiv N$ ;
- $\mathbf{subst}([N/x]a) \equiv a$ ,
- for all variables  $a \neq x$ ;
- $\mathbf{subst}([N/x](P Q)) \equiv (\mathbf{subst}([N/x]P) \mathbf{subst}([N/x]Q))$ ;
- $\mathbf{subst}([N/x](\lambda x \bullet P)) \equiv \lambda y \bullet P$ ;
- $\mathbf{subst}([N/x](\lambda y \bullet P)) \equiv \lambda y \bullet \mathbf{subst}([N/x]P)$ ,
- if  $x \neq y$  and  $y$  is not free in  $N$  or  $x$  is not free in  $P$ ;
- $\mathbf{subst}([N/x](\lambda y \bullet P)) \equiv \lambda z \bullet \mathbf{subst}([N/z] \mathbf{subst}([z/y]P))$ ,
- if  $y \neq x$  and  $y$  is free in  $N$  and  $x$  is free in  $P$   
(where  $z$  is not free in  $(N P)$ ).

$\alpha$ -Renaming and  $\beta$ -Reduction

**$\alpha$  and  $\beta$  Conversions:**

- $\alpha$ -renaming:  $\lambda x \bullet M$

If  $x, y$  are distinct variables then replacing  $x$  by  $y$  in  $\lambda x \bullet M$  results in  $\lambda y \bullet \text{subst}([y/x]M)$ . We can rename the formal parameter of a  $\lambda$ -function expression provided that no free variables of its body  $M$  thereby become bound.

- $\beta$ -reduction:  $(\lambda x \bullet M)(N)$

All free occurrences of  $x$  in  $M$  are replaced by the expression  $N$  provided that no free variables of  $N$  thereby become bound in the result.  $(\lambda x \bullet M)(N) \equiv \text{subst}([N/x]M)$

**Function Signatures**

For sorts we may want to postulate some functions:

**Sorts and Function Signatures:****type**

A, B, C

**value**

obs\_B:  $A \rightarrow B$ ,

obs\_C:  $A \rightarrow C$ ,

gen\_A:  $B \times C \rightarrow A$

**Function Definitions**

Functions can be defined explicitly:

**Explicit Function Definitions:****value**

f: Arguments  $\rightarrow$  Result

f(args)  $\equiv$  DValueExpr

g: Arguments  $\overset{\sim}{\rightarrow}$  Result

g(args)  $\equiv$  ValueAndStateChangeClause

pre P(args)

Or functions can be defined implicitly:

**Implicit Function Definitions:****value**

f: Arguments  $\rightarrow$  Result

f(args) as result

post P1(args,result)

g: Arguments  $\overset{\sim}{\rightarrow}$  Result

g(args) as result

pre P2(args)

post P3(args,result)

The symbol  $\overset{\sim}{\rightarrow}$  indicates that the function is partial and thus not defined for all arguments. Partial functions should be assisted by preconditions stating the criteria for arguments to be meaningful to the function.

**Other Applicative Expressions**

## A.5 Simple let Expressions

Simple (i.e., nonrecursive) **let** expressions:

### Let Expressions:

**let**  $a = \mathcal{E}_d$  **in**  $\mathcal{E}_b(a)$  **end**

is an “expanded” form of:

$(\lambda a. \mathcal{E}_b(a))(\mathcal{E}_d)$

### Recursive let Expressions

Recursive **let** expressions are written as:

### Recursive let Expressions:

**let**  $f = \lambda a:A \bullet E(f)$  **in**  $B(f,a)$  **end**

is “the same” as:

**let**  $f = YF$  **in**  $B(f,a)$  **end**

where:

$F \equiv \lambda g \bullet \lambda a \bullet (E(g))$  and  $YF = F(YF)$

### Predicative let Expressions

Predicative **let** expressions:

### Predicative let Expressions:

**let**  $a:A \bullet \mathcal{P}(a)$  **in**  $\mathcal{B}(a)$  **end**

express the selection of a value  $a$  of type  $A$  which satisfies a predicate  $\mathcal{P}(a)$  for evaluation in the body  $\mathcal{B}(a)$ .

### Pattern and “Wild Card” let Expressions

*Patterns* and *wild cards* can be used:

### Patterns:

**let**  $\{a\} \cup s = \text{set}$  **in** ... **end**

**let**  $\{a, \_ \} \cup s = \text{set}$  **in** ... **end**

**let**  $(a,b,\dots,c) = \text{cart}$  **in** ... **end**

**let**  $(a, \_, \dots, c) = \text{cart}$  **in** ... **end**

**let**  $\langle a \rangle^\ell = \text{list}$  **in** ... **end**

**let**  $\langle a, \_, b \rangle^\ell = \text{list}$  **in** ... **end**

**let**  $[a \mapsto b] \cup m = \text{map}$  **in** ... **end**

**let**  $[a \mapsto b, \_] \cup m = \text{map}$  **in** ... **end**

### Conditionals

Various kinds of conditional expressions are offered by RSL:

**Conditionals:**

```
if b_expr then c_expr else a_expr
end
```

```
if b_expr then c_expr end \equiv /* same as: */
 if b_expr then c_expr else skip end
```

```
if b_expr_1 then c_expr_1
elseif b_expr_2 then c_expr_2
elseif b_expr_3 then c_expr_3
...
elseif b_expr_n then c_expr_n end
```

```
case expr of
 choice_pattern_1 \rightarrow expr_1,
 choice_pattern_2 \rightarrow expr_2,
 ...
 choice_pattern_n_or_wild_card \rightarrow expr_n
end
```

Operator/Operand Expressions

**Operator/Operand Expressions:**

```
 \langle Expr $\rangle ::=$
 \langle Prefix_Op \rangle \langle Expr \rangle
 | \langle Expr \rangle \langle Infix_Op \rangle \langle Expr \rangle
 | \langle Expr \rangle \langle Suffix_Op \rangle
 | ...
 \langle Prefix_Op $\rangle ::=$
 - | ~ | \cup | \cap | card | len | inds | elems | hd | tl | dom | rng
 \langle Infix_Op $\rangle ::=$
 = | \neq | \equiv | + | - | * | \uparrow | / | < | \leq | \geq | > | \wedge | \vee | \Rightarrow
 | \in | \notin | \cup | \cap | \ | \subset | \subseteq | \supseteq | \supset | \wedge | \dagger | \circ
 \langle Suffix_Op $\rangle ::=$!
```

Imperative Constructs

## A.6 Statements and State Changes

Often, following the RAISE method, software development starts with highly abstract-applicative constructs which, through stages of refinements, are turned into concrete and imperative constructs. Imperative constructs are thus inevitable in RSL.

**Statements and State Change:**

```
Unit
value
stmt: Unit \rightarrow Unit
stmt()
```

- Statements accept no arguments.

- Statement execution changes the state (of declared variables).
- **Unit**  $\rightarrow$  **Unit** designates a function from states to states.
- Statements, *stmt*, denote state-to-state changing functions.
- Writing () as “only” arguments to a function “means” that () is an argument of type **Unit**.

Variables and Assignment

**Variables and Assignment:**

0. **variable** *v*:Type := expression
1. *v* := expr

## A.7 Statement Sequences and skip

Sequencing is expressed using the ‘;’ operator. **skip** is the empty statement having no value or side-effect.

**Statement Sequences and skip:**

2. **skip**
3. *stm\_1*; *stm\_2*; ...; *stm\_n*

## A.8 Imperative Conditionals

**Imperative Conditionals:**

4. **if** expr **then** *stm\_c* **else** *stm\_a* **end**
5. **case** *e* **of**: *p\_1*  $\rightarrow$  *S\_1*(*p\_1*), ..., *p\_n*  $\rightarrow$  *S\_n*(*p\_n*) **end**

Iterative Conditionals

**Iterative Conditionals:**

6. **while** expr **do** *stm* **end**
7. **do** *stmt* **until** expr **end**

## A.9 Iterative Sequencing

**Iterative Sequencing:**

8. **for** *e* **in** *list\_expr* • *P*(*b*) **do** *S*(*b*) **end**

Process Constructs

## A.10 Process Channels

Let *A* and *B* stand for two types of (channel) messages and *i*:KIdx for channel array indexes, then:

**Process Channels:**

```
channel c:A
channel { k[i]:B • i:KIdx }
```

declare a channel,  $c$ , and a set (an array) of channels,  $k[i]$ , capable of communicating values of the designated types (A and B).

**Process Composition**

Let  $P$  and  $Q$  stand for names of process functions, i.e., of functions which express willingness to engage in input and/or output events, thereby communicating over declared channels. Let  $P()$  and  $Q$  stand for process expressions, then:

**Process Composition:**

```
P || Q Parallel composition
P [] Q Nondeterministic external choice (either/or)
P [] Q Nondeterministic internal choice (either/or)
P # Q Interlock parallel composition
```

express the parallel ( $||$ ) of two processes, or the nondeterministic choice between two processes: either external ( $[]$ ) or internal ( $[]$ ). The interlock ( $\#$ ) composition expresses that the two processes are forced to communicate only with one another, until one of them terminates.

**Input/Output Events**

Let  $c$ ,  $k[i]$  and  $e$  designate channels of type A and B, then:

**Input/Output Events:**

```
c ?, k[i] ? Input
c ! e, k[i] ! e Output
```

expresses the willingness of a process to engage in an event that “reads” an input, respectively “writes” an output.

**Process Definitions**

The below signatures are just examples. They emphasise that process functions must somehow express, in their signature, via which channels they wish to engage in input and output events.

**Process Definitions:****value**

```
P: Unit → in c out k[i]
Unit
Q: i:KIdx → out c in k[i] Unit
```

```
P() ≡ ... c ? ... k[i] ! e ...
Q(i) ≡ ... k[i] ? ... c ! e ...
```

The process function definitions (i.e., their bodies) express possible events.

**Simple RSL Specifications**

Often, we do not want to encapsulate small specifications in schemes, classes, and objects, as is often done in RSL. An RSL specification is simply a sequence of one or more types, values (including functions), variables, channels and axioms:

**Simple RSL Specifications:**

```
type
...
variable
...
channel
...
value
...
axiom
...
```

In practice a full specification repeats the above listings many times, once for each “module” (i.e., aspect, facet, view) of specification. Each of these modules may be “wrapped” into scheme, class or object definitions.<sup>1</sup>

---

<sup>1</sup>For schemes, classes and objects we refer to [13, Chap. 10]