

The Definition of Standard ML

The Definition of Standard ML (Revised)

Robin Milner, Mads Tofte, Robert Harper and David MacQueen

The MIT Press
Cambridge, Massachusetts
London, England

©1997 Robin Milner

All rights reserved. No part of this book may be reproduced in any form by any electronic or mechanical means (including photocopying, recording, or information storage and retrieval) without permission in writing from the publisher.

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

The definition of standard ML: revised / Robin Milner . . . et al.

p. cm.

Includes bibliographical references and index.

ISBN 0-262-63181-4 (alk. paper)

1. ML (Computer program language) I. Milner, R. (Robin), 1934-
QA76.73.M6D44 1997
005.13'3—dc21

97-59

CIP

Contents

1	Introduction	1
2	Syntax of the Core	3
2.1	Reserved Words	3
2.2	Special constants	3
2.3	Comments	4
2.4	Identifiers	5
2.5	Lexical analysis	6
2.6	Infix operators	6
2.7	Derived Forms	7
2.8	Grammar	7
2.9	Syntactic Restrictions	9
3	Syntax of Modules	12
3.1	Reserved Words	12
3.2	Identifiers	12
3.3	Infix operators	12
3.4	Grammar for Modules	13
3.5	Syntactic Restrictions	13
4	Static Semantics for the Core	16
4.1	Simple Objects	16
4.2	Compound Objects	17
4.3	Projection, Injection and Modification	17
4.4	Types and Type functions	19
4.5	Type Schemes	19
4.6	Scope of Explicit Type Variables	20
4.7	Non-expansive Expressions	21
4.8	Closure	21
4.9	Type Structures and Type Environments	22
4.10	Inference Rules	23
4.11	Further Restrictions	31
5	Static Semantics for Modules	32
5.1	Semantic Objects	32
5.2	Type Realisation	32

5.3	Signature Instantiation	33
5.4	Functor Signature Instantiation	33
5.5	Enrichment	33
5.6	Signature Matching	34
5.7	Inference Rules	34
6	Dynamic Semantics for the Core	41
6.1	Reduced Syntax	41
6.2	Simple Objects	41
6.3	Compound Objects	42
6.4	Basic Values	43
6.5	Basic Exceptions	43
6.6	Function Closures	43
6.7	Inference Rules	44
7	Dynamic Semantics for Modules	53
7.1	Reduced Syntax	53
7.2	Compound Objects	53
7.3	Inference Rules	55
8	Programs	60
A	Appendix: Derived Forms	63
B	Appendix: Full Grammar	71
C	Appendix: The Initial Static Basis	77
D	Appendix: The Initial Dynamic Basis	80
E	Overloading	82
E.1	Overloaded special constants	82
E.2	Overloaded value identifiers	83
F	Appendix: The Development of ML	85
G	Appendix: What is New?	95
G.1	Type Abbreviations in Signatures	95
G.2	Opaque Signature Matching	96

G.3	Sharing	98
G.3.1	Type Sharing	98
G.3.2	The equality attribute of specified types	99
G.3.3	Structure Sharing	100
G.4	Value Polymorphism	101
G.5	Identifier Status	102
G.6	Replication of Datatypes	103
G.7	Local Datatypes	105
G.8	Principal Environments	106
G.9	Consistency and Admissibility	107
G.10	Special Constants	107
G.11	Comments	107
G.12	Infix Operators	108
G.13	Non-expansive Expressions	108
G.14	Rebinding of built-in identifiers	108
G.15	Grammar for Modules	108
G.16	Closure Restrictions	108
G.17	Specifications	109
G.18	Scope of Explicit Type Variables	109
G.19	The Initial Basis	109
G.20	Overloading	110
G.21	Reals	110

Preface

A precise description of a programming language is a prerequisite for its implementation and for its use. The description can take many forms, each suited to a different purpose. A common form is a reference manual, which is usually a careful narrative description of the meaning of each construction in the language, often backed up with a formal presentation of the grammar (for example, in Backus-Naur form). This gives the programmer enough understanding for many of his purposes. But it is ill-suited for use by an implementer, or by someone who wants to formulate laws for equivalence of programs, or by a programmer who wants to design programs with mathematical rigour.

This document is a formal description of both the *grammar* and the *meaning* of a language which is both designed for large projects and widely used. As such, it aims to serve the whole community of people seriously concerned with the language. At a time when it is increasingly understood that programs must withstand rigorous analysis, particularly for systems where safety is critical, a rigorous language presentation is even important for negotiators and contractors; for a robust program written in an insecure language is like a house built upon sand.

Most people have not looked at a rigorous language presentation before. To help them particularly, but also to put the present work in perspective for those more theoretically prepared, it will be useful here to say something about three things: the nature of Standard ML, the task of language definition in general, and the form of the present Definition. We also briefly describe the recent revisions to the Definition.

Standard ML

Standard ML is a functional programming language, in the sense that the full power of mathematical functions is present. But it grew in response to a particular programming task, for which it was equipped also with full imperative power, and a sophisticated exception mechanism. It has an advanced form of parametric modules, aimed at organised development of large programs. Finally it is strongly typed, and it was the first language to provide a particular form of polymorphic type which makes the strong typing remarkably flexible. This combination of ingredients has not made it unduly large, but their novelty has been a fascinating challenge to semantic method (of which we say more below).

ML has evolved over twenty years as a fusion of many ideas from many people. This evolution is described in some detail in Appendix F of the book, where also we acknowledge all those who have contributed to it, both in design and in implementation.

‘ML’ stands for *meta language*; this is the term logicians use for a language in which other (formal or informal) languages are discussed and analysed. Originally ML was conceived as a medium for finding and performing proofs in a logical language. Conducting rigorous argument as dialogue between person and machine has been a growing research topic throughout these twenty years. The difficulties are enormous, and make stern demands upon the programming language which is used for this dialogue. Those who are not familiar with computer-assisted reasoning may be surprised that a programming language, which was designed for this rather esoteric activity, should ever lay claim to being *generally* useful. On reflection, they should not be surprised. LISP is a prime example of a language invented for esoteric purposes and becoming widely used. LISP was invented for use in artificial intelligence (AI); the important thing about AI here is not that it is esoteric, but that it is difficult and varied; so much so, that anything which works well for it must work well for many other applications too.

The same can be said about the initial purpose of ML, but with a different emphasis. Rigorous proofs are complex things, which need varied and sophisticated presentation – particularly on the screen in interactive mode. Furthermore the proof methods, or strategies, involved are some of the most complex algorithms which we know. This all applies equally to AI, but one demand is made more strongly by proof than perhaps by any other application: the demand for rigour.

This demand established the character of ML. In order to be sure that, when the user and the computer claim to have together performed a rigorous argument, their claim is justified, it was seen that the language must be strongly typed. On the other hand, to be useful in a difficult application, the type system had to be rather flexible, and permit the machine to guide the user rather than impose a burden upon him. A reasonable solution was found, in which the machine helps the user significantly by inferring his types for him. Thereby the machine also confers complete reliability on his programs, in this sense: If a program claims that a certain result follows from the rules of reasoning which the user has supplied, then the claim may be fully trusted.

The principle of inferring useful structural information about programs is also represented, at the level of program modules, by the inference of *signa-*

tures. Signatures describe the interfaces between modules, and are vital for robust large-scale programs. When the user combines modules, the signature discipline prevents him from mismatching their interfaces. By programming with interfaces and parametric modules, it becomes possible to focus on the structure of a large system, and to compile parts of it in isolation from one another – even when the system is incomplete.

This emphasis on types and signatures has had a profound effect on the language Definition. Over half this document is devoted to inferring types and signatures for programs. But the method used is exactly the same as for inferring what *values* a program delivers; indeed, a type or signature is the result of a kind of abstract evaluation of a program phrase.

In designing ML, the interplay among three activities – language design, definition and implementation – was extremely close. This was particularly true for the newest part, the parametric modules. This part of the language grew from an initial proposal by David MacQueen, itself highly developed; but both formal definition and implementation had a strong influence on the detailed design. In general, those who took part in the three activities cannot now imagine how they could have been properly done separately.

Language Definition

Every programming language presents its own conceptual view of computation. This view is usually indicated by the names used for the phrase classes of the language, or by its keywords: terms like *package*, *module*, *structure*, *exception*, *channel*, *type*, *procedure*, *reference*, *sharing*, These terms also have their abstract counterparts, which may be called *semantic objects*; these are what people really have in mind when they use the language, or discuss it, or think in it. Also, it is these objects, not the syntax, which represent the particular conceptual view of each language; they are the character of the language. Therefore a definition of the language must be in terms of these objects.

As is commonly done in programming language semantics, we shall loosely talk of these semantic objects as *meanings*. Of course, it is perfectly possible to understand the semantic theory of a language, and yet be unable to understand the meaning of a particular program, in the sense of its *intention* or *purpose*. The aim of a language definition is not to formalise everything which could possibly be called the meaning of a program, but to establish a theory of semantic objects upon which the understanding of particular

programs may rest.

The job of a language-definer is twofold. First – as we have already suggested – he must create a world of meanings appropriate for the language, and must find a way of saying what these meanings precisely are. Here, he meets a problem; notation of *some* kind must be used to denote and describe these meanings – but not a *programming language* notation, unless he is passing the buck and defining one programming language in terms of another. Given a concern for rigour, mathematical notation is an obvious choice. Moreover, it is not enough just to write down mathematical definitions. The world of meanings only becomes meaningful if the objects possess nice properties, which make them tractable. So the language-definer really has to develop a small *theory* of his meanings, in the same way that a mathematician develops a theory. Typically, after initially defining some objects, the mathematician goes on to verify properties which indicate that they are objects worth studying. It is this part, a kind of scene-setting, which the language-definer shares with the mathematician. Of course he can take many objects and their theories directly from mathematics, such as functions, relations, trees, sequences, But he must also give some special theory for the objects which make his language particular, as we do for types, structures and signatures in this book; otherwise his language definition may be formal but will give no insight.

The second part of the definer's job is to define *evaluation* precisely. This means that he must define at least *what* meaning, M , results from evaluating any phrase P of his language (though he need not explain exactly *how* the meaning results; that is he need not give the full detail of every computation). This part of his job must be formal to some extent, if only because the phrases P of his language are indeed formal objects. But there is another reason for formality. The task is complex and error-prone, and therefore demands a high level of explicit organisation (which is, largely, the meaning of 'formality'); moreover, it will be used to specify an equally complex, error-prone and formal construction: an implementation.

We shall now explain the keystone of our semantic method. First, we need a slight but important refinement. A phrase P is never evaluated *in vacuo* to a meaning M , but always *against a background*; this background – call it B – is itself a semantic object, being a distillation of the meanings preserved from evaluation of earlier phrases (typically variable declarations, procedure declarations, etc.). In fact evaluation is background-dependent – M depends upon B as well as upon P .

The keystone of the method, then, is a certain kind of assertion about evaluation; it takes the form

$$B \vdash P \Rightarrow M$$

and may be pronounced: ‘Against the background B , the phrase P evaluates to the meaning M ’. *The formal purpose of this Definition is no more, and no less, than to decree exactly which assertions of this form are true.* This could be achieved in many ways. We have chosen to do it in a structured way, as others have, by giving rules which allow assertions about a *compound* phrase P to be inferred from assertions about its *constituent* phrases P_1, \dots, P_n .

We have written the Definition in a form suggested by the previous remarks. That is, we have defined our semantic objects in mathematical notation which is completely independent of Standard ML, and we have developed just enough of their theory to give sense to our rules of evaluation.

Following another suggestion above, we have factored our task by describing *abstract* evaluation – the inference and checking of types and signatures (which can be done at compile-time) – completely separately from *concrete* evaluation. It really is a factorisation, because a *full* value in all its glory – you can think of it as a concrete object with a type attached – never has to be presented.

The Revision of Standard ML

The Definition of Standard ML was published in 1990. Since then the implementation technology of the language has advanced enormously, and its users have multiplied. The language and its Definition have therefore incited close scrutiny, evaluation, much approval, sometimes strong criticism.

The originators of the language have sifted this response, and found that there are inadequacies in the original language and its formal Definition. They are of three kinds: missing features which many users want; complex and little-used features which most users can do without; and mistakes of definition. What is remarkable is that these inadequacies are rather few, and that they are rather uncontroversial.

This new version of the Definition addresses the three kinds of inadequacy respectively by additions, subtractions and corrections. But we have only made such amendments when one or more aspects of SML – the language itself, its usage, its implementation, its formal Definition – have thus become simpler, without complicating the other aspects. It is worth noting that

even the additions meet this criterion; for example we have introduced type abbreviations in signatures to simplify the use of the language, but the way we have done it has even simplified the Definition too. In fact, after our changes the formal Definition has fewer rules.

In this exercise we have consulted the major implementers and several users, and have found broad agreement. In the 1990 Definition it was predicted that further versions of the Definition would be produced as the language develops, with the intention to minimise the number of versions. This is the first revised version, and we foresee no others. The changes that have been made to the 1990 Definition are enumerated in Appendix G.

The resulting document is, we hope, valuable as the essential point of reference for Standard ML. If it is to play this role well, it must be supplemented by other literature. Many expository books have already been written, and this Definition will be useful as a background reference for their readers. We became convinced, while writing the 1990 Definition, that we could not discuss many questions without making it far too long. Such questions are: Why were certain design choices made? What are their implications for programming? Was there a good alternative meaning for some constructs, or was our hand forced? What different forms of phrase are equivalent? What is the proof of certain claims? Many of these questions are not answered by pedagogic texts either. We therefore wrote a Commentary on the 1990 Definition to assist people in reading it, and to serve as a bridge between the Definition and other texts. Though in part outdated by the present revision, the Commentary still largely fulfils its purpose.

There exist several textbooks on programming with Standard ML[46, 44, 56, 50]. The second edition of Paulson's book[46] conforms with the present revision.

We wish to thank Dave Berry, Lars Birkedal, Martin Elsman, Stefan Kahrs and John Reppy for many detailed comments and suggestions which have assisted the revision.

Robin Milner Mads Tofte Robert Harper David MacQueen

November 1996

1 Introduction

This document formally defines Standard ML.

To understand the method of definition, at least in broad terms, it helps to consider how an implementation of ML is naturally organised. ML is an interactive language, and a *program* consists of a sequence of *top-level declarations*; the execution of each declaration modifies the top-level environment, which we call a *basis*, and reports the modification to the user.

In the execution of a declaration there are three phases: *parsing*, *elaboration*, and *evaluation*. Parsing determines the grammatical form of a declaration. Elaboration, the *static* phase, determines whether it is well-typed and well-formed in other ways, and records relevant type or form information in the basis. Finally evaluation, the *dynamic* phase, determines the value of the declaration and records relevant value information in the basis. Corresponding to these phases, our formal definition divides into three parts: grammatical rules, elaboration rules, and evaluation rules. Furthermore, the basis is divided into the *static* basis and the *dynamic* basis; for example, a variable which has been declared is associated with a type in the static basis and with a value in the dynamic basis.

In an implementation, the basis need not be so divided. But for the purpose of formal definition, it eases presentation and understanding to keep the static and dynamic parts of the basis separate. This is further justified by programming experience. A large proportion of errors in ML programs are discovered during elaboration, and identified as errors of type or form, so it follows that it is useful to perform the elaboration phase separately. In fact, elaboration without evaluation is part of what is normally called *compilation*; once a declaration (or larger entity) is compiled one wishes to evaluate it – repeatedly – without re-elaboration, from which it follows that it is useful to perform the evaluation phase separately.

A further factoring of the formal definition is possible, because of the structure of the language. ML consists of a lower level called the *Core language* (or *Core* for short), a middle level concerned with programming-in-the-large called *Modules*, and a very small upper level called *Programs*. With the three phases described above, there is therefore a possibility of nine components in the complete language definition. We have allotted one section to each of these components, except that we have combined the parsing, elaboration and evaluation of Programs in one section. The scheme for the

ensuing seven sections is therefore as follows:

	<i>Core</i>	<i>Modules</i>	<i>Programs</i>
<i>Syntax</i>	Section 2	Section 3	
<i>Static Semantics</i>	Section 4	Section 5	Section 8
<i>Dynamic Semantics</i>	Section 6	Section 7	

The Core provides many phrase classes, for programming convenience. But about half of these classes are derived forms, whose meaning can be given by translation into the other half which we call the *Bare* language. Thus each of the three parts for the Core treats only the bare language; the derived forms are treated in Appendix A. This appendix also contains a few derived forms for Modules. A full grammar for the language is presented in Appendix B.

In Appendices C and D the *initial basis* is detailed. This basis, divided into its static and dynamic parts, contains the static and dynamic meanings of a small set of predefined identifiers. A richer basis is defined in a separate document [18].

The semantics is presented in a form known as Natural Semantics. It consists of a set of rules allowing *sentences* of the form

$$A \vdash \textit{phrase} \Rightarrow A'$$

to be inferred, where A is often a basis (static or dynamic) and A' a semantic object – often a type in the static semantics and a value in the dynamic semantics. One should read such a sentence as follows: “against the background provided by A , the phrase *phrase* elaborates – or evaluates – to the object A' ”. Although the rules themselves are formal the semantic objects, particularly the static ones, are the subject of a mathematical theory which is presented in a succinct form in the relevant sections.

The robustness of the semantics depends upon theorems. Usually these have been proven, but the proof is not included.

2 Syntax of the Core

2.1 Reserved Words

The following are the *reserved words* used in the Core. They may not (except =) be used as identifiers.

```
abstype  and  andalso  as  case  datatype  do  else
end      exception  fn  fun  handle  if  in  infix
infixr  let  local  nonfix  of  op  open  orelse
raise   rec  then  type  val  with  withtype  while
( )    [ ]  { }  ,  :  ;  ...  _  |  =  =>  ->  #
```

2.2 Special constants

An *integer constant (in decimal notation)* is an optional negation symbol (~) followed by a non-empty sequence of decimal digits 0, ..., 9. An *integer constant (in hexadecimal notation)* is an optional negation symbol followed by 0x followed by a non-empty sequence of hexadecimal digits 0, ..., 9 and a, ..., f. (A, ..., F may be used as alternatives for a, ..., f.)

A *word constant (in decimal notation)* is 0w followed by a non-empty sequence of decimal digits. A *word constant (in hexadecimal notation)* is 0wx followed by a non-empty sequence of hexadecimal digits. A *real constant* is an integer constant in decimal notation, possibly followed by a point (.) and one or more decimal digits, possibly followed by an exponent symbol (E or e) and an integer constant in decimal notation; at least one of the optional parts must occur, hence no integer constant is a real constant. Examples: 0.7 3.32E5 3E~7 . Non-examples: 23 .3 4.E5 1E2.0 .

We assume an underlying alphabet of N characters ($N \geq 256$), numbered 0 to $N - 1$, which agrees with the ASCII character set on the characters numbered 0 to 127. The interval $[0, N - 1]$ is called the *ordinal range* of the alphabet. A *string constant* is a sequence, between quotes ("), of zero or more printable characters (i.e., numbered 33–126), spaces or escape sequences. Each escape sequence starts with the escape character \ , and stands for a character sequence. The escape sequences are:

\a	A single character interpreted by the system as alert (ASCII 7)
\b	Backspace (ASCII 8)
\t	Horizontal tab (ASCII 9)

<code>\n</code>	Linefeed, also known as newline (ASCII 10)
<code>\v</code>	Vertical tab (ASCII 11)
<code>\f</code>	Form feed (ASCII 12)
<code>\r</code>	Carriage return (ASCII 13)
<code>\^c</code>	The control character c , where c may be any character with number 64–95. The number of <code>\^c</code> is 64 less than the number of c .
<code>\ddd</code>	The single character with number ddd (3 decimal digits denoting an integer in the ordinal range of the alphabet).
<code>\uxxxx</code>	The single character with number $xxxx$ (4 hexadecimal digits denoting an integer in the ordinal range of the alphabet).
<code>\"</code>	"
<code>\\</code>	\
<code>\f · · f\</code>	This sequence is ignored, where $f · · f$ stands for a sequence of one or more formatting characters.

The *formatting characters* are a subset of the non-printable characters including at least space, tab, newline, formfeed. The last form allows long strings to be written on more than one line, by writing `\` at the end of one line and at the start of the next.

A *character constant* is a sequence of the form `#s`, where s is a string constant denoting a string of size one character.

Libraries may provide multiple numeric types and multiple string types. To each string type corresponds an alphabet with ordinal range $[0, N - 1]$ for some $N \geq 256$; each alphabet must agree with the ASCII character set on the characters numbered 0 to 127. When multiple alphabets are supported, all characters of a given string constant are interpreted over the same alphabet. For each special constant, overloading resolution is used for determining the type of the constant (see Appendix E).

We denote by SCon the class of *special constants*, i.e., the integer, real, word, character and string constants; we shall use scn to range over SCon.

2.3 Comments

A *comment* is any character sequence within comment brackets `(* *)` in which comment brackets are properly nested. No space is allowed between the two characters which make up a comment bracket `(*` or `*)`. An unmatched `(*` should be detected by the compiler.

2.4 Identifiers

The classes of *identifiers* for the Core are shown in Figure 1. We use *vid*, *tyvar* to range over VId, TyVar etc. For each class X marked “long” there is a class longX of *long identifiers*; if *x* ranges over X then *longx* ranges over longX. The syntax of these long identifiers is given by the following:

$$\begin{aligned} \text{long}x & ::= x && \text{identifier} \\ & \quad \text{strid}_1 \dots \text{strid}_n . x && \text{qualified identifier } (n \geq 1) \end{aligned}$$

The qualified identifiers constitute a link between the Core and the Modules. Throughout this document, the term “identifier”, occurring without an adjective, refers to non-qualified identifiers only.

An identifier is either *alphanumeric*: any sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime, or *symbolic*: any non-empty sequence of the following *symbols*

! % & \$ # + - / : < = > ? @ \ ~ ' ^ | *

In either case, however, reserved words are excluded. This means that for example # and | are not identifiers, but ## and |=| are identifiers. The only exception to this rule is that the symbol =, which is a reserved word, is also allowed as an identifier to stand for the equality predicate. The identifier = may not be re-bound; this precludes any syntactic ambiguity.

A type variable *tyvar* may be any alphanumeric identifier starting with a prime; the subclass EtyVar of TyVar, the *equality* type variables, consists of those which start with two or more primes. The classes VId, TyCon and Lab are represented by identifiers not starting with a prime. However, * is excluded from TyCon, to avoid confusion with the derived form of tuple type (see Figure 23). The class Lab is extended to include the *numeric* labels 1 2 3 ..., i.e. any numeral not starting with 0. The identifier class StrId is represented by alphanumeric identifiers not starting with a prime.

TyVar is therefore disjoint from the other four classes. Otherwise, the syntax class of an occurrence of identifier *id* in a Core phrase (ignoring derived forms, Section 2.7) is determined thus:

VId	(value identifiers)	long
TyVar	(type variables)	
TyCon	(type constructors)	long
Lab	(record labels)	
StrId	(structure identifiers)	long

Figure 1: Identifiers

1. Immediately before “.” – i.e. in a long identifier – or in an `open` declaration, *id* is a structure identifier. The following rules assume that all occurrences of structure identifiers have been removed.
2. At the start of a component in a record type, record pattern or record expression, *id* is a record label.
3. Elsewhere in types *id* is a type constructor.
4. Elsewhere, *id* is a value identifier.

By means of the above rules a compiler can determine the class to which each identifier occurrence belongs; for the remainder of this document we shall therefore assume that the classes are all disjoint.

2.5 Lexical analysis

Each item of lexical analysis is either a reserved word, a numeric label, a special constant or a long identifier. Comments and formatting characters separate items (except within string constants; see Section 2.2) and are otherwise ignored. At each stage the longest next item is taken.

2.6 Infix operators

An identifier may be given *infix status* by the `infix` or `infixr` directive, which may occur as a declaration; this status only pertains to its use as a *vid* within the scope (see below) of the directive, and in these uses it is called an *infix operator*. (Note that qualified identifiers never have infix status.) If *vid* has infix status, then “*exp*₁ *vid* *exp*₂” (resp. “*pat*₁ *vid* *pat*₂”) may occur – in parentheses if necessary – wherever the application “*vid*{1=*exp*₁, 2=*exp*₂}” or its derived form “*vid*(*exp*₁, *exp*₂)” (resp. “*vid*(*pat*₁, *pat*₂)”) would otherwise occur. On the other hand, an occurrence of any long identifier (qualified or not) prefixed by `op` is treated as non-infix. The only required use of `op` is in prefixing a non-infix occurrence of an identifier *vid* which has infix status; elsewhere `op`, where permitted, has no effect. Infix status is cancelled by the `nonfix` directive. We refer to the three directives collectively as *fixity directives*.

The form of the fixity directives is as follows ($n \geq 1$):

$$\text{infix } \langle d \rangle \text{ vid}_1 \cdots \text{vid}_n$$

$$\text{infixr } \langle d \rangle \text{ } vid_1 \cdots vid_n$$

$$\text{nonfix } vid_1 \cdots vid_n$$

where $\langle d \rangle$ is an optional decimal digit d indicating binding precedence. A higher value of d indicates tighter binding; the default is 0. `infix` and `infixr` dictate left and right associativity respectively. In an expression of the form $exp_1 \text{ } vid_1 \text{ } exp_2 \text{ } vid_2 \text{ } exp_3$, where vid_1 and vid_2 are infix operators with the same precedence, either both must associate to the left or both must associate to the right. For example, suppose that `<<` and `>>` have equal precedence, but associate to the left and right respectively; then

$$\begin{array}{ll} x \ll y \ll z & \text{parses as } (x \ll y) \ll z \\ x \gg y \gg z & \text{parses as } x \gg (y \gg z) \\ x \ll y \gg z & \text{is illegal} \\ x \gg y \ll z & \text{is illegal} \end{array}$$

The precedence of infix operators relative to other expression and pattern constructions is given in Appendix B.

The *scope* of a fixity directive *dir* is the ensuing program text, except that if *dir* occurs in a declaration *dec* in either of the phrases

$$\text{let } dec \text{ in } \cdots \text{end}$$

$$\text{local } dec \text{ in } \cdots \text{end}$$

then the scope of *dir* does not extend beyond the phrase. Further scope limitations are imposed for Modules (see Section 3.3).

These directives and `op` are omitted from the semantic rules, since they affect only parsing.

2.7 Derived Forms

There are many standard syntactic forms in ML whose meaning can be expressed in terms of a smaller number of syntactic forms, called the *bare language*. These derived forms, and their equivalent forms in the bare language, are given in Appendix A.

2.8 Grammar

The phrase classes for the Core are shown in Figure 2. We use the variable *atexp* to range over `AtExp`, etc. The grammatical rules for the Core are shown in Figures 3 and 4.

AtExp	atomic expressions
ExpRow	expression rows
Exp	expressions
Match	matches
Mrule	match rules
Dec	declarations
ValBind	value bindings
TypBind	type bindings
DatBind	datatype bindings
ConBind	constructor bindings
ExBind	exception bindings
AtPat	atomic patterns
PatRow	pattern rows
Pat	patterns
Ty	type expressions
TyRow	type-expression rows

Figure 2: Core Phrase Classes

The following conventions are adopted in presenting the grammatical rules, and in their interpretation:

- The brackets $\langle \rangle$ enclose optional phrases.
- For any syntax class X (over which x ranges) we define the syntax class $Xseq$ (over which $xseq$ ranges) as follows:

$$\begin{array}{ll}
 xseq ::= x & \text{(singleton sequence)} \\
 & \text{(empty sequence)} \\
 & (x_1, \dots, x_n) \quad \text{(sequence, } n \geq 1)
 \end{array}$$

(Note that the “ \dots ” used here, meaning syntactic iteration, must not be confused with “ \dots ” which is a reserved word of the language.)

- Alternative forms for each phrase class are in order of decreasing precedence; this resolves ambiguity in parsing, as explained in Appendix B.
- L (resp. R) means left (resp. right) association.

- The syntax of types binds more tightly than that of expressions.
- Each iterated construct (e.g. *match*, ...) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. “*fn match*”, if this occurs within a larger match.

<i>atpat</i>	::=	<i>_</i>	wildcard
		<i>scon</i>	special constant
		$\langle \text{op} \rangle \text{longvid}$	value identifier
		$\{ \langle \text{patrow} \rangle \}$	record
		(pat)	
<i>patrow</i>	::=	\dots	wildcard
		$\text{lab} = \text{pat} \langle , \text{patrow} \rangle$	pattern row
<i>pat</i>	::=	<i>atpat</i>	atomic
		$\langle \text{op} \rangle \text{longvid} \text{atpat}$	constructed pattern
		$\text{pat}_1 \text{vid} \text{pat}_2$	infix value construction
		$\text{pat} : \text{ty}$	typed
		$\langle \text{op} \rangle \text{vid} \langle : \text{ty} \rangle \text{as} \text{pat}$	layered
<i>ty</i>	::=	<i>tyvar</i>	type variable
		$\{ \langle \text{tyrow} \rangle \}$	record type expression
		<i>tyseq longtycon</i>	type construction
		$\text{ty} \rightarrow \text{ty}'$	function type expression (R)
		(ty)	
<i>tyrow</i>	::=	$\text{lab} : \text{ty} \langle , \text{tyrow} \rangle$	type-expression row

Figure 3: Grammar: Patterns and Type expressions

2.9 Syntactic Restrictions

- No expression row, pattern row or type-expression row may bind the same *lab* twice.
- No binding *valbind*, *tybind*, *datbind* or *exbind* may bind the same identifier twice; this applies also to value identifiers within a *datbind*.

- No *tyvarseq* may contain the same *tyvar* twice.
- For each value binding $pat = exp$ within **rec**, *exp* must be of the form **fn** *match*. The derived form of function-value binding given in Appendix A, page 67, necessarily obeys this restriction.
- No *datbind*, *valbind* or *exbind* may bind **true**, **false**, **nil**, **::** or **ref**. No *datbind* or *exbind* may bind it.
- No real constant may occur in a pattern.
- In a value declaration **val** *tyvarseq* *valbind*, if *valbind* contains another value declaration **val** *tyvarseq'* *valbind'* then *tyvarseq* and *tyvarseq'* must be disjoint. In other words, no type variable may be scoped by two value declarations of which one occurs inside the other. This restriction applies after *tyvarseq* and *tyvarseq'* have been extended to include implicitly scoped type variables, as explained in Section 4.6.

<i>atexp</i>	::=	<i>scon</i> <i><op>longvid</i> { <i><exprow></i> } let <i>dec</i> in <i>exp</i> end (<i>exp</i>)	special constant value identifier record local declaration
<i>exprow</i>	::=	<i>lab = exp</i> < , <i>exprow</i> >	expression row
<i>exp</i>	::=	<i>atexp</i> <i>exp atexp</i> <i>exp₁ vid exp₂</i> <i>exp : ty</i> <i>exp handle match</i> raise <i>exp</i> fn <i>match</i>	atomic application (L) infix application typed (L) handle exception raise exception function
<i>match</i>	::=	<i>mrule</i> < <i>match</i> >	
<i>mrule</i>	::=	<i>pat => exp</i>	
<i>dec</i>	::=	val <i>tyvarseq valbind</i> type <i>typbind</i> datatype <i>datbind</i> datatype <i>tycon</i> == datatype <i>longtycon</i> abstype <i>datbind with dec end</i> exception <i>exbind</i> local <i>dec₁</i> in <i>dec₂</i> end open <i>longstrid₁ ... longstrid_n</i> <i>dec₁</i> <;> <i>dec₂</i> infix < <i>d</i> > <i>vid₁ ... vid_n</i> infixr < <i>d</i> > <i>vid₁ ... vid_n</i> nonfix <i>vid₁ ... vid_n</i>	value declaration type declaration datatype declaration datatype replication abstype declaration exception declaration local declaration open declaration ($n \geq 1$) empty declaration sequential declaration infix (L) directive infix (R) directive nonfix directive
<i>valbind</i>	::=	<i>pat = exp</i> < and <i>valbind</i> > rec <i>valbind</i>	
<i>typbind</i>	::=	<i>tyvarseq tycon = ty</i> < and <i>typbind</i> >	
<i>datbind</i>	::=	<i>tyvarseq tycon = conbind</i> < and <i>datbind</i> >	
<i>conbind</i>	::=	< op > <i>vid</i> < of <i>ty</i> > < <i>conbind</i> >	
<i>exbind</i>	::=	< op > <i>vid</i> < of <i>ty</i> > < and <i>exbind</i> > < op > <i>vid =</i> < op > <i>longvid</i> < and <i>exbind</i> >	

Figure 4: Grammar: Expressions, Matches, Declarations and Bindings

3 Syntax of Modules

For Modules there are further reserved words, identifier classes and derived forms. There are no further special constants; comments and lexical analysis are as for the Core. The derived forms for modules appear in Appendix A.

3.1 Reserved Words

The following are the additional reserved words used in Modules.

```
eqtype    functor    include    sharing    sig
signature  struct     structure  where     :>
```

3.2 Identifiers

The additional identifier classes for Modules are SigId (signature identifiers) and FunId (functor identifiers). Functor and signature identifiers must be alphanumeric, not starting with a prime. The class of each identifier occurrence is determined by the grammatical rules which follow. Henceforth, therefore, we consider all identifier classes to be disjoint.

3.3 Infix operators

In addition to the scope rules for fixity directives given for the Core syntax, there is a further scope limitation: if *dir* occurs in a structure-level declaration *strdec* in any of the phrases

```
let strdec in ... end
```

```
local strdec in ... end
```

```
struct strdec end
```

then the scope of *dir* does not extend beyond the phrase.

One effect of this limitation is that fixity is local to a basic structure expression — in particular, to such an expression occurring as a functor body.

StrExp	structure expressions
StrDec	structure-level declarations
StrBind	structure bindings
SigExp	signature expressions
SigDec	signature declarations
SigBind	signature bindings
Spec	specifications
ValDesc	value descriptions
TypDesc	type descriptions
DatDesc	datatype descriptions
ConDesc	constructor descriptions
ExDesc	exception descriptions
StrDesc	structure descriptions
FunDec	functor declarations
FunBind	functor bindings
TopDec	top-level declarations

Figure 5: Modules Phrase Classes

3.4 Grammar for Modules

The phrase classes for Modules are shown in Figure 5. We use the variable *strexp* to range over StrExp, etc. The conventions adopted in presenting the grammatical rules for Modules are the same as for the Core. The grammatical rules are shown in Figures 6, 7 and 8.

3.5 Syntactic Restrictions

- No binding *strbind*, *sigbind*, or *funbind* may bind the same identifier twice.
- No description *valdesc*, *typdesc*, *datdesc*, *exdesc* or *strdesc* may describe the same identifier twice; this applies also to value identifiers within a *datdesc*.
- No *tyvarseq* may contain the same *tyvar* twice.

<i>strex</i>	::=	struct <i>strdec</i> end <i>longstrid</i> <i>strex</i> : <i>sigexp</i> <i>strex</i> :> <i>sigexp</i> <i>funid</i> (<i>strex</i>) let <i>strdec</i> in <i>strex</i> end	basic structure identifier transparent constraint opaque constraint functor application local declaration
<i>strdec</i>	::=	<i>dec</i> structure <i>strbind</i> local <i>strdec</i> ₁ in <i>strdec</i> ₂ end <i>strdec</i> ₁ <;> <i>strdec</i> ₂	declaration structure local empty sequential
<i>strbind</i>	::=	<i>strid</i> = <i>strex</i> < and <i>strbind</i> >	
<i>sigexp</i>	::=	sig <i>spec</i> end <i>sigid</i> <i>sigexp</i> where type <i>tyvarseq longtycon</i> = <i>ty</i>	basic signature identifier type realisation
<i>sigdec</i>	::=	signature <i>sigbind</i>	
<i>sigbind</i>	::=	<i>sigid</i> = <i>sigexp</i> < and <i>sigbind</i> >	

Figure 6: Grammar: Structure and Signature Expressions

- Any *tyvar* occurring on the right side of a *datdesc* of the form *tyvarseq tycon* = ... must occur in the *tyvarseq*; similarly, in signature expressions of the form *sigexp where type tyvarseq longtycon* = *ty*, any *tyvar* occurring in *ty* must occur in *tyvarseq*.
- No *datdesc*, *valdesc* or *exdesc* may describe **true**, **false**, **nil**, **::** or **ref**. No *datdesc* or *exdesc* may describe it.

<i>spec</i>	::=	val <i>valdesc</i>	value
		type <i>typdesc</i>	type
		eqtype <i>typdesc</i>	eqtype
		datatype <i>datdesc</i>	datatype
		datatype <i>tycon</i> -- datatype <i>longtycon</i>	replication
		exception <i>exdesc</i>	exception
		structure <i>strdesc</i>	structure
		include <i>sigexp</i>	include
			empty
		<i>spec</i> ₁ < ; > <i>spec</i> ₂	sequential
		<i>spec</i> sharing type	sharing
		<i>longtycon</i> ₁ = ... = <i>longtycon</i> _{<i>n</i>}	(<i>n</i> ≥ 2)
<i>valdesc</i>	::=	<i>vid</i> : <i>ty</i> < and <i>valdesc</i> >	
<i>typdesc</i>	::=	<i>tyvarseq</i> <i>tycon</i> < and <i>typdesc</i> >	
<i>datdesc</i>	::=	<i>tyvarseq</i> <i>tycon</i> = <i>condesc</i> < and <i>datdesc</i> >	
<i>condesc</i>	::=	<i>vid</i> < of <i>ty</i> > < <i>condesc</i> >	
<i>exdesc</i>	::=	<i>vid</i> < of <i>ty</i> > < and <i>exdesc</i> >	
<i>strdesc</i>	::=	<i>strid</i> : <i>sigexp</i> < and <i>strdesc</i> >	

Figure 7: Grammar: Specifications

<i>fundec</i>	::=	functor <i>funbind</i>	
<i>funbind</i>	::=	<i>funid</i> (<i>strid</i> : <i>sigexp</i>) = <i>strexpr</i>	functor binding
		< and <i>funbind</i> >	
<i>topdec</i>	::=	<i>strdec</i> < <i>topdec</i> >	structure-level declaration
		<i>sigdec</i> < <i>topdec</i> >	signature declaration
		<i>fundec</i> < <i>topdec</i> >	functor declaration

Restriction: No *topdec* may contain, as an initial segment, a *strdec* followed by a semicolon.

Figure 8: Grammar: Functors and Top-level Declarations

4 Static Semantics for the Core

Our first task in presenting the semantics – whether for Core or Modules, static or dynamic – is to define the objects concerned. In addition to the class of *syntactic* objects, which we have already defined, there are classes of so-called *semantic* objects used to describe the meaning of the syntactic objects. Some classes contain *simple* semantic objects; such objects are usually identifiers or names of some kind. Other classes contain *compound* semantic objects, such as types or environments, which are constructed from component objects.

4.1 Simple Objects

All semantic objects in the static semantics of the entire language are built from identifiers and two further kinds of simple objects: type constructor names and identifier status descriptors. Type constructor names are the values taken by type constructors; we shall usually refer to them briefly as type names, but they are to be clearly distinguished from type variables and type constructors. The simple object classes, and the variables ranging over them, are shown in Figure 9. We have included TyVar in the table to make visible the use of α in the semantics to range over TyVar.

α or <i>tyvar</i>	\in TyVar	type variables
t	\in TyName	type names
is	\in IdStatus = {c, e, v}	identifier status descriptors

Figure 9: Simple Semantic Objects

Each $\alpha \in$ TyVar possesses a boolean *equality* attribute, which determines whether or not it *admits equality*, i.e. whether it is a member of EtyVar (defined on page 5).

Each $t \in$ TyName has an arity $k \geq 0$, and also possesses an equality attribute. We denote the class of type names with arity k by TyName^(k).

With each special constant *scon* we associate a type name type(*scon*) which is either `int`, `real`, `word`, `char` or `string` as indicated by Section 2.2. (However, see Appendix E concerning types of overloaded special constants.)

4.2 Compound Objects

When A and B are sets $\text{Fin } A$ denotes the set of finite subsets of A , and $A \xrightarrow{\text{fin}} B$ denotes the set of *finite maps* (partial functions with finite domain) from A to B . The domain and range of a finite map, f , are denoted $\text{Dom } f$ and $\text{Ran } f$. A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \dots, a_k \mapsto b_k\}$, $k \geq 0$; in particular the empty map is $\{\}$. We shall use the form $\{x \mapsto e ; \phi\}$ – a form of set comprehension – to stand for the finite map f whose domain is the set of values x which satisfy the condition ϕ , and whose value on this domain is given by $f(x) = e$.

When f and g are finite maps the map $f + g$, called *f modified by g*, is the finite map with domain $\text{Dom } f \cup \text{Dom } g$ and values

$$(f + g)(a) = \text{if } a \in \text{Dom } g \text{ then } g(a) \text{ else } f(a).$$

The compound objects for the static semantics of the Core Language are shown in Figure 10. We take \cup to mean disjoint union over semantic object classes. We also understand all the defined object classes to be disjoint.

Note that Λ and \forall bind type variables. For any semantic object A , $\text{tynames } A$ and $\text{tyvars } A$ denote respectively the set of type names and the set of type variables occurring free in A .

Also note that a value environment maps value identifiers to a pair of a type scheme and an identifier status. If $VE(\text{vid}) = (\sigma, \text{is})$, we say that *vid has status is in VE*. An occurrence of a value identifier which is elaborated in VE is referred to as a *value variable*, a *value constructor* or an *exception constructor*, depending on whether its status in VE is \mathbf{v} , \mathbf{c} or \mathbf{e} , respectively.

4.3 Projection, Injection and Modification

Projection: We often need to select components of tuples – for example, the value-environment component of a context. In such cases we rely on metavariable names to indicate which component is selected. For instance “ VE of E ” means “the value-environment component of E ”.

Moreover, when a tuple contains a finite map we shall “apply” the tuple to an argument, relying on the syntactic class of the argument to determine the relevant function. For instance $C(\text{tycon})$ means $(TE \text{ of } C)\text{tycon}$ and $C(\text{vid})$ means $(VE \text{ of } (E \text{ of } C))(\text{vid})$.

$$\begin{aligned}
\tau &\in \text{Type} = \text{TyVar} \cup \text{RowType} \cup \text{FunType} \cup \text{ConsType} \\
(\tau_1, \dots, \tau_k) \text{ or } \tau^{(k)} &\in \text{Type}^k \\
(\alpha_1, \dots, \alpha_k) \text{ or } \alpha^{(k)} &\in \text{TyVar}^k \\
\varrho &\in \text{RowType} = \text{Lab} \xrightarrow{\text{fin}} \text{Type} \\
\tau \rightarrow \tau' &\in \text{FunType} = \text{Type} \times \text{Type} \\
&\text{ConsType} = \cup_{k \geq 0} \text{ConsType}^{(k)} \\
\tau^{(k)} t &\in \text{ConsType}^{(k)} = \text{Type}^k \times \text{TyName}^{(k)} \\
\theta \text{ or } \Lambda \alpha^{(k)}. \tau &\in \text{TypeFcn} = \cup_{k \geq 0} \text{TyVar}^k \times \text{Type} \\
\sigma \text{ or } \forall \alpha^{(k)}. \tau &\in \text{TypeScheme} = \cup_{k \geq 0} \text{TyVar}^k \times \text{Type} \\
(\theta, VE) &\in \text{TyStr} = \text{TypeFcn} \times \text{ValEnv} \\
SE &\in \text{StrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Env} \\
TE &\in \text{TyEnv} = \text{TyCon} \xrightarrow{\text{fin}} \text{TyStr} \\
VE &\in \text{ValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{TypeScheme} \times \text{IdStatus} \\
E \text{ or } (SE, TE, VE) &\in \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{ValEnv} \\
T &\in \text{TyNameSet} = \text{Fin}(\text{TyName}) \\
U &\in \text{TyVarSet} = \text{Fin}(\text{TyVar}) \\
C \text{ or } T, U, E &\in \text{Context} = \text{TyNameSet} \times \text{TyVarSet} \times \text{Env}
\end{aligned}$$

Figure 10: Compound Semantic Objects

Finally, environments may be applied to long identifiers. For instance if $longvid = strid_1 \dots strid_k.vid$ then $E(longvid)$ means

$$(VE \text{ of } (SE \text{ of } \dots (SE \text{ of } (SE \text{ of } E) strid_1) strid_2 \dots) strid_k) vid.$$

Injection: Components may be injected into tuple classes; for example, “ VE in Env ” means the environment $(\{\}, \{\}, VE)$.

Modification: The modification of one map f by another map g , written $f + g$, has already been mentioned. It is commonly used for environment modification, for example $E + E'$. Often, empty components will be left implicit in a modification; for example $E + VE$ means $E + (\{\}, \{\}, VE)$. For set components, modification means union, so that $C + (T, VE)$ means

$$((T \text{ of } C) \cup T, U \text{ of } C, (E \text{ of } C) + VE)$$

Finally, we frequently need to modify a context C by an environment E (or a type environment TE say), at the same time extending T of C to include the

type names of E (or of TE say). We therefore define $C \oplus TE$, for example, to mean $C + (\text{tynames } TE, TE)$.

4.4 Types and Type functions

A type τ is an *equality type*, or *admits equality*, if it is of one of the forms

- α , where α admits equality;
- $\{lab_1 \mapsto \tau_1, \dots, lab_n \mapsto \tau_n\}$, where each τ_i admits equality;
- $\tau^{(k)}t$, where t and all members of $\tau^{(k)}$ admit equality;
- $(\tau')\text{ref}$.

A type function $\theta = \Lambda\alpha^{(k)}. \tau$ has arity k ; the bound variables must be distinct. Two type functions are considered equal if they only differ in their choice of bound variables (alpha-conversion). In particular, the equality attribute has no significance in a bound variable of a type function; for example, $\Lambda\alpha.\alpha \rightarrow \alpha$ and $\Lambda\beta.\beta \rightarrow \beta$ are equal type functions even if α admits equality but β does not. If t has arity k , then we write t to mean $\Lambda\alpha^{(k)}. \alpha^{(k)}t$ (eta-conversion); thus $\text{TyName} \subseteq \text{TypeFcn}$. $\theta = \Lambda\alpha^{(k)}. \tau$ is an *equality type function*, or *admits equality*, if when the type variables $\alpha^{(k)}$ are chosen to admit equality then τ also admits equality.

We write the application of a type function θ to a vector $\tau^{(k)}$ of types as $\tau^{(k)}\theta$. If $\theta = \Lambda\alpha^{(k)}. \tau$ we set $\tau^{(k)}\theta = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ (beta-conversion).

We write $\tau\{\theta^{(k)}/t^{(k)}\}$ for the result of substituting type functions $\theta^{(k)}$ for type names $t^{(k)}$ in τ . We assume that all beta-conversions are carried out after substitution, so that for example

$$(\tau^{(k)}t)\{\Lambda\alpha^{(k)}. \tau/t\} = \tau\{\tau^{(k)}/\alpha^{(k)}\}.$$

4.5 Type Schemes

A type scheme $\sigma = \forall\alpha^{(k)}. \tau$ *generalises* a type τ' , written $\sigma \succ \tau'$, if $\tau' = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ for some $\tau^{(k)}$, where each member τ_i of $\tau^{(k)}$ admits equality if α_i does. If $\sigma' = \forall\beta^{(l)}. \tau'$ then σ *generalises* σ' , written $\sigma \succ \sigma'$, if $\sigma \succ \tau'$ and

$\beta^{(l)}$ contains no free type variable of σ . It can be shown that $\sigma \succ \sigma'$ iff, for all τ'' , whenever $\sigma' \succ \tau''$ then also $\sigma \succ \tau''$.

Two type schemes σ and σ' are considered equal if they can be obtained from each other by renaming and reordering of bound type variables, and deleting type variables from the prefix which do not occur in the body. Here, in contrast to the case for type functions, the equality attribute must be preserved in renaming; for example $\forall\alpha.\alpha \rightarrow \alpha$ and $\forall\beta.\beta \rightarrow \beta$ are only equal if either both α and β admit equality, or neither does. It can be shown that $\sigma = \sigma'$ iff $\sigma \succ \sigma'$ and $\sigma' \succ \sigma$.

We consider a type τ to be a type scheme, identifying it with $\forall().\tau$.

4.6 Scope of Explicit Type Variables

In the Core language, a type or datatype binding can explicitly introduce type variables whose scope is that binding. Moreover, in a value declaration `val tyvarseq valbind`, the sequence *tyvarseq* binds type variables: a type variable occurs free in `val tyvarseq valbind` iff it occurs free in *valbind* and is not in the sequence *tyvarseq*. However, explicit binding of type variables at `val` is optional, so we still have to account for the scope of an explicit type variable occurring in the “: *ty*” of a typed expression or pattern or in the “of *ty*” of an exception binding. For the rest of this section, we consider such free occurrences of type variables only.

Every occurrence of a value declaration is said to *scope* a set of explicit type variables determined as follows.

First, a free occurrence of α in a value declaration `val tyvarseq valbind` is said to be *unguarded* if the occurrence is not part of a smaller value declaration within *valbind*. In this case we say that α *occurs unguarded* in the value declaration.

Then we say that α is *implicitly scoped at* a particular value declaration `val tyvarseq valbind` in a program if (1) α occurs unguarded in this value declaration, and (2) α does not occur unguarded in any larger value declaration containing the given one.

Henceforth, we assume that for every value declaration `val tyvarseq...` occurring in the program, every explicit type variable implicitly scoped at the `val` has been added to *tyvarseq* (subject to the syntactic constraint in Section 2.9). Thus for example, in the two declarations

```
val x = let val id:'a->'a = fn z=>z in id id end
```

```
val x = (let val id:'a->'a = fn z=>z in id id end; fn z=>z:'a)
```

the type variable 'a is scoped differently; they become respectively

```
val x = let val 'a id:'a->'a = fn z=>z in id id end
val 'a x = (let val id:'a->'a = fn z=>z in id id end; fn z=>z:'a)
```

Then, according to the inference rules in Section 4.10 the first example can be elaborated, but the second cannot since 'a is bound at the outer value declaration leaving no possibility of two different instantiations of the type of `id` in the application `id id`.

4.7 Non-expansive Expressions

In order to treat polymorphic references and exceptions, the set `Exp` of expressions is partitioned into two classes, the *expansive* and the *non-expansive* expressions. An expression is *non-expansive in context C* if, after replacing infix forms by their equivalent prefixed forms, and derived forms by their equivalent forms, it can be generated by the following grammar from the non-terminal *nexp*:

$$\begin{array}{ll}
 nexp ::= scon & nexprow ::= lab = nexp\langle, nexprow\rangle \\
 \langle op \rangle longvid & \\
 \{ \langle nexprow \rangle \} & conexp ::= (conexp\langle : ty \rangle) \\
 (nexp) & \langle op \rangle longvid \\
 conexp nexp & \\
 nexp : ty & \\
 fn match &
 \end{array}$$

Restriction: Within a *conexp*, we require *longvid* \neq `ref` and *is of* $C(longvid) \in \{c, e\}$.

All other expressions are said to be *expansive (in C)*. The idea is that the dynamic evaluation of a non-expansive expression will neither generate an exception nor extend the domain of the memory, while the evaluation of an expansive expression might.

4.8 Closure

Let τ be a type and A a semantic object. Then $\text{Clos}_A(\tau)$, the *closure* of τ with respect to A , is the type scheme $\forall \alpha^{(k)}. \tau$, where $\alpha^{(k)} = \text{tyvars}(\tau) \setminus \text{tyvars } A$.

Commonly, A will be a context C . We abbreviate the *total* closure $\text{Clos}_{\{\}}(\tau)$ to $\text{Clos}(\tau)$. If the range of a value environment VE contains only types (rather than arbitrary type schemes) we set

$$\text{Clos}_A VE = \{vid \mapsto (\text{Clos}_A(\tau), is) ; VE(vid) = (\tau, is)\}$$

Closing a value environment VE that stems from the elaboration of a value binding $valbind$ requires extra care to ensure type security of references and exceptions and correct scoping of explicit type variables. Recall that $valbind$ is not allowed to bind the same variable twice. Thus, for each $vid \in \text{Dom } VE$ there is a unique $pat = exp$ in $valbind$ which binds vid . If $VE(vid) = (\tau, is)$, let $\text{Clos}_{C, valbind} VE(vid) = (\forall \alpha^{(k)}. \tau, is)$, where

$$\alpha^{(k)} = \begin{cases} \text{tyvars } \tau \setminus \text{tyvars } C, & \text{if } exp \text{ is non-expansive in } C; \\ (), & \text{if } exp \text{ is expansive in } C. \end{cases}$$

4.9 Type Structures and Type Environments

A type structure (θ, VE) is *well-formed* if either $VE = \{\}$, or θ is a type name t . (The latter case arises, with $VE \neq \{\}$, in `datatype` declarations.) An object or assembly A of semantic objects is *well-formed* if every type structure occurring in A is well-formed.

A type structure (t, VE) is said to *respect equality* if, whenever t admits equality, then either $t = \mathbf{ref}$ (see Appendix C) or, for each $VE(vid)$ of the form $(\forall \alpha^{(k)}. (\tau \rightarrow \alpha^{(k)} t), is)$, the type function $\Lambda \alpha^{(k)}. \tau$ also admits equality. (This ensures that the equality predicate $=$ will be applicable to a constructed value (vid, v) of type $\tau^{(k)} t$ only when it is applicable to the value v itself, whose type is $\tau\{\tau^{(k)}/\alpha^{(k)}\}$.) A type environment TE *respects equality* if all its type structures do so.

Let TE be a type environment, and let T be the set of type names t such that (t, VE) occurs in TE for some $VE \neq \{\}$. Then TE is said to *maximise equality* if (a) TE respects equality, and also (b) if any larger subset of T were to admit equality (without any change in the equality attribute of any type names not in T) then TE would cease to respect equality.

For any TE of the form

$$TE = \{tycon_i \mapsto (t_i, VE_i) ; 1 \leq i \leq k\},$$

where no VE_i is the empty map, and for any E we define $\text{Abs}(TE, E)$ to be the environment obtained from E and TE as follows. First, let $\text{Abs}(TE)$

be the type environment $\{tycon_i \mapsto (t_i, \{\}) ; 1 \leq i \leq k\}$ in which all value environments VE_i have been replaced by the empty map. Let t'_1, \dots, t'_k be new distinct type names none of which admit equality. Then $\text{Abs}(TE, E)$ is the result of simultaneously substituting t'_i for t_i , $1 \leq i \leq k$, throughout $\text{Abs}(TE) + E$. (The effect of the latter substitution is to ensure that the use of equality on an **abstype** is restricted to the **with** part.)

4.10 Inference Rules

Each rule of the semantics allows inferences among sentences of the form

$$A \vdash \textit{phrase} \Rightarrow A'$$

where A is usually a context, *phrase* is a phrase of the Core, and A' is a semantic object – usually a type or an environment. It may be pronounced “*phrase* elaborates to A' in (context) A ”. Some rules have extra hypotheses not of this form; they are called *side conditions*.

In the presentation of the rules, phrases within single angle brackets $\langle \rangle$ are called *first options*, and those within double angle brackets $\langle\langle \rangle\rangle$ are called *second options*. To reduce the number of rules, we have adopted the following convention:

In each instance of a rule, the first options must be either all present or all absent; similarly the second options must be either all present or all absent.

Although not assumed in our definitions, it is intended that every context $C = T, U, E$ has the property that tynames $E \subseteq T$. Thus T may be thought of, loosely, as containing all type names which “have been generated”. It is necessary to include T as a separate component in a context, since tynames E may not contain all the type names which have been generated; one reason is that a context T, \emptyset, E is a projection of the basis $B = T, F, G, E$ whose other components F and G could contain other such names – recorded in T but not present in E . Of course, remarks about what “has been generated” are not precise in terms of the semantic rules. But the following precise result may easily be demonstrated:

Let S be a sentence $T, U, E \vdash \textit{phrase} \Rightarrow A$ such that tynames $E \subseteq T$, and let S' be a sentence $T', U', E' \vdash \textit{phrase}' \Rightarrow A'$ occurring in a proof of S ; then also tynames $E' \subseteq T'$.

Atomic Expressions

$$\boxed{C \vdash atexp \Rightarrow \tau}$$

$$\frac{}{C \vdash scon \Rightarrow \text{type}(scon)} \quad (1)$$

$$\frac{C(\text{longvid}) = (\sigma, is) \quad \sigma \succ \tau}{C \vdash \text{longvid} \Rightarrow \tau} \quad (2)$$

$$\frac{\langle C \vdash \text{exprow} \Rightarrow \varrho \rangle}{C \vdash \{ \langle \text{exprow} \rangle \} \Rightarrow \{ \} \langle + \varrho \rangle \text{ in Type}} \quad (3)$$

$$\frac{C \vdash \text{dec} \Rightarrow E \quad C \oplus E \vdash \text{exp} \Rightarrow \tau \quad \text{tynames } \tau \subseteq T \text{ of } C}{C \vdash \text{let } \text{dec} \text{ in } \text{exp} \text{ end} \Rightarrow \tau} \quad (4)$$

$$\frac{C \vdash \text{exp} \Rightarrow \tau}{C \vdash (\text{exp}) \Rightarrow \tau} \quad (5)$$

Comments:

- (2) The instantiation of type schemes allows different occurrences of a single *longvid* to assume different types. Note that the identifier status is not used in this rule.
- (4) The use of \oplus , here and elsewhere, ensures that type names generated by the first sub-phrase are different from type names generated by the second sub-phrase. The side condition prevents type names generated by *dec* from escaping outside the local declaration.

Expression Rows

$$\boxed{C \vdash \text{exprow} \Rightarrow \varrho}$$

$$\frac{C \vdash \text{exp} \Rightarrow \tau \quad \langle C \vdash \text{exprow} \Rightarrow \varrho \rangle}{C \vdash \text{lab} = \text{exp} \langle , \text{exprow} \rangle \Rightarrow \{ \text{lab} \mapsto \tau \} \langle + \varrho \rangle} \quad (6)$$

Expressions

$$\boxed{C \vdash \text{exp} \Rightarrow \tau}$$

$$\frac{C \vdash atexp \Rightarrow \tau}{C \vdash atexp \Rightarrow \tau} \quad (7)$$

$$\frac{C \vdash \text{exp} \Rightarrow \tau' \rightarrow \tau \quad C \vdash atexp \Rightarrow \tau'}{C \vdash \text{exp} atexp \Rightarrow \tau} \quad (8)$$

$$\frac{C \vdash \text{exp} \Rightarrow \tau \quad C \vdash \text{ty} \Rightarrow \tau}{C \vdash \text{exp} : \text{ty} \Rightarrow \tau} \quad (9)$$

$$\frac{C \vdash \text{exp} \Rightarrow \tau \quad C \vdash \text{match} \Rightarrow \text{exn} \rightarrow \tau}{C \vdash \text{exp handle match} \Rightarrow \tau} \quad (10)$$

$$\frac{C \vdash \text{exp} \Rightarrow \text{exn}}{C \vdash \text{raise exp} \Rightarrow \tau} \quad (11)$$

$$\frac{C \vdash \text{match} \Rightarrow \tau}{C \vdash \text{fn match} \Rightarrow \tau} \quad (12)$$

Comments:

- (7) The relational symbol \vdash is overloaded for all syntactic classes (here atomic expressions and expressions).
- (9) Here τ is determined by C and ty . Notice that type variables in ty cannot be instantiated in obtaining τ ; thus the expression $1 : 'a$ will not elaborate successfully, nor will the expression $(\text{fn } x \Rightarrow x) : 'a \rightarrow 'b$. The effect of type variables in an explicitly typed expression is to indicate exactly the degree of polymorphism present in the expression.
- (11) Note that τ does not occur in the premise; thus a **raise** expression has “arbitrary” type.

Matches

$$\boxed{C \vdash \text{match} \Rightarrow \tau}$$

$$\frac{C \vdash \text{mrule} \Rightarrow \tau \quad \langle C \vdash \text{match} \Rightarrow \tau \rangle}{C \vdash \text{mrule} \langle | \text{match} \rangle \Rightarrow \tau} \quad (13)$$

Match Rules

$$\boxed{C \vdash \text{mrule} \Rightarrow \tau}$$

$$\frac{C \vdash \text{pat} \Rightarrow (VE, \tau) \quad C + VE \vdash \text{exp} \Rightarrow \tau' \quad \text{tynames } VE \subseteq T \text{ of } C}{C \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \tau \rightarrow \tau'} \quad (14)$$

Comment: This rule allows new free type variables to enter the context. These new type variables will be chosen, in effect, during the elaboration of pat (i.e., in the inference of the first hypothesis). In particular, their choice may have to be made to agree with type variables present in any explicit type expression occurring within exp (see rule 9).

Declarations

$$\boxed{C \vdash dec \Rightarrow E}$$

$$\frac{U = \text{tyvars}(\text{tyvarseq}) \quad C + U \vdash \text{valbind} \Rightarrow VE \quad VE' = \text{Clos}_{C, \text{valbind}} VE \quad U \cap \text{tyvars} VE' = \emptyset}{C \vdash \text{val } \text{tyvarseq } \text{valbind} \Rightarrow VE' \text{ in Env}} \quad (15)$$

$$\frac{C \vdash \text{typbind} \Rightarrow TE}{C \vdash \text{type } \text{typbind} \Rightarrow TE \text{ in Env}} \quad (16)$$

$$\frac{C \oplus TE \vdash \text{datbind} \Rightarrow VE, TE \quad \forall (t, VE') \in \text{Ran } TE, t \notin (T \text{ of } C) \quad TE \text{ maximises equality}}{C \vdash \text{datatype } \text{datbind} \Rightarrow (VE, TE) \text{ in Env}} \quad (17)$$

$$\frac{C(\text{longtycon}) = (\theta, VE) \quad TE = \{\text{tycon} \mapsto (\theta, VE)\}}{C \vdash \text{datatype } \text{tycon} \text{ --- datatype } \text{longtycon} \Rightarrow (VE, TE) \text{ in Env}} \quad (18)$$

$$\frac{C \oplus TE \vdash \text{datbind} \Rightarrow VE, TE \quad \forall (t, VE') \in \text{Ran } TE, t \notin (T \text{ of } C) \quad C \oplus (VE, TE) \vdash dec \Rightarrow E \quad TE \text{ maximises equality}}{C \vdash \text{abstype } \text{datbind with } dec \text{ end} \Rightarrow \text{Abs}(TE, E)} \quad (19)$$

$$\frac{C \vdash \text{exbind} \Rightarrow VE}{C \vdash \text{exception } \text{exbind} \Rightarrow VE \text{ in Env}} \quad (20)$$

$$\frac{C \vdash dec_1 \Rightarrow E_1 \quad C \oplus E_1 \vdash dec_2 \Rightarrow E_2}{C \vdash \text{local } dec_1 \text{ in } dec_2 \text{ end} \Rightarrow E_2} \quad (21)$$

$$\frac{C(\text{longstrid}_1) = E_1 \quad \dots \quad C(\text{longstrid}_n) = E_n}{C \vdash \text{open } \text{longstrid}_1 \dots \text{longstrid}_n \Rightarrow E_1 + \dots + E_n} \quad (22)$$

$$\overline{C \vdash \quad \Rightarrow \{\}} \text{ in Env} \quad (23)$$

$$\frac{C \vdash dec_1 \Rightarrow E_1 \quad C \oplus E_1 \vdash dec_2 \Rightarrow E_2}{C \vdash dec_1 \langle ; \rangle dec_2 \Rightarrow E_1 + E_2} \quad (24)$$

Comments:

- (15) Here VE will contain types rather than general type schemes. The closure of VE allows value identifiers to be used polymorphically, via rule 2.

The side-condition on U ensures that the type variables in $tyvarseq$ are bound by the closure operation, if they occur free in the range of VE .

On the other hand, if the phrase `val tyvarseq valbind` occurs inside some larger value binding `val tyvarseq' valbind'` then no type variable α listed in $tyvarseq'$ will become bound by the $\text{Clos}_{C, \text{valbind}} VE$ operation; for α must be in U of C and hence excluded from closure by the definition of the closure operation (Section 4.8, page 22) since U of $C \subseteq \text{tyvars } C$.

- (17),(19) The side conditions express that the elaboration of each datatype binding generates new type names and that as many of these new names as possible admit equality. Adding TE to the context on the left of the \vdash captures the recursive nature of the binding.
- (18) Note that no new type name is generated (i.e., datatype replication is not generative).
- (19) The Abs operation was defined in Section 4.9, page 23.
- (20) No closure operation is used here, as this would make the type system unsound. Example: `exception E of 'a; val it = (raise E 5) handle E f => f(2)` .

Value Bindings

$$\boxed{C \vdash \text{valbind} \Rightarrow VE}$$

$$\frac{C \vdash \text{pat} \Rightarrow (VE, \tau) \quad C \vdash \text{exp} \Rightarrow \tau \quad \langle C \vdash \text{valbind} \Rightarrow VE' \rangle}{C \vdash \text{pat} = \text{exp} \langle \text{and valbind} \rangle \Rightarrow VE \langle + VE' \rangle} \quad (25)$$

$$\frac{C + VE \vdash \text{valbind} \Rightarrow VE \quad \text{tynames } VE \subseteq T \text{ of } C}{C \vdash \text{rec valbind} \Rightarrow VE} \quad (26)$$

Comments:

- (25) When the option is present we have $\text{Dom } VE \cap \text{Dom } VE' = \emptyset$ by the syntactic restrictions.
- (26) Modifying C by VE on the left captures the recursive nature of the binding. From rule 25 we see that any type scheme occurring in VE

will have to be a type. Thus each use of a recursive function in its own body must be assigned the same type. Also note that $C + VE$ may overwrite identifier status. For example, the program `datatype t = f; val rec f = fn x => x;` is legal.

Type Bindings

$$\boxed{C \vdash \text{typbind} \Rightarrow TE}$$

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad C \vdash ty \Rightarrow \tau \quad \langle C \vdash \text{typbind} \Rightarrow TE \rangle}{C \vdash \text{tyvarseq tycon} = ty \langle \text{and typbind} \rangle \Rightarrow \{ \text{tycon} \mapsto (\Lambda \alpha^{(k)}. \tau, \{ \}) \} \langle + TE \rangle} \quad (27)$$

Comment: The syntactic restrictions ensure that the type function $\Lambda \alpha^{(k)}. \tau$ satisfies the well-formedness constraint of Section 4.4 and they ensure $\text{tycon} \notin \text{Dom } TE$.

Datatype Bindings

$$\boxed{C \vdash \text{datbind} \Rightarrow VE, TE}$$

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad C, \alpha^{(k)} t \vdash \text{conbind} \Rightarrow VE \quad \text{arity } t = k \quad \langle C \vdash \text{datbind}' \Rightarrow VE', TE' \quad \forall (t', VE'') \in \text{Ran } TE', t \neq t' \rangle}{C \vdash \text{tyvarseq tycon} = \text{conbind} \langle \text{and datbind}' \rangle \Rightarrow (\text{Clos } VE \langle + VE' \rangle, \{ \text{tycon} \mapsto (t, \text{Clos } VE) \}) \langle + TE' \rangle} \quad (28)$$

Comment: The syntactic restrictions ensure $\text{Dom } VE \cap \text{Dom } VE' = \emptyset$ and $\text{tycon} \notin \text{Dom } TE'$.

Constructor Bindings

$$\boxed{C, \tau \vdash \text{conbind} \Rightarrow VE}$$

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \quad \langle \langle C, \tau \vdash \text{conbind} \Rightarrow VE \rangle \rangle}{C, \tau \vdash \text{vid} \langle \text{of } ty \rangle \langle \langle \mid \text{conbind} \rangle \rangle \Rightarrow \{ \text{vid} \mapsto (\tau, \mathbf{c}) \} \langle + \{ \text{vid} \mapsto (\tau' \rightarrow \tau, \mathbf{c}) \} \rangle \langle \langle + VE \rangle \rangle} \quad (29)$$

Comment: By the syntactic restrictions $\text{vid} \notin \text{Dom } VE$.

Exception Bindings

$$\boxed{C \vdash \text{exbind} \Rightarrow VE}$$

$$\frac{\langle C \vdash ty \Rightarrow \tau \rangle \quad \langle \langle C \vdash \text{exbind} \Rightarrow VE \rangle \rangle}{C \vdash \text{vid} \langle \text{of } ty \rangle \langle \langle \text{and exbind} \rangle \rangle \Rightarrow \{ \text{vid} \mapsto (\mathbf{exn}, \mathbf{e}) \} \langle + \{ \text{vid} \mapsto (\tau \rightarrow \mathbf{exn}, \mathbf{e}) \} \rangle \langle \langle + VE \rangle \rangle} \quad (30)$$

$$\frac{C(\text{longvid}) = (\tau, \mathbf{e}) \quad \langle C \vdash \text{exbind} \Rightarrow VE \rangle}{C \vdash \text{vid} = \text{longvid} \langle \text{and exbind} \rangle \Rightarrow \{ \text{vid} \mapsto (\tau, \mathbf{e}) \} \langle + VE \rangle} \quad (31)$$

Comments:

(30) Notice that τ may contain type variables.

(30),(31) For each C and exbind , there is at most one VE satisfying $C \vdash \text{exbind} \Rightarrow VE$.

Atomic Patterns

$$\boxed{C \vdash \text{atpat} \Rightarrow (VE, \tau)}$$

$$\overline{C \vdash _ \Rightarrow (\{\}, \tau)} \quad (32)$$

$$\overline{C \vdash \text{scon} \Rightarrow (\{\}, \text{type}(\text{scon}))} \quad (33)$$

$$\frac{\text{vid} \notin \text{Dom}(C) \text{ or is of } C(\text{vid}) = \mathbf{v}}{C \vdash \text{vid} \Rightarrow (\{ \text{vid} \mapsto (\tau, \mathbf{v}) \}, \tau)} \quad (34)$$

$$\frac{C(\text{longvid}) = (\sigma, \text{is}) \quad \text{is} \neq \mathbf{v} \quad \sigma \succ \tau^{(k)}t}{C \vdash \text{longvid} \Rightarrow (\{\}, \tau^{(k)}t)} \quad (35)$$

$$\frac{\langle C \vdash \text{patrow} \Rightarrow (VE, \varrho) \rangle}{C \vdash \{ \langle \text{patrow} \rangle \} \Rightarrow (\{ \} \langle + VE \rangle, \{ \} \langle + \varrho \rangle \text{ in Type})} \quad (36)$$

$$\frac{C \vdash \text{pat} \Rightarrow (VE, \tau)}{C \vdash (\text{pat}) \Rightarrow (VE, \tau)} \quad (37)$$

Comments:

(34), (35) The context C determines which of these two rules applies. In rule 34, note that vid can assume a type, not a general type scheme.

Pattern Rows

$$\boxed{C \vdash \text{patrow} \Rightarrow (VE, \varrho)}$$

$$\overline{C \vdash \dots \Rightarrow (\{\}, \varrho)} \quad (38)$$

$$\frac{C \vdash \text{pat} \Rightarrow (VE, \tau) \quad \langle C \vdash \text{patrow} \Rightarrow (VE', \varrho) \quad \text{Dom } VE \cap \text{Dom } VE' = \emptyset \rangle}{C \vdash \text{lab} = \text{pat} \langle _ , \text{patrow} \rangle \Rightarrow (VE \langle + VE' \rangle, \{ \text{lab} \mapsto \tau \} \langle + \varrho \rangle)} \quad (39)$$

Comment:

(39) The syntactic restrictions ensure $lab \notin \text{Dom } \varrho$.

Patterns

$$\boxed{C \vdash pat \Rightarrow (VE, \tau)}$$

$$\frac{C \vdash atpat \Rightarrow (VE, \tau)}{C \vdash atpat \Rightarrow (VE, \tau)} \quad (40)$$

$$\frac{C(\text{longvid}) = (\sigma, is) \quad is \neq \mathbf{v} \quad \sigma \succ \tau' \rightarrow \tau \quad C \vdash atpat \Rightarrow (VE, \tau')}{C \vdash \text{longvid } atpat \Rightarrow (VE, \tau)} \quad (41)$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \quad C \vdash ty \Rightarrow \tau}{C \vdash pat : ty \Rightarrow (VE, \tau)} \quad (42)$$

$$\frac{\begin{array}{c} vid \notin \text{Dom}(C) \text{ or } is \text{ of } C(vid) = \mathbf{v} \\ \langle C \vdash ty \Rightarrow \tau \rangle \quad C \vdash pat \Rightarrow (VE, \tau) \quad vid \notin \text{Dom } VE \end{array}}{C \vdash vid(: ty) \text{ as } pat \Rightarrow (\{vid \mapsto (\tau, \mathbf{v})\} + VE, \tau)} \quad (43)$$

Type Expressions

$$\boxed{C \vdash ty \Rightarrow \tau}$$

$$\frac{tyvar = \alpha}{C \vdash tyvar \Rightarrow \alpha} \quad (44)$$

$$\frac{\langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash \{ \langle tyrow \rangle \} \Rightarrow \{ \} \langle + \varrho \rangle \text{ in Type}} \quad (45)$$

$$\frac{\begin{array}{c} tyseq = ty_1 \cdots ty_k \quad C \vdash ty_i \Rightarrow \tau_i \ (1 \leq i \leq k) \\ C(\text{longtycon}) = (\theta, VE) \end{array}}{C \vdash tyseq \text{ longtycon} \Rightarrow \tau^{(k)}\theta} \quad (46)$$

$$\frac{C \vdash ty \Rightarrow \tau \quad C \vdash ty' \Rightarrow \tau'}{C \vdash ty \rightarrow ty' \Rightarrow \tau \rightarrow \tau'} \quad (47)$$

$$\frac{C \vdash ty \Rightarrow \tau}{C \vdash (ty) \Rightarrow \tau} \quad (48)$$

Comments:

(46) Recall that for $\tau^{(k)}\theta$ to be defined, θ must have arity k .

Type-expression Rows

$$\boxed{C \vdash tyrow \Rightarrow \varrho}$$

$$\frac{C \vdash ty \Rightarrow \tau \quad \langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash lab : ty \langle , tyrow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \varrho \rangle} \quad (49)$$

Comment: The syntactic constraints ensure $lab \notin \text{Dom } \varrho$.

4.11 Further Restrictions

There are a few restrictions on programs which should be enforced by a compiler, but are better expressed apart from the preceding Inference Rules. They are:

1. For each occurrence of a record pattern containing a record wildcard, i.e. of the form $\{lab_1=pat_1, \dots, lab_m=pat_m, \dots\}$ the program context must determine uniquely the domain $\{lab_1, \dots, lab_n\}$ of its row type, where $m \leq n$; thus, the context must determine the labels $\{lab_{m+1}, \dots, lab_n\}$ of the fields to be matched by the wildcard. For this purpose, an explicit type constraint may be needed.
2. In a match of the form $pat_1 \Rightarrow exp_1 \mid \dots \mid pat_n \Rightarrow exp_n$ the pattern sequence pat_1, \dots, pat_n should be *irredundant*; that is, each pat_j must match some value (of the right type) which is not matched by pat_i for any $i < j$. In the context $\text{fn } match$, the *match* must also be *exhaustive*; that is, every value (of the right type) must be matched by some pat_i . The compiler must give warning on violation of these restrictions, but should still compile the match. The restrictions are inherited by derived forms; in particular, this means that in the function-value binding $\text{vid } atpat_1 \dots atpat_n \langle : ty \rangle = exp$ (consisting of one clause only), each separate $atpat_i$ should be exhaustive by itself.
3. For each value binding $pat = exp$ the compiler must issue a report (but still compile) if pat is not exhaustive. This will detect a mistaken declaration like $\text{val nil} = exp$ in which the user expects to declare a new variable `nil` (whereas the language dictates that `nil` is here a constant pattern, so no variable gets declared). However, this warning should not be given when the binding is a component of a top-level declaration $\text{val } valbind$; e.g. $\text{val } x :: l = exp_1 \text{ and } y = exp_2$ is not faulted by the compiler at top level, but may of course generate a `Bind` exception (see Section 6.5).

5 Static Semantics for Modules

5.1 Semantic Objects

The simple objects for Modules static semantics are exactly as for the Core. The compound objects are those for the Core, augmented by those in Figure 11.

$$\begin{aligned}
 \Sigma \text{ or } (T)E &\in \text{Sig} = \text{TyNameSet} \times \text{Env} \\
 \Phi \text{ or } (T)(E, (T')E') &\in \text{FunSig} = \text{TyNameSet} \times (\text{Env} \times \text{Sig}) \\
 G &\in \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Sig} \\
 F &\in \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunSig} \\
 B \text{ or } T, F, G, E &\in \text{Basis} = \text{TyNameSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env}
 \end{aligned}$$

Figure 11: Further Compound Semantic Objects

The prefix (T) , in signatures and functor signatures, binds type names. Certain operations require a change of bound names in semantic objects; see for example Section 5.2. When bound type names are changed, we demand that all of their attributes (i.e. equality and arity) are preserved.

The operations of projection, injection and modification are as for the Core. Moreover, we define C of B to be the context $(T$ of B, \emptyset, E of $B)$, i.e. with an empty set of explicit type variables. Also, we frequently need to modify a basis B by an environment E (or a structure environment SE say), at the same time extending T of B to include the type names of E (or of SE say). We therefore define $B \oplus SE$, for example, to mean $B + (\text{tynames } SE, SE)$.

There is no separate kind of semantic object to represent structures: structure expressions elaborate to environments, just as structure-level declarations do. Thus, notions which are commonly associated with structures (for example the notion of matching a structure against a signature) are defined in terms of environments.

5.2 Type Realisation

A *(type) realisation* is a map $\varphi : \text{TyName} \rightarrow \text{TypeFcn}$ such that t and $\varphi(t)$ have the same arity, and if t admits equality then so does $\varphi(t)$.

The *support* $\text{Supp } \varphi$ of a type realisation φ is the set of type names t for which $\varphi(t) \neq t$. The *yield* $\text{Yield } \varphi$ of a realisation φ is the set of type names which occur in some $\varphi(t)$ for which $t \in \text{Supp } \varphi$.

Realisations φ are extended to apply to all semantic objects; their effect is to replace each name t by $\varphi(t)$. In applying φ to an object with bound names, such as a signature $(T)E$, first bound names must be changed so that, for each binding prefix (T) ,

$$T \cap (\text{Supp } \varphi \cup \text{Yield } \varphi) = \emptyset .$$

5.3 Signature Instantiation

An environment E_2 is an *instance* of a signature $\Sigma_1 = (T_1)E_1$, written $\Sigma_1 \geq E_2$, if there exists a realisation φ such that $\varphi(E_1) = E_2$ and $\text{Supp } \varphi \subseteq T_1$.

5.4 Functor Signature Instantiation

A pair $(E, (T')E')$ is called a *functor instance*. Given $\Phi = (T_1)(E_1, (T'_1)E'_1)$, a functor instance $(E_2, (T'_2)E'_2)$ is an *instance* of Φ , written $\Phi \geq (E_2, (T'_2)E'_2)$, if there exists a realisation φ such that $\varphi(E_1, (T'_1)E'_1) = (E_2, (T'_2)E'_2)$ and $\text{Supp } \varphi \subseteq T_1$.

5.5 Enrichment

In matching an environment to a signature, the environment will be allowed both to have more components, and to be more polymorphic, than (an instance of) the signature. Precisely, we define enrichment of environments and type structures recursively as follows.

An environment $E_1 = (SE_1, TE_1, VE_1)$ *enriches* another environment $E_2 = (SE_2, TE_2, VE_2)$, written $E_1 \succ E_2$, if

1. $\text{Dom } SE_1 \supseteq \text{Dom } SE_2$, and $SE_1(\text{strid}) \succ SE_2(\text{strid})$ for all $\text{strid} \in \text{Dom } SE_2$
2. $\text{Dom } TE_1 \supseteq \text{Dom } TE_2$, and $TE_1(\text{tycon}) \succ TE_2(\text{tycon})$ for all $\text{tycon} \in \text{Dom } TE_2$

3. $\text{Dom } VE_1 \supseteq \text{Dom } VE_2$, and $VE_1(vid) \succ VE_2(vid)$ for all $vid \in \text{Dom } VE_2$, where $(\sigma_1, is_1) \succ (\sigma_2, is_2)$ means $\sigma_1 \succ \sigma_2$ and

$$is_1 = is_2 \quad \text{or} \quad is_2 = \mathbf{v}$$

Finally, a type structure (θ_1, VE_1) enriches another type structure (θ_2, VE_2) , written $(\theta_1, VE_1) \succ (\theta_2, VE_2)$, if

1. $\theta_1 = \theta_2$
2. Either $VE_1 = VE_2$ or $VE_2 = \{\}$

5.6 Signature Matching

An environment E matches a signature Σ_1 if there exists an environment E^- such that $\Sigma_1 \geq E^- \prec E$. Thus matching is a combination of instantiation and enrichment. There is at most one such E^- , given Σ_1 and E .

5.7 Inference Rules

As for the Core, the rules of the Modules static semantics allow sentences of the form

$$A \vdash \textit{phrase} \Rightarrow A'$$

to be inferred, where in this case A is either a basis, a context or an environment and A' is a semantic object. The convention for options is as in the Core semantics.

Although not assumed in our definitions, it is intended that every basis $B = T, F, G, E$ in which a *topdec* is elaborated has the property that $\text{tynames } F \cup \text{tynames } G \cup \text{tynames } E \subseteq T$. The following Theorem can be proved:

Let S be an inferred sentence $B \vdash \textit{topdec} \Rightarrow B'$ in which B satisfies the above condition. Then B' also satisfies the condition.

Moreover, if S' is a sentence of the form $B'' \vdash \textit{phrase} \Rightarrow A$ occurring in a proof of S , where *phrase* is any Modules phrase, then B'' also satisfies the condition.

Finally, if $T, U, E \vdash \textit{phrase} \Rightarrow A$ occurs in a proof of S , where *phrase* is a phrase of Modules or of the Core, then $\text{tynames } E \subseteq T$.

Structure Expressions

$B \vdash \text{strex} \Rightarrow E$

$$\frac{B \vdash \text{strdec} \Rightarrow E}{B \vdash \text{struct strdec end} \Rightarrow \bar{E}} \quad (50)$$

$$\frac{B(\text{longstrid}) = E}{B \vdash \text{longstrid} \Rightarrow E} \quad (51)$$

$$\frac{B \vdash \text{strex} \Rightarrow E \quad B \vdash \text{sigexp} \Rightarrow \Sigma \quad \Sigma \geq E' \prec E}{B \vdash \text{strex} : \text{sigexp} \Rightarrow E'} \quad (52)$$

$$\frac{B \vdash \text{strex} \Rightarrow E \quad B \vdash \text{sigexp} \Rightarrow (T')E' \quad (T')E' \geq E'' \prec E \quad T' \cap (T \text{ of } B) = \emptyset}{B \vdash \text{strex} : > \text{sigexp} \Rightarrow E'} \quad (53)$$

$$\frac{B \vdash \text{strex} \Rightarrow E \quad B(\text{funid}) \geq (E'', (T')E'), E \succ E'' \quad (\text{tynames } E \cup T \text{ of } B) \cap T' = \emptyset}{B \vdash \text{funid} (\text{strex}) \Rightarrow E'} \quad (54)$$

$$\frac{B \vdash \text{strdec} \Rightarrow E_1 \quad B \oplus E_1 \vdash \text{strex} \Rightarrow E_2}{B \vdash \text{let strdec in strex end} \Rightarrow E_2} \quad (55)$$

Comments:

(54) The side condition $(\text{tynames } E \cup T \text{ of } B) \cap T' = \emptyset$ can always be satisfied by renaming bound names in $(T')E'$; it ensures that the generated datatypes receive new names.

Let $B(\text{funid}) = (T)(E_f, (T')E'_f)$. Let φ be a realisation such that $\varphi(E_f, (T')E'_f) = (E'', (T')E')$. Sharing between argument and result specified in the declaration of the functor *funid* is represented by the occurrence of the same name in both E_f and E'_f , and this repeated occurrence is preserved by φ , yielding sharing between the argument structure E and the result structure E' of this functor application.

(55) The use of \oplus , here and elsewhere, ensures that type names generated by the first sub-phrase are distinct from names generated by the second sub-phrase.

Structure-level Declarations

$$\boxed{B \vdash \text{strdec} \Rightarrow E}$$

$$\frac{C \text{ of } B \vdash \text{dec} \Rightarrow E}{B \vdash \text{dec} \Rightarrow E} \quad (56)$$

$$\frac{B \vdash \text{strbind} \Rightarrow SE}{B \vdash \text{structure strbind} \Rightarrow SE \text{ in Env}} \quad (57)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B \oplus E_1 \vdash \text{strdec}_2 \Rightarrow E_2}{B \vdash \text{local strdec}_1 \text{ in strdec}_2 \text{ end} \Rightarrow E_2} \quad (58)$$

$$\frac{}{B \vdash \quad \Rightarrow \{\} \text{ in Env}} \quad (59)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B \oplus E_1 \vdash \text{strdec}_2 \Rightarrow E_2}{B \vdash \text{strdec}_1 \langle ; \rangle \text{strdec}_2 \Rightarrow E_1 + E_2} \quad (60)$$

Structure Bindings

$$\boxed{B \vdash \text{strbind} \Rightarrow SE}$$

$$\frac{B \vdash \text{strexpr} \Rightarrow E \quad \langle B + \text{tynames } E \vdash \text{strbind} \Rightarrow SE \rangle}{B \vdash \text{strid} = \text{strexpr} \langle \text{and strbind} \rangle \Rightarrow \{\text{strid} \mapsto E\} \langle + SE \rangle} \quad (61)$$

Signature Expressions

$$\boxed{B \vdash \text{sigexp} \Rightarrow E}$$

$$\frac{B \vdash \text{spec} \Rightarrow E}{B \vdash \text{sig spec end} \Rightarrow E} \quad (62)$$

$$\frac{B(\text{sigid}) = (T)E \quad T \cap (T \text{ of } B) = \emptyset}{B \vdash \text{sigid} \Rightarrow E} \quad (63)$$

$$\frac{B \vdash \text{sigexp} \Rightarrow E \quad \text{tyvarseq} = \alpha^{(k)} \quad C \text{ of } B \vdash \text{ty} \Rightarrow \tau \quad E(\text{longtycon}) = (t, VE) \quad t \notin T \text{ of } B \quad \varphi = \{t \mapsto \Lambda \alpha^{(k)}. \tau\} \quad \Lambda \alpha^{(k)}. \tau \text{ admits equality, if } t \text{ does} \quad \varphi(E) \text{ well-formed}}{B \vdash \text{sigexp where type tyvarseq longtycon} = \text{ty} \Rightarrow \varphi(E)} \quad (64)$$

Comments:

- (63) The bound names of $B(\text{sigid})$ can always be renamed to satisfy $T \cap (T \text{ of } B) = \emptyset$, if necessary.

$$\boxed{B \vdash \text{sigexp} \Rightarrow \Sigma}$$

$$\frac{B \vdash \text{sigexp} \Rightarrow E \quad T = \text{tynames } E \setminus (T \text{ of } B)}{B \vdash \text{sigexp} \Rightarrow (T)E} \quad (65)$$

Comment: A signature expression *sigexp* which is an immediate constituent of a signature binding, a signature constraint, or a functor binding is elaborated to a signature, see rules 52, 53, 67 and 86.

Signature Declarations

$$\boxed{B \vdash \text{sigdec} \Rightarrow G}$$

$$\frac{B \vdash \text{sigbind} \Rightarrow G}{B \vdash \text{signature sigbind} \Rightarrow G} \quad (66)$$

Signature Bindings

$$\boxed{B \vdash \text{sigbind} \Rightarrow G}$$

$$\frac{B \vdash \text{sigexp} \Rightarrow \Sigma \quad \langle B \vdash \text{sigbind} \Rightarrow G \rangle}{B \vdash \text{sigid} = \text{sigexp} \langle \text{and sigbind} \rangle \Rightarrow \{ \text{sigid} \mapsto \Sigma \} \langle + G \rangle} \quad (67)$$

Specifications

$$\boxed{B \vdash \text{spec} \Rightarrow E}$$

$$\frac{C \text{ of } B \vdash \text{valdesc} \Rightarrow VE}{B \vdash \text{val valdesc} \Rightarrow \text{Clos}VE \text{ in Env}} \quad (68)$$

$$\frac{C \text{ of } B \vdash \text{typdesc} \Rightarrow TE \quad \forall (t, VE) \in \text{Ran } TE, t \text{ does not admit equality}}{B \vdash \text{type typdesc} \Rightarrow TE \text{ in Env}} \quad (69)$$

$$\frac{C \text{ of } B \vdash \text{typdesc} \Rightarrow TE \quad \forall (t, VE) \in \text{Ran } TE, t \text{ admits equality}}{B \vdash \text{eqtype typdesc} \Rightarrow TE \text{ in Env}} \quad (70)$$

$$\frac{C \text{ of } B \oplus TE \vdash \text{datdesc} \Rightarrow VE, TE \quad \forall (t, VE') \in \text{Ran } TE, t \notin T \text{ of } B \\ TE \text{ maximises equality}}{B \vdash \text{datatype datdesc} \Rightarrow (VE, TE) \text{ in Env}} \quad (71)$$

$$\frac{B(\text{longtycon}) = (\theta, VE) \quad TE = \{\text{tycon} \mapsto (\theta, VE)\}}{B \vdash \text{datatype } \text{tycon} \text{ --- datatype } \text{longtycon} \Rightarrow (VE, TE) \text{ in Env}} \quad (72)$$

$$\frac{C \text{ of } B \vdash \text{exdesc} \Rightarrow VE}{B \vdash \text{exception } \text{exdesc} \Rightarrow VE \text{ in Env}} \quad (73)$$

$$\frac{B \vdash \text{strdesc} \Rightarrow SE}{B \vdash \text{structure } \text{strdesc} \Rightarrow SE \text{ in Env}} \quad (74)$$

$$\frac{B \vdash \text{sigexp} \Rightarrow E}{B \vdash \text{include } \text{sigexp} \Rightarrow E} \quad (75)$$

$$\frac{}{B \vdash \quad \Rightarrow \{\} \text{ in Env}} \quad (76)$$

$$\frac{B \vdash \text{spec}_1 \Rightarrow E_1 \quad B \oplus E_1 \vdash \text{spec}_2 \Rightarrow E_2 \quad \text{Dom}(E_1) \cap \text{Dom}(E_2) = \emptyset}{B \vdash \text{spec}_1 \langle ; \rangle \text{spec}_2 \Rightarrow E_1 + E_2} \quad (77)$$

$$\frac{\begin{array}{l} B \vdash \text{spec} \Rightarrow E \quad E(\text{longtycon}_i) = (t_i, VE_i), i = 1..n \\ t \in \{t_1, \dots, t_n\} \quad t \text{ admits equality, if some } t_i \text{ does} \\ \{t_1, \dots, t_n\} \cap T \text{ of } B = \emptyset \quad \varphi = \{t_1 \mapsto t, \dots, t_n \mapsto t\} \end{array}}{B \vdash \text{spec sharing type } \text{longtycon}_1 = \dots = \text{longtycon}_n \Rightarrow \varphi(E)} \quad (78)$$

Comments:

(68) VE is determined by B and valdesc .

(69)–(71) The type names in TE are new.

(73) VE is determined by B and exdesc and contains monotypes only.

(77) Note that no sequential specification is allowed to specify the same identifier twice.

Value Descriptions

$$\boxed{C \vdash \text{valdesc} \Rightarrow VE}$$

$$\frac{C \vdash \text{ty} \Rightarrow \tau \quad \langle C \vdash \text{valdesc} \Rightarrow VE \rangle}{C \vdash \text{vid} : \text{ty} \langle \text{and valdesc} \rangle \Rightarrow \{\text{vid} \mapsto (\tau, \mathbf{v})\} \langle + VE \rangle} \quad (79)$$

Type Descriptions

$$\boxed{C \vdash \text{tydesc} \Rightarrow TE}$$

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad t \notin T \text{ of } C \quad \text{arity } t = k}{\langle C \vdash \text{tydesc} \Rightarrow TE \quad t \notin \text{tynames } TE \rangle} \quad (80)$$

$$C \vdash \text{tyvarseq } \text{tycon} \langle \text{and } \text{tydesc} \rangle \Rightarrow \{ \text{tycon} \mapsto (t, \{\}) \} \langle + TE \rangle$$

Comment: Note that the value environment in the resulting type structure must be empty. For example, `datatype s=C type t sharing type t=s` is a legal specification, but the type structure bound to `t` does not bind any value constructors.

Datatype Descriptions

$$\boxed{C \vdash \text{datdesc} \Rightarrow VE, TE}$$

$$\frac{\text{tyvarseq} = \alpha^{(k)} \quad C, \alpha^{(k)}t \vdash \text{condesc} \Rightarrow VE \quad \text{arity } t = k}{\langle C \vdash \text{datdesc}' \Rightarrow VE', TE' \quad \forall (t', VE'') \in \text{Ran } TE', t \neq t' \rangle} \quad (81)$$

$$C \vdash \text{tyvarseq } \text{tycon} = \text{condesc} \langle \text{and } \text{datdesc}' \rangle \Rightarrow$$

$$\text{ClosVE} \langle + VE' \rangle, \{ \text{tycon} \mapsto (t, \text{ClosVE}) \} \langle + TE' \rangle$$

Constructor Descriptions

$$\boxed{C, \tau \vdash \text{condesc} \Rightarrow VE}$$

$$\frac{\langle C \vdash \text{ty} \Rightarrow \tau' \rangle \quad \langle \langle C, \tau \vdash \text{condesc} \Rightarrow VE \rangle \rangle}{C, \tau \vdash \text{vid} \langle \text{of } \text{ty} \rangle \langle \langle \mid \text{condesc} \rangle \rangle \Rightarrow} \quad (82)$$

$$\{ \text{vid} \mapsto (\tau, \text{c}) \} \langle + \{ \text{vid} \mapsto (\tau' \rightarrow \tau, \text{c}) \} \rangle \langle \langle + VE \rangle \rangle$$

Exception Descriptions

$$\boxed{C \vdash \text{exdesc} \Rightarrow VE}$$

$$\frac{\langle C \vdash \text{ty} \Rightarrow \tau \quad \text{tyvars}(\tau) = \emptyset \rangle \quad \langle \langle C \vdash \text{exdesc} \Rightarrow VE \rangle \rangle}{C \vdash \text{vid} \langle \text{of } \text{ty} \rangle \langle \langle \text{and } \text{exdesc} \rangle \rangle \Rightarrow} \quad (83)$$

$$\{ \text{vid} \mapsto (\text{exn}, \text{e}) \} \langle + \{ \text{vid} \mapsto (\tau \rightarrow \text{exn}, \text{e}) \} \rangle \langle \langle + VE \rangle \rangle$$

Structure Descriptions

$$\boxed{B \vdash \text{strdesc} \Rightarrow SE}$$

$$\frac{B \vdash \text{sigexp} \Rightarrow E \quad \langle B + \text{tynames } E \vdash \text{strdesc} \Rightarrow SE \rangle}{B \vdash \text{strid} : \text{sigexp} \langle \text{and } \text{strdesc} \rangle \Rightarrow \{ \text{strid} \mapsto E \} \langle + SE \rangle} \quad (84)$$

Functor Declarations

$$\boxed{B \vdash \text{fundec} \Rightarrow F}$$

$$\frac{B \vdash \text{funbind} \Rightarrow F}{B \vdash \text{functor funbind} \Rightarrow F} \quad (85)$$

Functor Bindings

$$\boxed{B \vdash \text{funbind} \Rightarrow F}$$

$$\frac{\begin{array}{l} B \vdash \text{sigexp} \Rightarrow (T)E \quad B \oplus \{\text{strid} \mapsto E\} \vdash \text{strex} \Rightarrow E' \\ T \cap (T \text{ of } B) = \emptyset \quad T' = \text{tynames } E' \setminus ((T \text{ of } B) \cup T) \\ \langle B \vdash \text{funbind} \Rightarrow F \rangle \end{array}}{B \vdash \text{funid } (\text{strid} : \text{sigexp}) = \text{strex } \langle \text{and funbind} \rangle \Rightarrow \{ \text{funid} \mapsto (T)(E, (T')E') \} \langle + F \rangle} \quad (86)$$

Comment: Since \oplus is used, any type name t in E acts like a constant in the functor body; in particular, it ensures that further names generated during elaboration of the body are distinct from t . The set T' is chosen such that every name free in $(T)E$ or $(T)(E, (T')E')$ is free in B .

Top-level Declarations

$$\boxed{B \vdash \text{topdec} \Rightarrow B'}$$

$$\frac{\begin{array}{l} B \vdash \text{strdec} \Rightarrow E \quad \langle B \oplus E \vdash \text{topdec} \Rightarrow B' \rangle \\ B'' = (\text{tynames } E, E) \text{ in Basis } \langle +B' \rangle \quad \text{tyvars } B'' = \emptyset \end{array}}{B \vdash \text{strdec } \langle \text{topdec} \rangle \Rightarrow B''} \quad (87)$$

$$\frac{\begin{array}{l} B \vdash \text{sigdec} \Rightarrow G \quad \langle B \oplus G \vdash \text{topdec} \Rightarrow B' \rangle \\ B'' = (\text{tynames } G, G) \text{ in Basis } \langle +B' \rangle \end{array}}{B \vdash \text{sigdec } \langle \text{topdec} \rangle \Rightarrow B''} \quad (88)$$

$$\frac{\begin{array}{l} B \vdash \text{fundec} \Rightarrow F \quad \langle B \oplus F \vdash \text{topdec} \Rightarrow B' \rangle \\ B'' = (\text{tynames } F, F) \text{ in Basis } \langle +B' \rangle \quad \text{tyvars } B'' = \emptyset \end{array}}{B \vdash \text{fundec } \langle \text{topdec} \rangle \Rightarrow B''} \quad (89)$$

Comments:

(87)–(89) No free type variables enter the basis: if $B \vdash \text{topdec} \Rightarrow B'$ then $\text{tyvars}(B') = \emptyset$.

6 Dynamic Semantics for the Core

6.1 Reduced Syntax

Since types are mostly dealt with in the static semantics, the Core syntax is reduced by the following transformations, for the purpose of the dynamic semantics:

- All explicit type ascriptions “: *ty*” are omitted, and qualifications “of *ty*” are omitted from constructor and exception bindings.
- The Core phrase classes Ty and TyRow are omitted.

6.2 Simple Objects

All objects in the dynamic semantics are built from identifier classes together with the simple object classes shown (with the variables which range over them) in Figure 12.

a	\in	Addr	addresses
en	\in	ExName	exception names
b	\in	BasVal	basic values
sv	\in	SVal	special values
		{FAIL}	failure

Figure 12: Simple Semantic Objects

Addr and ExName are infinite sets. BasVal is described below. SVal is the class of values denoted by the special constants SCon. Each integer, word or real constant denotes a value according to normal mathematical conventions; each string or character constant denotes a sequence of characters as explained in Section 2.2. The value denoted by *scon* is written $\text{val}(scon)$. FAIL is the result of a failing attempt to match a value and a pattern. Thus FAIL is neither a value nor an exception, but simply a semantic object used in the rules to express operationally how matching proceeds.

Exception constructors evaluate to exception names. This is to accommodate the generative nature of exception bindings; each evaluation of a declaration of an exception constructor binds it to a new unique name.

$$\begin{aligned}
v &\in \text{Val} = \{:=\} \cup \text{SVal} \cup \text{BasVal} \cup \text{VId} \\
&\quad \cup (\text{VId} \times \text{Val}) \cup \text{ExVal} \\
&\quad \cup \text{Record} \cup \text{Addr} \cup \text{FcnClosure} \\
r &\in \text{Record} = \text{Lab} \xrightarrow{\text{fin}} \text{Val} \\
e &\in \text{ExVal} = \text{ExName} \cup (\text{ExName} \times \text{Val}) \\
[e] \text{ or } p &\in \text{Pack} = \text{ExVal} \\
(\text{match}, E, VE) &\in \text{FcnClosure} = \text{Match} \times \text{Env} \times \text{ValEnv} \\
\text{mem} &\in \text{Mem} = \text{Addr} \xrightarrow{\text{fin}} \text{Val} \\
\text{ens} &\in \text{ExNameSet} = \text{Fin}(\text{ExName}) \\
(\text{mem}, \text{ens}) \text{ or } s &\in \text{State} = \text{Mem} \times \text{ExNameSet} \\
(SE, TE, VE) \text{ or } E &\in \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{ValEnv} \\
SE &\in \text{StrEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Env} \\
TE &\in \text{TyEnv} = \text{TyCon} \xrightarrow{\text{fin}} \text{ValEnv} \\
VE &\in \text{ValEnv} = \text{VId} \xrightarrow{\text{fin}} \text{Val} \times \text{IdStatus}
\end{aligned}$$

Figure 13: Compound Semantic Objects

6.3 Compound Objects

The compound objects for the dynamic semantics are shown in Figure 13. Many conventions and notations are adopted as in the static semantics; in particular projection, injection and modification all retain their meaning. We generally omit the injection functions taking VId , $\text{VId} \times \text{Val}$ etc into Val . For records $r \in \text{Record}$ however, we write this injection explicitly as “in Val ”; this accords with the fact that there is a separate phrase class ExpRow , whose members evaluate to records.

We take \cup to mean disjoint union over semantic object classes. We also understand all the defined object classes to be disjoint. A particular case deserves mention; ExVal and Pack (exception values and packets) are isomorphic classes, but the latter class corresponds to exceptions which have been raised, and therefore has different semantic significance from the former, which is just a subclass of values.

Although the same names, e.g. E for an environment, are used as in the static semantics, the objects denoted are different. This need cause no

confusion since the static and dynamic semantics are presented separately.

6.4 Basic Values

The basic values in `BasVal` are values bound to predefined value variables. In this document, we take `BasVal` to be the singleton set `{=}`; however, libraries may define a larger set of basic values. The meaning of basic values is represented by a function

$$\text{APPLY} : \text{BasVal} \times \text{Val} \rightarrow \text{Val} \cup \text{Pack}$$

which satisfies that $\text{APPLY}(=, \{1 \mapsto v_1, 2 \mapsto v_2\})$ is `true` or `false` according as the values v_1 and v_2 are, or are not, identical values.

6.5 Basic Exceptions

A subset `BasExName` \subset `ExName` of the exception names are bound to predefined exception constructors in the initial dynamic basis (see Appendix D). These names are denoted by the identifiers to which they are bound in the initial basis, and are as follows:

`Match` `Bind`

The exceptions `Match` and `Bind` are raised upon failure of pattern-matching in evaluating a function `fn match` or a `valbind`, as detailed in the rules to follow. Recall from Section 4.11 that in the context `fn match`, the `match` must be irredundant and exhaustive and that the compiler should flag the `match` if it violates these restrictions. The exception `Match` can only be raised for a match which is not exhaustive, and has therefore been flagged by the compiler.

6.6 Function Closures

The informal understanding of a *function closure* $(match, E, VE)$ is as follows: when the function closure is applied to a value v , `match` will be evaluated against v , in the environment E modified in a special sense by VE . The domain $\text{Dom } VE$ of this third component contains those identifiers to be treated recursively in the evaluation. To achieve this effect, the evaluation of `match` will take place not in $E + VE$ but in $E + \text{Rec } VE$, where

$$\text{Rec} : \text{ValEnv} \rightarrow \text{ValEnv}$$

is defined as follows:

- $\text{Dom}(\text{Rec } VE) = \text{Dom } VE$
- If $VE(vid) \notin \text{FcnClosure} \times \{\mathbf{v}\}$, then $(\text{Rec } VE)(vid) = VE(vid)$
- If $VE(vid) = ((match', E', VE'), \mathbf{v})$ then $(\text{Rec } VE)(vid) = ((match', E', VE), \mathbf{v})$

The effect is that, before application of $(match, E, VE)$ to v , the function closures in $\text{Ran } VE$ are “unrolled” once, to prepare for their possible recursive application during the evaluation of $match$ upon v .

This device is adopted to ensure that all semantic objects are finite (by controlling the unrolling of recursion). The operator Rec is invoked in just two places in the semantic rules: in the rule for recursive value bindings of the form “**rec valbind**”, and in the rule for evaluating an application expression “*exp atexp*” in the case that exp evaluates to a function closure.

6.7 Inference Rules

The semantic rules allow sentences of the form

$$s, A \vdash \textit{phrase} \Rightarrow A', s'$$

to be inferred, where A is usually an environment, A' is some semantic object and s, s' are the states before and after the evaluation represented by the sentence. Some hypotheses in rules are not of this form; they are called *side-conditions*. The convention for options is the same as for the Core static semantics.

In most rules the states s and s' are omitted from sentences; they are only included for those rules which are directly concerned with the state – either referring to its contents or changing it. When omitted, the convention for restoring them is as follows. If the rule is presented in the form

$$\frac{A_1 \vdash \textit{phrase}_1 \Rightarrow A'_1 \quad A_2 \vdash \textit{phrase}_2 \Rightarrow A'_2 \quad \dots \quad \dots \quad A_n \vdash \textit{phrase}_n \Rightarrow A'_n}{A \vdash \textit{phrase} \Rightarrow A'}$$

then the full form is intended to be

$$\frac{s_0, A_1 \vdash \textit{phrase}_1 \Rightarrow A'_1, s_1 \quad s_1, A_2 \vdash \textit{phrase}_2 \Rightarrow A'_2, s_2 \quad \dots \quad \dots \quad s_{n-1}, A_n \vdash \textit{phrase}_n \Rightarrow A'_n, s_n}{s_0, A \vdash \textit{phrase} \Rightarrow A', s_n}$$

(Any side-conditions are left unaltered). Thus the left-to-right order of the hypotheses indicates the order of evaluation. Note that in the case $n = 0$, when there are no hypotheses (except possibly side-conditions), we have $s_n = s_0$; this implies that the rule causes no side effect. The convention is called the *state convention*, and must be applied to each version of a rule obtained by inclusion or omission of its options.

A second convention, the *exception convention*, is adopted to deal with the propagation of exception packets p . For each rule whose full form (ignoring side-conditions) is

$$\frac{s_1, A_1 \vdash \textit{phrase}_1 \Rightarrow A'_1, s'_1 \quad \cdots \quad s_n, A_n \vdash \textit{phrase}_n \Rightarrow A'_n, s'_n}{s, A \vdash \textit{phrase} \Rightarrow A', s'}$$

and for each k , $1 \leq k \leq n$, for which the result A'_k is not a packet p , an extra rule is added of the form

$$\frac{s_1, A_1 \vdash \textit{phrase}_1 \Rightarrow A'_1, s'_1 \quad \cdots \quad s_k, A_k \vdash \textit{phrase}_k \Rightarrow p', s'}{s, A \vdash \textit{phrase} \Rightarrow p', s'}$$

where p' does not occur in the original rule.¹ This indicates that evaluation of phrases in the hypothesis terminates with the first whose result is a packet (other than one already treated in the rule), and this packet is the result of the phrase in the conclusion.

A third convention is that we allow compound variables (variables built from the variables in Figure 13 and the symbol “/”) to range over unions of semantic objects. For instance the compound variable v/p ranges over $\text{Val} \cup \text{Pack}$. We also allow x/FAIL to range over $X \cup \{\text{FAIL}\}$ where x ranges over X ; furthermore, we extend environment modification to allow for failure as follows:

$$VE + \text{FAIL} = \text{FAIL}.$$

Atomic Expressions

$$\boxed{E \vdash \textit{atexp} \Rightarrow v/p}$$

$$\frac{}{E \vdash \textit{scon} \Rightarrow \text{val}(\textit{scon})} \quad (90)$$

$$\frac{E(\textit{longvid}) = (v, \textit{is})}{E \vdash \textit{longvid} \Rightarrow v} \quad (91)$$

¹There is one exception to the exception convention; no extra rule is added for rule 104 which deals with handlers, since a handler is the only means by which propagation of an exception can be arrested.

$$\frac{\langle E \vdash \text{exprow} \Rightarrow r \rangle}{E \vdash \{ \langle \text{exprow} \rangle \} \Rightarrow \{ \} \langle + r \rangle \text{ in Val}} \quad (92)$$

$$\frac{E \vdash \text{dec} \Rightarrow E' \quad E + E' \vdash \text{exp} \Rightarrow v}{E \vdash \text{let dec in exp end} \Rightarrow v} \quad (93)$$

$$\frac{E \vdash \text{exp} \Rightarrow v}{E \vdash (\text{exp}) \Rightarrow v} \quad (94)$$

Comments:

- (91) As in the static semantics, value identifiers are looked up in the environment and the identifier status is not used.

Expression Rows

$$\boxed{E \vdash \text{exprow} \Rightarrow r/p}$$

$$\frac{E \vdash \text{exp} \Rightarrow v \quad \langle E \vdash \text{exprow} \Rightarrow r \rangle}{E \vdash \text{lab} = \text{exp} \langle , \text{exprow} \rangle \Rightarrow \{ \text{lab} \mapsto v \} \langle + r \rangle} \quad (95)$$

Comment: We may think of components as being evaluated from left to right, because of the state and exception conventions.

Expressions

$$\boxed{E \vdash \text{exp} \Rightarrow v/p}$$

$$\frac{E \vdash \text{atexp} \Rightarrow v}{E \vdash \text{atexp} \Rightarrow v} \quad (96)$$

$$\frac{E \vdash \text{exp} \Rightarrow \text{vid} \quad \text{vid} \neq \text{ref} \quad E \vdash \text{atexp} \Rightarrow v}{E \vdash \text{exp atexp} \Rightarrow (\text{vid}, v)} \quad (97)$$

$$\frac{E \vdash \text{exp} \Rightarrow \text{en} \quad E \vdash \text{atexp} \Rightarrow v}{E \vdash \text{exp atexp} \Rightarrow (\text{en}, v)} \quad (98)$$

$$\frac{s, E \vdash \text{exp} \Rightarrow \text{ref}, s' \quad s', E \vdash \text{atexp} \Rightarrow v, s'' \quad a \notin \text{Dom}(\text{mem of } s'')}{s, E \vdash \text{exp atexp} \Rightarrow a, s'' + \{a \mapsto v\}} \quad (99)$$

$$\frac{s, E \vdash \text{exp} \Rightarrow :=, s' \quad s', E \vdash \text{atexp} \Rightarrow \{1 \mapsto a, 2 \mapsto v\}, s''}{s, E \vdash \text{exp atexp} \Rightarrow \{ \} \text{ in Val}, s'' + \{a \mapsto v\}} \quad (100)$$

$$\frac{E \vdash \text{exp} \Rightarrow b \quad E \vdash \text{atexp} \Rightarrow v \quad \text{APPLY}(b, v) = v'/p}{E \vdash \text{exp atexp} \Rightarrow v'/p} \quad (101)$$

$$\frac{E \vdash \text{exp} \Rightarrow (\text{match}, E', VE) \quad E \vdash \text{atexp} \Rightarrow v \quad E' + \text{Rec } VE, v \vdash \text{match} \Rightarrow v'}{E \vdash \text{exp atexp} \Rightarrow v'} \quad (102)$$

$$\frac{E \vdash \text{exp} \Rightarrow (\text{match}, E', VE) \quad E \vdash \text{atexp} \Rightarrow v \quad E' + \text{Rec } VE, v \vdash \text{match} \Rightarrow \text{FAIL}}{E \vdash \text{exp atexp} \Rightarrow [\text{Match}]} \quad (103)$$

$$\frac{E \vdash \text{exp} \Rightarrow v}{E \vdash \text{exp handle match} \Rightarrow v} \quad (104)$$

$$\frac{E \vdash \text{exp} \Rightarrow [e] \quad E, e \vdash \text{match} \Rightarrow v}{E \vdash \text{exp handle match} \Rightarrow v} \quad (105)$$

$$\frac{E \vdash \text{exp} \Rightarrow [e] \quad E, e \vdash \text{match} \Rightarrow \text{FAIL}}{E \vdash \text{exp handle match} \Rightarrow [e]} \quad (106)$$

$$\frac{E \vdash \text{exp} \Rightarrow e}{E \vdash \text{raise exp} \Rightarrow [e]} \quad (107)$$

$$\overline{E \vdash \text{fn match} \Rightarrow (\text{match}, E, \{\})} \quad (108)$$

Comments:

- (99) The side condition ensures that a new address is chosen. There are no rules concerning disposal of inaccessible addresses.
- (97)–(103) Note that none of the rules for function application has a premise in which the operator evaluates to a constructed value, a record or an address. This is because we are interested in the evaluation of well-typed programs only, and in such programs *exp* will always have a functional type.

- (104) This is the only rule to which the exception convention does not apply. If the operator evaluates to a packet then rule 105 or rule 106 must be used.
- (106) Packets that are not handled by the *match* propagate.
- (108) The third component of the function closure is empty because the *match* does not introduce new recursively defined values.

Matches

$$\boxed{E, v \vdash \text{match} \Rightarrow v'/p/\text{FAIL}}$$

$$\frac{E, v \vdash \text{mrule} \Rightarrow v'}{E, v \vdash \text{mrule} \langle \mid \text{match} \rangle \Rightarrow v'} \quad (109)$$

$$\frac{E, v \vdash \text{mrule} \Rightarrow \text{FAIL}}{E, v \vdash \text{mrule} \Rightarrow \text{FAIL}} \quad (110)$$

$$\frac{E, v \vdash \text{mrule} \Rightarrow \text{FAIL} \quad E, v \vdash \text{match} \Rightarrow v'/\text{FAIL}}{E, v \vdash \text{mrule} \mid \text{match} \Rightarrow v'/\text{FAIL}} \quad (111)$$

Comment: A value v occurs on the left of the turnstile, in evaluating a *match*. We may think of a *match* as being evaluated *against* a value; similarly, we may think of a pattern as being evaluated *against* a value. Alternative match rules are tried from left to right.

Match Rules

$$\boxed{E, v \vdash \text{mrule} \Rightarrow v'/p/\text{FAIL}}$$

$$\frac{E, v \vdash \text{pat} \Rightarrow VE \quad E + VE \vdash \text{exp} \Rightarrow v'}{E, v \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow v'} \quad (112)$$

$$\frac{E, v \vdash \text{pat} \Rightarrow \text{FAIL}}{E, v \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \text{FAIL}} \quad (113)$$

Declarations

$$\boxed{E \vdash \text{dec} \Rightarrow E'/p}$$

$$\frac{E \vdash \text{valbind} \Rightarrow VE}{E \vdash \text{val tyvarseq valbind} \Rightarrow VE \text{ in Env}} \quad (114)$$

$$\frac{\vdash \text{typbind} \Rightarrow TE}{E \vdash \text{type typbind} \Rightarrow TE \text{ in Env}} \quad (115)$$

$$\frac{\vdash \text{datbind} \Rightarrow VE, TE}{E \vdash \text{datatype } \text{datbind} \Rightarrow (VE, TE) \text{ in Env}} \quad (116)$$

$$\frac{E(\text{longtycon}) = VE}{E \vdash \text{datatype } \text{tycon} \text{ ==- datatype } \text{longtycon} \Rightarrow (VE, \{\text{tycon} \mapsto VE\}) \text{ in Env}} \quad (117)$$

$$\frac{\vdash \text{datbind} \Rightarrow VE, TE \quad E + VE \vdash \text{dec} \Rightarrow E'}{E \vdash \text{abstype } \text{datbind with } \text{dec end} \Rightarrow E'} \quad (118)$$

$$\frac{E \vdash \text{exbind} \Rightarrow VE}{E \vdash \text{exception } \text{exbind} \Rightarrow VE \text{ in Env}} \quad (119)$$

$$\frac{E \vdash \text{dec}_1 \Rightarrow E_1 \quad E + E_1 \vdash \text{dec}_2 \Rightarrow E_2}{E \vdash \text{local } \text{dec}_1 \text{ in } \text{dec}_2 \text{ end} \Rightarrow E_2} \quad (120)$$

$$\frac{E(\text{longstrid}_1) = E_1 \quad \cdots \quad E(\text{longstrid}_n) = E_n}{E \vdash \text{open } \text{longstrid}_1 \cdots \text{longstrid}_n \Rightarrow E_1 + \cdots + E_n} \quad (121)$$

$$\frac{}{E \vdash \quad \Rightarrow \{\} \text{ in Env}} \quad (122)$$

$$\frac{E \vdash \text{dec}_1 \Rightarrow E_1 \quad E + E_1 \vdash \text{dec}_2 \Rightarrow E_2}{E \vdash \text{dec}_1 \langle ; \rangle \text{dec}_2 \Rightarrow E_1 + E_2} \quad (123)$$

Value Bindings

$$\boxed{E \vdash \text{valbind} \Rightarrow VE/p}$$

$$\frac{E \vdash \text{exp} \Rightarrow v \quad E, v \vdash \text{pat} \Rightarrow VE \quad \langle E \vdash \text{valbind} \Rightarrow VE' \rangle}{E \vdash \text{pat} = \text{exp} \langle \text{and } \text{valbind} \rangle \Rightarrow VE \langle + VE' \rangle} \quad (124)$$

$$\frac{E \vdash \text{exp} \Rightarrow v \quad E, v \vdash \text{pat} \Rightarrow \text{FAIL}}{E \vdash \text{pat} = \text{exp} \langle \text{and } \text{valbind} \rangle \Rightarrow [\text{Bind}]} \quad (125)$$

$$\frac{E \vdash \text{valbind} \Rightarrow VE}{E \vdash \text{rec } \text{valbind} \Rightarrow \text{Rec } VE} \quad (126)$$

Type Bindings

$$\boxed{\vdash \text{typbind} \Rightarrow TE}$$

$$\frac{\langle \vdash \text{typbind} \Rightarrow TE \rangle}{\vdash \text{tyvarseq } \text{tycon} = \text{ty} \langle \text{and } \text{typbind} \rangle \Rightarrow \{\text{tycon} \mapsto \{\}\} \langle + TE \rangle} \quad (127)$$

Datatype Bindings

$$\boxed{\vdash \text{datbind} \Rightarrow VE, TE}$$

$$\frac{\vdash \text{conbind} \Rightarrow VE \quad \langle \vdash \text{datbind}' \Rightarrow VE', TE' \rangle}{\vdash \text{tyvarseq tycon}=\text{conbind} \langle \text{and datbind}' \rangle \Rightarrow VE \langle +VE' \rangle, \{ \text{tycon} \mapsto VE \} \langle +TE' \rangle} \quad (128)$$

Constructor Bindings

$$\boxed{\vdash \text{conbind} \Rightarrow VE}$$

$$\frac{\langle \vdash \text{conbind} \Rightarrow VE \rangle}{\vdash \text{vid} \langle | \text{conbind} \rangle \Rightarrow \{ \text{vid} \mapsto (\text{vid}, \mathbf{c}) \} \langle +VE \rangle} \quad (129)$$

Exception Bindings

$$\boxed{E \vdash \text{exbind} \Rightarrow VE}$$

$$\frac{\text{en} \notin \text{ens of } s \quad s' = s + \{ \text{en} \} \quad \langle s', E \vdash \text{exbind} \Rightarrow VE, s'' \rangle}{s, E \vdash \text{vid} \langle \text{and exbind} \rangle \Rightarrow \{ \text{vid} \mapsto (\text{en}, \mathbf{e}) \} \langle +VE \rangle, s' \langle ' \rangle} \quad (130)$$

$$\frac{E(\text{longvid}) = (\text{en}, \mathbf{e}) \quad \langle E \vdash \text{exbind} \Rightarrow VE \rangle}{E \vdash \text{vid} = \text{longvid} \langle \text{and exbind} \rangle \Rightarrow \{ \text{vid} \mapsto (\text{en}, \mathbf{e}) \} \langle +VE \rangle} \quad (131)$$

Comments:

(130) The two side conditions ensure that a new exception name is generated and recorded as “used” in subsequent states.

Atomic Patterns

$$\boxed{E, v \vdash \text{atpat} \Rightarrow VE/\text{FAIL}}$$

$$\overline{E, v \vdash _ \Rightarrow \{ \}} \quad (132)$$

$$\frac{v = \text{val}(s\text{con})}{E, v \vdash s\text{con} \Rightarrow \{ \}} \quad (133)$$

$$\frac{v \neq \text{val}(s\text{con})}{E, v \vdash s\text{con} \Rightarrow \text{FAIL}} \quad (134)$$

$$\frac{\text{vid} \notin \text{Dom}(E) \text{ or } \text{is of } E(\text{vid}) = \mathbf{v}}{E, v \vdash \text{vid} \Rightarrow \{ \text{vid} \mapsto (v, \mathbf{v}) \}} \quad (135)$$

$$\frac{E(\text{longvid}) = (v, \text{is}) \quad \text{is} \neq \mathbf{v}}{E, v \vdash \text{longvid} \Rightarrow \{ \}} \quad (136)$$

$$\frac{E(\text{longvid}) = (v', is) \quad is \neq \mathbf{v} \quad v \neq v'}{E, v \vdash \text{longvid} \Rightarrow \text{FAIL}} \quad (137)$$

$$\frac{v = \{\}\langle +r \rangle \text{ in Val} \quad \langle E, r \vdash \text{patrow} \Rightarrow VE/\text{FAIL} \rangle}{E, v \vdash \{ \langle \text{patrow} \rangle \} \Rightarrow \{\}\langle +VE/\text{FAIL} \rangle} \quad (138)$$

$$\frac{E, v \vdash \text{pat} \Rightarrow VE/\text{FAIL}}{E, v \vdash (\text{pat}) \Rightarrow VE/\text{FAIL}} \quad (139)$$

Comments:

(134), (137) Any evaluation resulting in FAIL must do so because rule 134, rule 137, rule 145, or rule 147 has been applied.

Pattern Rows

$$\boxed{E, r \vdash \text{patrow} \Rightarrow VE/\text{FAIL}}$$

$$\overline{E, r \vdash \dots \Rightarrow \{\}} \quad (140)$$

$$\frac{E, r(\text{lab}) \vdash \text{pat} \Rightarrow \text{FAIL}}{E, r \vdash \text{lab} = \text{pat} \langle , \text{patrow} \rangle \Rightarrow \text{FAIL}} \quad (141)$$

$$\frac{E, r(\text{lab}) \vdash \text{pat} \Rightarrow VE \quad \langle E, r \vdash \text{patrow} \Rightarrow VE'/\text{FAIL} \rangle}{E, r \vdash \text{lab} = \text{pat} \langle , \text{patrow} \rangle \Rightarrow VE \langle + VE'/\text{FAIL} \rangle} \quad (142)$$

Comments:

(141),(142) For well-typed programs lab will be in the domain of r .

Patterns

$$\boxed{E, v \vdash \text{pat} \Rightarrow VE/\text{FAIL}}$$

$$\frac{E, v \vdash \text{atpat} \Rightarrow VE/\text{FAIL}}{E, v \vdash \text{atpat} \Rightarrow VE/\text{FAIL}} \quad (143)$$

$$\frac{E(\text{longvid}) = (vid, \mathbf{c}) \quad vid \neq \mathbf{ref} \quad v = (vid, v') \quad E, v' \vdash \text{atpat} \Rightarrow VE/\text{FAIL}}{E, v \vdash \text{longvid atpat} \Rightarrow VE/\text{FAIL}} \quad (144)$$

$$\frac{E(\text{longvid}) = (vid, \mathbf{c}) \quad vid \neq \mathbf{ref} \quad v \notin \{vid\} \times \text{Val}}{E, v \vdash \text{longvid atpat} \Rightarrow \text{FAIL}} \quad (145)$$

$$\frac{E(\text{longvid}) = (en, \mathbf{e}) \quad v = (en, v') \quad E, v' \vdash \text{atpat} \Rightarrow VE/\text{FAIL}}{E, v \vdash \text{longvid atpat} \Rightarrow VE/\text{FAIL}} \quad (146)$$

$$\frac{E(\text{longvid}) = (en, \mathbf{e}) \quad v \notin \{en\} \times \text{Val}}{E, v \vdash \text{longvid atpat} \Rightarrow \text{FAIL}} \quad (147)$$

$$\frac{s(a) = v \quad s, E, v \vdash \text{atpat} \Rightarrow VE/\text{FAIL}, s}{s, E, a \vdash \text{ref atpat} \Rightarrow VE/\text{FAIL}, s} \quad (148)$$

$$\frac{E, v \vdash \text{pat} \Rightarrow VE/\text{FAIL}}{E, v \vdash \text{vid as pat} \Rightarrow \{\text{vid} \mapsto (v, \mathbf{v})\} + VE/\text{FAIL}} \quad (149)$$

Comments:

(145),(147) Any evaluation resulting in FAIL must do so because rule 134, rule 137, rule 145, or rule 147 has been applied.

7 Dynamic Semantics for Modules

7.1 Reduced Syntax

Since signature expressions are mostly dealt with in the static semantics, the dynamic semantics need only take limited account of them. However, they cannot be ignored completely; the reason is that an explicit signature ascription plays the rôle of restricting the “view” of a structure – that is, restricting the domains of its component environments and imposing identifier status on value identifiers. The syntax is therefore reduced by the following transformations (in addition to those for the Core), for the purpose of the dynamic semantics of Modules:

- Qualifications “of *ty*” are omitted from constructor and exception descriptions.
- Any qualification **sharing type** ... on a specification or **where type** ... on a signature expression is omitted.

7.2 Compound Objects

The compound objects for the Modules dynamic semantics, extra to those for the Core dynamic semantics, are shown in Figure 14. An *interface*

$$\begin{aligned}
 (\textit{strid} : I, \textit{stexp}, B) &\in \text{FunctorClosure} \\
 &= (\text{StrId} \times \text{Int}) \times \text{StrExp} \times \text{Basis} \\
 I \text{ or } (SI, TI, VI) &\in \text{Int} = \text{StrInt} \times \text{TyInt} \times \text{ValInt} \\
 SI &\in \text{StrInt} = \text{StrId} \xrightarrow{\text{fn}} \text{Int} \\
 TI &\in \text{TyInt} = \text{TyCon} \xrightarrow{\text{fn}} \text{ValInt} \\
 VI &\in \text{ValInt} = \text{VId} \xrightarrow{\text{fn}} \text{IdStatus} \\
 G &\in \text{SigEnv} = \text{SigId} \xrightarrow{\text{fn}} \text{Int} \\
 F &\in \text{FunEnv} = \text{FunId} \xrightarrow{\text{fn}} \text{FunctorClosure} \\
 (F, G, E) \text{ or } B &\in \text{Basis} = \text{FunEnv} \times \text{SigEnv} \times \text{Env} \\
 (G, I) \text{ or } IB &\in \text{IntBasis} = \text{SigEnv} \times \text{Int}
 \end{aligned}$$

Figure 14: Compound Semantic Objects

$I \in \text{Int}$ represents a “view” of a structure. Specifications and signature expressions will evaluate to interfaces; moreover, during the evaluation of a specification or signature expression, structures (to which a specification or signature expression may refer via datatype replicating specifications) are represented only by their interfaces. To extract a value interface from a dynamic value environment we define the operation $\text{Inter} : \text{ValEnv} \rightarrow \text{ValInt}$ as follows:

$$\text{Inter}(VE) = \{vid \mapsto is ; VE(vid) = (v, is)\}$$

In other words, $\text{Inter}(VE)$ is the value interface obtained from VE by removing all values from VE . We then extend Inter to a function $\text{Inter} : \text{Env} \rightarrow \text{Int}$ as follows:

$$\text{Inter}(SE, TE, VE) = (SI, TI, VI)$$

where $VI = \text{Inter}(VE)$ and

$$\begin{aligned} SI &= \{strid \mapsto \text{Inter } E ; SE(strid) = E\} \\ TI &= \{tycon \mapsto \text{Inter } VE' ; TE(tycon) = VE'\} \end{aligned}$$

An *interface basis* $IB = (G, I)$ is a value-free part of a basis, sufficient to evaluate signature expressions and specifications. The function Inter is extended to create an interface basis from a basis B as follows:

$$\text{Inter}(F, G, E) = (G, \text{Inter } E)$$

A further operation

$$\downarrow : \text{Env} \times \text{Int} \rightarrow \text{Env}$$

is required, to cut down an environment E to a given interface I , representing the effect of an explicit signature ascription. We first define $\downarrow : \text{ValEnv} \times \text{ValInt} \rightarrow \text{ValEnv}$ by

$$VE \downarrow VI = \{vid \mapsto (v, is) ; VE(vid) = (v, is') \text{ and } VI(vid) = is\}$$

(Note that VE and VI need not have the same domain and that the identifier status is taken from VI .) We then define $\downarrow : \text{StrEnv} \times \text{StrInt} \rightarrow \text{StrEnv}$, $\downarrow : \text{TyEnv} \times \text{TyInt} \rightarrow \text{TyEnv}$ and $\downarrow : \text{Env} \times \text{Int} \rightarrow \text{Env}$ simultaneously as follows:

$$\begin{aligned}
SE \downarrow SI &= \{strid \mapsto E \downarrow I ; SE(strid) = E \text{ and } SI(strid) = I\} \\
TE \downarrow TI &= \{tycon \mapsto VE' \downarrow VI' ; TE(tycon) = VE' \text{ and } TI(tycon) = VI'\} \\
(SE, TE, VE) \downarrow (SI, TI, VI) &= (SE \downarrow SI, TE \downarrow TI, VE \downarrow VI)
\end{aligned}$$

It is important to note that an interface can also be obtained from the *static* value Σ of a signature expression; it is obtained by first replacing every type structure (θ, VE) in the range of every type environment TE by VE and then replacing each pair (σ, is) in the range of every value environment VE by is . Thus in an implementation interfaces would naturally be obtained from the static elaboration; we choose to give separate rules here for obtaining them in the dynamic semantics since we wish to maintain our separation of the static and dynamic semantics, for reasons of presentation.

7.3 Inference Rules

The semantic rules allow sentences of the form

$$s, A \vdash phrase \Rightarrow A', s'$$

to be inferred, where A is either a basis, a signature environment or empty, A' is some semantic object and s, s' are the states before and after the evaluation represented by the sentence. Some hypotheses in rules are not of this form; they are called *side-conditions*. The convention for options is the same as for the Core static semantics.

The state and exception conventions are adopted as in the Core dynamic semantics. However, it may be shown that the only Modules phrases whose evaluation may cause a side-effect or generate an exception packet are of the form *strex*, *strdec*, *strbind* or *topdec*.

Structure Expressions

$$\boxed{B \vdash strexp \Rightarrow E/p}$$

$$\frac{B \vdash strdec \Rightarrow E}{B \vdash \mathbf{struct} \ strdec \ \mathbf{end} \Rightarrow E} \quad (150)$$

$$\frac{B(longstrid) = E}{B \vdash longstrid \Rightarrow E} \quad (151)$$

$$\frac{B \vdash strexp \Rightarrow E \quad \text{Inter } B \vdash sigexp \Rightarrow I}{B \vdash strexp : sigexp \Rightarrow E \downarrow I} \quad (152)$$

$$\frac{B \vdash \text{strex}p \Rightarrow E \quad \text{Inter } B \vdash \text{sigexp} \Rightarrow I}{B \vdash \text{strex}p : > \text{sigexp} \Rightarrow E \downarrow I} \quad (153)$$

$$\frac{B(\text{funid}) = (\text{strid} : I, \text{strex}p', B') \quad B \vdash \text{strex}p \Rightarrow E \quad B' + \{\text{strid} \mapsto E \downarrow I\} \vdash \text{strex}p' \Rightarrow E'}{B \vdash \text{funid} (\text{strex}p) \Rightarrow E'} \quad (154)$$

$$\frac{B \vdash \text{strdec} \Rightarrow E \quad B + E \vdash \text{strex}p \Rightarrow E'}{B \vdash \text{let } \text{strdec} \text{ in } \text{strex}p \text{ end} \Rightarrow E'} \quad (155)$$

Comments:

- (154) Before the evaluation of the functor body $\text{strex}p'$, the actual argument E is cut down by the formal parameter interface I , so that any opening of strid resulting from the evaluation of $\text{strex}p'$ will produce no more components than anticipated during the static elaboration.

Structure-level Declarations

$$\boxed{B \vdash \text{strdec} \Rightarrow E/p}$$

$$\frac{E \text{ of } B \vdash \text{dec} \Rightarrow E'}{B \vdash \text{dec} \Rightarrow E'} \quad (156)$$

$$\frac{B \vdash \text{strbind} \Rightarrow SE}{B \vdash \text{structure } \text{strbind} \Rightarrow SE \text{ in Env}} \quad (157)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B + E_1 \vdash \text{strdec}_2 \Rightarrow E_2}{B \vdash \text{local } \text{strdec}_1 \text{ in } \text{strdec}_2 \text{ end} \Rightarrow E_2} \quad (158)$$

$$\frac{}{B \vdash \quad \Rightarrow \{\} \text{ in Env}} \quad (159)$$

$$\frac{B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B + E_1 \vdash \text{strdec}_2 \Rightarrow E_2}{B \vdash \text{strdec}_1 \langle ; \rangle \text{strdec}_2 \Rightarrow E_1 + E_2} \quad (160)$$

Structure Bindings

$$\boxed{B \vdash \text{strbind} \Rightarrow SE/p}$$

$$\frac{B \vdash \text{strex}p \Rightarrow E \quad \langle B \vdash \text{strbind} \Rightarrow SE \rangle}{B \vdash \text{strid} = \text{strex}p \langle \text{and } \text{strbind} \rangle \Rightarrow \{\text{strid} \mapsto E\} \langle + SE \rangle} \quad (161)$$

Signature Expressions

$$\boxed{IB \vdash sigexp \Rightarrow I}$$

$$\frac{IB \vdash spec \Rightarrow I}{IB \vdash \mathbf{sig\ spec\ end} \Rightarrow I} \quad (162)$$

$$\frac{IB(sigid) = I}{IB \vdash sigid \Rightarrow I} \quad (163)$$

Signature Declarations

$$\boxed{IB \vdash sigdec \Rightarrow G}$$

$$\frac{IB \vdash sigbind \Rightarrow G}{IB \vdash \mathbf{signature\ sigbind} \Rightarrow G} \quad (164)$$

Signature Bindings

$$\boxed{IB \vdash sigbind \Rightarrow G}$$

$$\frac{IB \vdash sigexp \Rightarrow I \quad \langle IB \vdash sigbind \Rightarrow G \rangle}{IB \vdash sigid = sigexp \langle \mathbf{and\ sigbind} \rangle \Rightarrow \{sigid \mapsto I\} \langle + G \rangle} \quad (165)$$

Specifications

$$\boxed{IB \vdash spec \Rightarrow I}$$

$$\frac{\vdash valdesc \Rightarrow VI}{IB \vdash \mathbf{val\ valdesc} \Rightarrow VI \text{ in Int}} \quad (166)$$

$$\frac{\vdash typdesc \Rightarrow TI}{IB \vdash \mathbf{type\ typdesc} \Rightarrow TI \text{ in Int}} \quad (167)$$

$$\frac{\vdash typdesc \Rightarrow TI}{IB \vdash \mathbf{eqtype\ typdesc} \Rightarrow TI \text{ in Int}} \quad (168)$$

$$\frac{\vdash datdesc \Rightarrow VI, TI}{IB \vdash \mathbf{datatype\ datdesc} \Rightarrow (VI, TI) \text{ in Int}} \quad (169)$$

$$\frac{IB(longtycon) = VI \quad TI = \{tycon \mapsto VI\}}{IB \vdash \mathbf{datatype\ tycon\ ---\ datatype\ longtycon} \Rightarrow (VI, TI) \text{ in Int}} \quad (170)$$

$$\frac{\vdash exdesc \Rightarrow VI}{IB \vdash \mathbf{exception\ exdesc} \Rightarrow VI \text{ in Int}} \quad (171)$$

$$\frac{IB \vdash strdesc \Rightarrow SI}{IB \vdash \mathbf{structure\ strdesc} \Rightarrow SI \text{ in Int}} \quad (172)$$

$$\frac{IB \vdash \text{sigexp} \Rightarrow I}{IB \vdash \text{include sigexp} \Rightarrow I} \quad (173)$$

$$\frac{}{IB \vdash \quad \Rightarrow \{\} \text{ in Int}} \quad (174)$$

$$\frac{IB \vdash \text{spec}_1 \Rightarrow I_1 \quad IB + I_1 \vdash \text{spec}_2 \Rightarrow I_2}{IB \vdash \text{spec}_1 \langle ; \rangle \text{spec}_2 \Rightarrow I_1 + I_2} \quad (175)$$

Value Descriptions

$$\boxed{IB \vdash \text{valdesc} \Rightarrow VI}$$

$$\frac{\langle IB \vdash \text{valdesc} \Rightarrow VI \rangle}{IB \vdash \text{vid} \langle \text{and valdesc} \rangle \Rightarrow \{\text{vid} \mapsto v\} \langle +VI \rangle} \quad (176)$$

Type Descriptions

$$\boxed{IB \vdash \text{typdesc} \Rightarrow TI}$$

$$\frac{\langle IB \vdash \text{typdesc} \Rightarrow TI \rangle}{IB \vdash \text{tyvarseq tycon} \langle \text{and typdesc} \rangle \Rightarrow \{\text{tycon} \mapsto \{\}\} \langle +TI \rangle} \quad (177)$$

Datatype Descriptions

$$\boxed{IB \vdash \text{datdesc} \Rightarrow VI, TI}$$

$$\frac{IB \vdash \text{condesc} \Rightarrow VI \quad \langle IB \vdash \text{datdesc}' \Rightarrow VI', TI' \rangle}{IB \vdash \text{tyvarseq tycon} = \text{condesc} \langle \text{and datdesc}' \rangle \Rightarrow VI \langle +VI' \rangle, \{\text{tycon} \mapsto VI\} \langle +TI' \rangle} \quad (178)$$

Constructor Descriptions

$$\boxed{IB \vdash \text{condesc} \Rightarrow VI}$$

$$\frac{\langle IB \vdash \text{condesc} \Rightarrow VI \rangle}{IB \vdash \text{vid} \langle | \text{condesc} \rangle \Rightarrow \{\text{vid} \mapsto c\} \langle +VI \rangle} \quad (179)$$

Exception Descriptions

$$\boxed{IB \vdash \text{exdesc} \Rightarrow VI}$$

$$\frac{\langle IB \vdash \text{exdesc} \Rightarrow VI \rangle}{IB \vdash \text{vid} \langle \text{and exdesc} \rangle \Rightarrow \{\text{vid} \mapsto e\} \langle +VI \rangle} \quad (180)$$

Structure Descriptions

$$\boxed{IB \vdash \text{strdesc} \Rightarrow SI}$$

$$\frac{IB \vdash \text{sigexp} \Rightarrow I \quad \langle IB \vdash \text{strdesc} \Rightarrow SI \rangle}{IB \vdash \text{strid} : \text{sigexp} \langle \text{and strdesc} \rangle \Rightarrow \{\text{strid} \mapsto I\} \langle +SI \rangle} \quad (181)$$

Functor Bindings

$$\boxed{B \vdash \text{funbind} \Rightarrow F}$$

$$\frac{\text{Inter } B \vdash \text{sigexp} \Rightarrow I \quad \langle IB \vdash \text{funbind} \Rightarrow F \rangle}{IB \vdash \text{funid } (\text{strid} : \text{sigexp}) = \text{strexpr } \langle \text{and funbind} \rangle \Rightarrow \{ \text{funid} \mapsto (\text{strid} : I, \text{strexpr}, B) \} \langle + F \rangle} \quad (182)$$

Functor Declarations

$$\boxed{B \vdash \text{fundec} \Rightarrow F}$$

$$\frac{B \vdash \text{funbind} \Rightarrow F}{B \vdash \text{functor funbind} \Rightarrow F} \quad (183)$$

Top-level Declarations

$$\boxed{B \vdash \text{topdec} \Rightarrow B'/p}$$

$$\frac{B \vdash \text{strdec} \Rightarrow E \quad B' = E \text{ in Basis} \quad \langle B + B' \vdash \text{topdec} \Rightarrow B'' \rangle}{B \vdash \text{strdec} \langle \text{topdec} \rangle \Rightarrow B' \langle ' \rangle} \quad (184)$$

$$\frac{\text{Inter } B \vdash \text{sigdec} \Rightarrow G \quad B' = G \text{ in Basis} \quad \langle B + B' \vdash \text{topdec} \Rightarrow B'' \rangle}{B \vdash \text{sigdec} \langle \text{topdec} \rangle \Rightarrow B' \langle ' \rangle} \quad (185)$$

$$\frac{B \vdash \text{fundec} \Rightarrow F \quad B' = F \text{ in Basis} \quad \langle B + B' \vdash \text{topdec} \Rightarrow B'' \rangle}{B \vdash \text{fundec} \langle \text{topdec} \rangle \Rightarrow B' \langle ' \rangle} \quad (186)$$

8 Programs

The phrase class Program of programs is defined as follows

$$program ::= topdec ; \langle program \rangle$$

Hitherto, the semantic rules have not exposed the interactive nature of the language. During an ML session the user can type in a phrase, more precisely a phrase of the form *topdec* as defined in Figure 8, page 15. Upon the following semicolon, the machine will then attempt to parse, elaborate and evaluate the phrase returning either a result or, if any of the phases fail, an error message. The outcome is significant for what the user subsequently types, so we need to answer questions such as: if the elaboration of a top-level declaration succeeds, but its evaluation fails, then does the result of the elaboration get recorded in the static basis?

In practice, ML implementations may provide a directive as a form of top-level declaration for including programs from files rather than directly from the terminal. In case a file consists of a sequence of top-level declarations (separated by semicolons) and the machine detects an error in one of these, it is probably sensible to abort the execution of the directive. Rather than introducing a distinction between, say, batch programs and interactive programs, we shall tacitly regard all programs as interactive, and leave to implementers to clarify how the inclusion of files, if provided, affects the updating of the static and dynamic basis. Moreover, we shall focus on elaboration and evaluation and leave the handling of parse errors to implementers (since it naturally depends on the kind of parser being employed). Hence, in this section the *execution* of a program means the combined elaboration and evaluation of the program.

So far, for simplicity, we have used the same notation B to stand for both a static and a dynamic basis, and this has been possible because we have never needed to discuss static and dynamic semantics at the same time. In giving the semantics of programs, however, let us rename as `StaticBasis` the class `Basis` defined in the static semantics of modules, Section 5.1, and let us use B_{STAT} to range over `StaticBasis`. Similarly, let us rename as `DynamicBasis` the class `Basis` defined in the dynamic semantics of modules, Section 7.2, and let us use B_{DYN} to range over `DynamicBasis`. We now define

$$B \text{ or } (B_{\text{STAT}}, B_{\text{DYN}}) \in \text{Basis} = \text{StaticBasis} \times \text{DynamicBasis}.$$

Further, we shall use \vdash_{STAT} for elaboration as defined in Section 5, and \vdash_{DYN} for evaluation as defined in Section 7. Then \vdash will be reserved for the execution of programs, which thus is expressed by a sentence of the form

$$s, B \vdash \text{program} \Rightarrow B', s'$$

This may be read as follows: starting in basis B with state s the execution of *program* results in a basis B' and a state s' .

It must be understood that executing a program never results in an exception. If the evaluation of a *topdec* yields an exception (for instance because of a **raise** expression) then the result of executing the program “*topdec* ;” is the original basis together with the state which is in force when the exception is generated. In particular, the exception convention of Section 6.7 is not applicable to the ensuing rules.

We represent the non-elaboration of a top-level declaration by $\dots \vdash_{\text{STAT}} \text{topdec} \not\Rightarrow$.

Programs

$$\boxed{s, B \vdash \text{program} \Rightarrow B', s'}$$

$$\frac{B_{\text{STAT}} \text{ of } B \vdash_{\text{STAT}} \text{topdec} \not\Rightarrow \quad \langle s, B \vdash \text{program} \Rightarrow B', s' \rangle}{s, B \vdash \text{topdec} ; \langle \text{program} \rangle \Rightarrow B'\langle' \rangle, s'\langle' \rangle} \quad (187)$$

$$\frac{B_{\text{STAT}} \text{ of } B \vdash_{\text{STAT}} \text{topdec} \Rightarrow B_{\text{STAT}}^{(1)} \quad s, B_{\text{DYN}} \text{ of } B \vdash_{\text{DYN}} \text{topdec} \Rightarrow p, s' \quad \langle s', B \vdash \text{program} \Rightarrow B', s'' \rangle}{s, B \vdash \text{topdec} ; \langle \text{program} \rangle \Rightarrow B'\langle' \rangle, s'\langle' \rangle} \quad (188)$$

$$\frac{B_{\text{STAT}} \text{ of } B \vdash_{\text{STAT}} \text{topdec} \Rightarrow B_{\text{STAT}}^{(1)} \quad s, B_{\text{DYN}} \text{ of } B \vdash_{\text{DYN}} \text{topdec} \Rightarrow B_{\text{DYN}}^{(1)}, s' \quad B' = B \oplus (B_{\text{STAT}}^{(1)}, B_{\text{DYN}}^{(1)}) \quad \langle s', B' \vdash \text{program} \Rightarrow B'', s'' \rangle}{s, B \vdash \text{topdec} ; \langle \text{program} \rangle \Rightarrow B'\langle' \rangle, s'\langle' \rangle} \quad (189)$$

Comments:

(187) A failing elaboration has no effect whatever.

(188) An evaluation which yields an exception nullifies the change in the static basis, but does not nullify side-effects on the state which may have occurred before the exception was raised.

Core language Programs

A program is called a *core language program* if it can be parsed in the reduced grammar defined as follows:

1. Replace the definition of top-level declarations by

$$\textit{topdec} ::= \textit{strdec}$$

2. Replace the definition of structure-level declarations by

$$\textit{strdec} ::= \textit{dec}$$

A Appendix: Derived Forms

Several derived grammatical forms are provided in the Core; they are presented in Figures 15, 16 and 17. Each derived form is given with its equivalent form. Thus, each row of the tables should be considered as a rewriting rule

Derived form \implies Equivalent form

and these rules may be applied repeatedly to a phrase until it is transformed into a phrase of the bare language. See Appendix B for the full Core grammar, including all the derived forms.

In the derived forms for tuples, in terms of records, we use \bar{n} to mean the ML numeral which stands for the natural number n .

Note that a new phrase class `FvalBind` of function-value bindings is introduced, accompanied by a new declaration form `fun tyvarseq fvalbind`. The mixed forms `val tyvarseq rec fvalbind`, `val tyvarseq fvalbind` and `fun tyvarseq valbind` are not allowed – though the first form arises during translation into the bare language.

The following notes refer to Figure 17:

- There is a version of the derived form for function-value binding which allows the function identifier to be infix; see Figure 21 in Appendix B.
- In the two forms involving `withtype`, the identifiers bound by `datbind` and by `typbind` must be distinct. Then the transformed binding `datbind'` in the equivalent form is obtained from `datbind` by expanding out all the definitions made by `typbind`. More precisely, if `typbind` is

$$tyvarseq_1 tycon_1 = ty_1 \text{ and } \dots \text{ and } tyvarseq_n tycon_n = ty_n$$

then `datbind'` is the result of simultaneous replacement (in `datbind`) of every type expression `tyseqi tyconi` ($1 \leq i \leq n$) by the corresponding defining expression

$$ty_i\{tyseq_i/tyvarseq_i\}$$

Figure 18 shows derived forms for functors. They allow functors to take, say, a single type or value as a parameter, in cases where it would seem clumsy to “wrap up” the argument as a structure expression.

Finally, Figure 19 shows the derived forms for specifications and signature expressions. The last derived form for specifications allows sharing

between structure identifiers as a shorthand for type sharing specifications.

The phrase

$$\textit{spec sharing } longstrid_1 = \dots = longstrid_k$$

is a derived form whose equivalent form is

```

spec
  sharing type longtycon1 = longtycon'1
  ...
  sharing type longtyconm = longtycon'm

```

determined as follows. First, note that *spec* specifies a set of (possibly long) type constructors and structure identifiers, either directly or via signature identifiers and `include` specifications. Then the equivalent form contains all type-sharing constraints of the form

$$\text{sharing type } \textit{longstrid}_i.\textit{longtycon} = \textit{longstrid}_j.\textit{longtycon}$$

($1 \leq i < j \leq k$), such that both sides of the equation are long type constructors specified by *spec*.

The meaning of the derived form does not depend on the order of the type-sharing constraints in the equivalent form.

Derived Form	Equivalent Form	
Expressions <i>exp</i>		
$()$	$\{ \}$	
(exp_1, \dots, exp_n)	$\{1=exp_1, \dots, \bar{n}=exp_n\}$	$(n \geq 2)$
$\# lab$	$fn \{lab=vid, \dots\} \Rightarrow vid$	$(vid \text{ new})$
$case \ exp \ \text{of} \ match$	$(fn \ match)(exp)$	
$if \ exp_1 \ \text{then} \ exp_2 \ \text{else} \ exp_3$	$case \ exp_1 \ \text{of} \ true \Rightarrow \ exp_2$ $\quad \quad \quad \ false \Rightarrow \ exp_3$	
$exp_1 \ \text{orelse} \ exp_2$	$if \ exp_1 \ \text{then} \ true \ \text{else} \ exp_2$	
$exp_1 \ \text{andalso} \ exp_2$	$if \ exp_1 \ \text{then} \ exp_2 \ \text{else} \ false$	
$(exp_1 ; \dots ; exp_n ; exp)$	$case \ exp_1 \ \text{of} \ (_) \Rightarrow$ $\quad \dots$ $case \ exp_n \ \text{of} \ (_) \Rightarrow \ exp$	$(n \geq 1)$
$let \ dec \ \text{in}$ $\quad \quad \quad exp_1 ; \dots ; exp_n \ \text{end}$	$let \ dec \ \text{in}$ $\quad \quad \quad (exp_1 ; \dots ; exp_n) \ \text{end}$	$(n \geq 2)$
$while \ exp_1 \ \text{do} \ exp_2$	$let \ \text{val} \ rec \ vid = fn \ () \Rightarrow$ $\quad \quad \quad if \ exp_1 \ \text{then} \ (exp_2 ; vid()) \ \text{else} \ ()$ $\quad \quad \quad in \ vid() \ \text{end}$	$(vid \ \text{new})$
$[exp_1, \dots, exp_n]$	$exp_1 :: \dots :: exp_n :: nil$	$(n \geq 0)$

Figure 15: Derived forms of Expressions

Derived Form	Equivalent Form	
Patterns <i>pat</i>		
$()$	$\{ \}$	
(pat_1, \dots, pat_n)	$\{1=pat_1, \dots, \bar{n}=pat_n\}$	$(n \geq 2)$
$[pat_1, \dots, pat_n]$	$pat_1 :: \dots :: pat_n :: nil$	$(n \geq 0)$
Pattern Rows <i>patrow</i>		
$vid\langle :ty \rangle \langle \text{as } pat \rangle \langle , \ patrow \rangle$	$vid = vid\langle :ty \rangle \langle \text{as } pat \rangle \langle , \ patrow \rangle$	
Type Expressions <i>ty</i>		
$ty_1 * \dots * ty_n$	$\{1:ty_1, \dots, \bar{n}:ty_n\}$	$(n \geq 2)$

Figure 16: Derived forms of Patterns and Type Expressions

Derived Form	Equivalent Form
Function-value Bindings <i>fvalbind</i>	
$\langle \text{op} \rangle \text{vid } \text{atpat}_{11} \cdots \text{atpat}_{1n} \langle : \text{ty} \rangle = \text{exp}_1$ $ \langle \text{op} \rangle \text{vid } \text{atpat}_{21} \cdots \text{atpat}_{2n} \langle : \text{ty} \rangle = \text{exp}_2$ $ \quad \dots \quad \dots$ $ \langle \text{op} \rangle \text{vid } \text{atpat}_{m1} \cdots \text{atpat}_{mn} \langle : \text{ty} \rangle = \text{exp}_m$ $\langle \text{and } fvalbind \rangle$	$\langle \text{op} \rangle \text{vid} = \text{fn } \text{vid}_1 \Rightarrow \dots \text{fn } \text{vid}_n \Rightarrow$ $\text{case } (\text{vid}_1, \dots, \text{vid}_n) \text{ of}$ $\quad (\text{atpat}_{11}, \dots, \text{atpat}_{1n}) \Rightarrow \text{exp}_1 \langle : \text{ty} \rangle$ $ (\text{atpat}_{21}, \dots, \text{atpat}_{2n}) \Rightarrow \text{exp}_2 \langle : \text{ty} \rangle$ $ \quad \dots \quad \dots$ $ (\text{atpat}_{m1}, \dots, \text{atpat}_{mn}) \Rightarrow \text{exp}_m \langle : \text{ty} \rangle$ $\langle \text{and } fvalbind \rangle$
($m, n \geq 1$; $\text{vid}_1, \dots, \text{vid}_n$ distinct and new)	
Declarations <i>dec</i>	
$\text{fun } \text{tyvarseq } fvalbind$	$\text{val } \text{tyvarseq } \text{rec } fvalbind$
$\text{datatype } \text{datbind } \text{withtype } \text{typbind}$	$\text{datatype } \text{datbind}' ; \text{type } \text{typbind}$
$\text{abstype } \text{datbind } \text{withtype } \text{typbind}$ $\quad \text{with } \text{dec } \text{end}$	$\text{abstype } \text{datbind}'$ $\quad \text{with type } \text{typbind} ; \text{dec } \text{end}$
(see note in text concerning $\text{datbind}'$)	

Figure 17: Derived forms of Function-value Bindings and Declarations

Derived Form	Equivalent Form
Structure Bindings <i>strbind</i>	
<i>strid</i> : <i>sigexp</i> = <i>strex</i> <and <i>strbind</i> >	<i>strid</i> = <i>strex</i> : <i>sigexp</i> <and <i>strbind</i> >
<i>strid</i> : > <i>sigexp</i> = <i>strex</i> <and <i>strbind</i> >	<i>strid</i> = <i>strex</i> : > <i>sigexp</i> <and <i>strbind</i> >
Structure Expressions <i>strex</i>	
<i>funid</i> (<i>strdec</i>)	<i>funid</i> (struct <i>strdec</i> end)
Functor Bindings <i>funbind</i>	
<i>funid</i> (<i>strid</i> : <i>sigexp</i>) : <i>sigexp'</i> = <i>strex</i> <and <i>funbind</i> >	<i>funid</i> (<i>strid</i> : <i>sigexp</i>) = <i>strex</i> : <i>sigexp'</i> <and <i>funbind</i> >
<i>funid</i> (<i>strid</i> : <i>sigexp</i>) : > <i>sigexp'</i> = <i>strex</i> <and <i>funbind</i> >	<i>funid</i> (<i>strid</i> : <i>sigexp</i>) = <i>strex</i> : > <i>sigexp'</i> <and <i>funbind</i> >
<i>funid</i> (<i>spec</i>) <: <i>sigexp</i> > = <i>strex</i> <and <i>funbind</i> >	<i>funid</i> (<i>strid</i> _v : sig <i>spec</i> end) = let open <i>strid</i> _v in <i>strex</i> <: <i>sigexp</i> > end <and <i>funbind</i> >
<i>funid</i> (<i>spec</i>) <:> <i>sigexp</i> > = <i>strex</i> <and <i>funbind</i> >	<i>funid</i> (<i>strid</i> _v : sig <i>spec</i> end) = let open <i>strid</i> _v in <i>strex</i> <:> <i>sigexp</i> > end <and <i>funbind</i> >
	(<i>strid</i> _v , new)
Programs <i>program</i>	
<i>exp</i> ; < <i>program</i> >	val it = <i>exp</i> ; < <i>program</i> >

Figure 18: Derived forms of Functors, Structure Bindings and Programs

B Appendix: Full Grammar

The full grammar of programs is exactly as given at the start of Section 8.

The full grammar of Modules consists of the grammar of Figures 5–8 in Section 3, together with the derived forms of Figures 18 and 19 in Appendix A.

The remainder of this Appendix is devoted to the full grammar of the Core. Roughly, it consists of the grammar of Section 2 augmented by the derived forms of Appendix A. But there is a further difference: two additional subclasses of the phrase class `Exp` are introduced, namely `AppExp` (application expressions) and `InfExp` (infix expressions). The inclusion relation among the four classes is as follows:

$$\text{AtExp} \subset \text{AppExp} \subset \text{InfExp} \subset \text{Exp}$$

The effect is that certain phrases, such as “`2 + while ... do ...`”, are now disallowed.

The grammatical rules are displayed in Figures 20, 21, 22 and 23. The grammatical conventions are exactly as in Section 2, namely:

- The brackets `< >` enclose optional phrases.
- For any syntax class `X` (over which x ranges) we define the syntax class `Xseq` (over which $xseq$ ranges) as follows:

$$\begin{aligned} xseq ::= x & \quad (\text{singleton sequence}) \\ & \quad (\text{empty sequence}) \\ & \quad (x_1, \dots, x_n) \quad (\text{sequence, } n \geq 1) \end{aligned}$$

(Note that the “...” used here, a meta-symbol indicating syntactic repetition, must not be confused with “...” which is a reserved word of the language.)

- Alternative forms for each phrase class are in order of decreasing precedence. This precedence resolves ambiguity in parsing in the following way. Suppose that a phrase class — we take `exp` as an example — has two alternative forms F_1 and F_2 , such that F_1 ends with an `exp` and F_2 starts with an `exp`. A specific case is

$$\begin{aligned} F_1: & \text{ if } exp_1 \text{ then } exp_2 \text{ else } exp_3 \\ F_2: & exp \text{ handle } match \end{aligned}$$

It will be enough to see how ambiguity is resolved in this specific case. Suppose that the lexical sequence

... .. if ... then ... else *exp* handle

is to be parsed, where *exp* stands for a lexical sequence which is already determined as a subphrase (if necessary by applying the precedence rule). Then the higher precedence of F_2 (in this case) dictates that *exp* associates to the right, i.e. that the correct parse takes the form

... .. if ... then ... else (*exp* handle ...) ...

not the form

... (... if ... then ... else *exp*) handle

Note particularly that the use of precedence does not decrease the class of admissible phrases; it merely rejects alternative ways of parsing certain phrases. In particular, the purpose is not to prevent a phrase, which is an instance of a form with higher precedence, having a constituent which is an instance of a form with lower precedence. Thus for example

if ... then while ... do ... else while ... do ...

is quite admissible, and will be parsed as

if ... then (while ... do ...) else (while ... do ...)

- L (resp. R) means left (resp. right) association.
- The syntax of types binds more tightly than that of expressions.
- Each iterated construct (e.g. *match*, ...) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. “**fn** *match*”, if this occurs within a larger match.

<i>atexp</i>	<i>::=</i>	<i>scon</i> <i><op>longvid</i> <i>{ <exprow > }</i> <i># lab</i> <i>()</i> <i>(exp₁ , … , exp_n)</i> <i>[exp₁ , … , exp_n]</i> <i>(exp₁ ; … ; exp_n)</i> <i>let dec in exp₁ ; … ; exp_n end</i> <i>(exp)</i>	special constant value identifier record record selector 0-tuple <i>n</i> -tuple, <i>n</i> ≥ 2 list, <i>n</i> ≥ 0 sequence, <i>n</i> ≥ 2 local declaration, <i>n</i> ≥ 1
<i>exprow</i>	<i>::=</i>	<i>lab = exp < , exprow ></i>	expression row
<i>appexp</i>	<i>::=</i>	<i>atexp</i> <i>appexp atexp</i>	application expression
<i>infxp</i>	<i>::=</i>	<i>appexp</i> <i>infxp₁ vid infxp₂</i>	infix expression
<i>exp</i>	<i>::=</i>	<i>infxp</i> <i>exp : ty</i> <i>exp₁ andalso exp₂</i> <i>exp₁ or else exp₂</i> <i>exp handle match</i> <i>raise exp</i> <i>if exp₁ then exp₂ else exp₃</i> <i>while exp₁ do exp₂</i> <i>case exp of match</i> <i>fn match</i>	typed (L) conjunction disjunction handle exception raise exception conditional iteration case analysis function
<i>match</i>	<i>::=</i>	<i>mrule < match ></i>	
<i>mrule</i>	<i>::=</i>	<i>pat => exp</i>	

Figure 20: Grammar: Expressions and Matches

<i>dec</i>	::= val <i>tyvarseq valbind</i> fun <i>tyvarseq fvalbind</i> type <i>typbind</i> datatype <i>datbind</i> ⟨withtype <i>typbind</i> ⟩ datatype <i>tycon</i> ==- datatype <i>longtycon</i> abstype <i>datbind</i> ⟨withtype <i>typbind</i> ⟩ with <i>dec</i> end exception <i>exbind</i> local <i>dec</i> ₁ in <i>dec</i> ₂ end open <i>longstrid</i> ₁ ⋯ <i>longstrid</i> _{<i>n</i>} <i>dec</i> ₁ ⟨;⟩ <i>dec</i> ₂ infix ⟨ <i>d</i> ⟩ <i>vid</i> ₁ ⋯ <i>vid</i> _{<i>n</i>} infixr ⟨ <i>d</i> ⟩ <i>vid</i> ₁ ⋯ <i>vid</i> _{<i>n</i>} nonfix <i>vid</i> ₁ ⋯ <i>vid</i> _{<i>n</i>}	value declaration function declaration type declaration datatype declaration datatype replication abstype declaration exception declaration local declaration open declaration, <i>n</i> ≥ 1 empty declaration sequential declaration infix (L) directive, <i>n</i> ≥ 1 infix (R) directive, <i>n</i> ≥ 1 nonfix directive, <i>n</i> ≥ 1
<i>valbind</i>	::= <i>pat</i> = <i>exp</i> ⟨and <i>valbind</i> ⟩ rec <i>valbind</i>	
<i>fvalbind</i>	::= ⟨op⟩ <i>vid atpat</i> ₁₁ ⋯ <i>atpat</i> _{1<i>n</i>} ⟨: <i>ty</i> ⟩= <i>exp</i> ₁ ⟨op⟩ <i>vid atpat</i> ₂₁ ⋯ <i>atpat</i> _{2<i>n</i>} ⟨: <i>ty</i> ⟩= <i>exp</i> ₂ ⋯ ⋯ ⟨op⟩ <i>vid atpat</i> _{<i>m</i>1} ⋯ <i>atpat</i> _{<i>m</i><i>n</i>} ⟨: <i>ty</i> ⟩= <i>exp</i> _{<i>m</i>} ⟨and <i>fvalbind</i> ⟩	<i>m, n</i> ≥ 1 See also note below
<i>typbind</i>	::= <i>tyvarseq tycon</i> = <i>ty</i> ⟨and <i>typbind</i> ⟩	
<i>datbind</i>	::= <i>tyvarseq tycon</i> = <i>conbind</i> ⟨and <i>datbind</i> ⟩	
<i>conbind</i>	::= ⟨op⟩ <i>vid</i> ⟨of <i>ty</i> ⟩ ⟨ <i>conbind</i> ⟩	
<i>exbind</i>	::= ⟨op⟩ <i>vid</i> ⟨of <i>ty</i> ⟩ ⟨and <i>exbind</i> ⟩ ⟨op⟩ <i>vid</i> = ⟨op⟩ <i>longvid</i> ⟨and <i>exbind</i> ⟩	

Note: In the *fvalbind* form, if *vid* has infix status then either *op* must be present, or *vid* must be infix. Thus, at the start of any clause, “*op vid (atpat, atpat′) ⋯*” may be written “*(atpat vid atpat′) ⋯*”; the parentheses may also be dropped if “: *ty*” or “=” follows immediately.

Figure 21: Grammar: Declarations and Bindings

$atpat$	$::=$	-	wildcard
		$scon$	special constant
		$\langle op \rangle longvid$	value identifier
		$\{ \langle patrow \rangle \}$	record
		$()$	0-tuple
		(pat_1, \dots, pat_n)	n -tuple, $n \geq 2$
		$[pat_1, \dots, pat_n]$	list, $n \geq 0$
		(pat)	
$patrow$	$::=$	\dots	wildcard
		$lab = pat \langle , patrow \rangle$	pattern row
		$vid \langle : ty \rangle \langle as pat \rangle \langle , patrow \rangle$	label as variable
pat	$::=$	$atpat$	atomic
		$\langle op \rangle longvid atpat$	constructed value
		$pat_1 vid pat_2$	constructed value (infix)
		$pat : ty$	typed
		$\langle op \rangle vid \langle : ty \rangle as pat$	layered

Figure 22: Grammar: Patterns

ty	$::=$	$tyvar$	type variable
		$\{ \langle tyrow \rangle \}$	record type expression
		$tyseq longtycon$	type construction
		$ty_1 * \dots * ty_n$	tuple type, $n \geq 2$
		$ty \rightarrow ty'$	function type expression (R)
		(ty)	
$tyrow$	$::=$	$lab : ty \langle , tyrow \rangle$	type-expression row

Figure 23: Grammar: Type expressions

C Appendix: The Initial Static Basis

In this appendix (and the next) we define a minimal initial basis for execution. Richer bases may be provided by libraries. We shall indicate components of the initial basis by the subscript 0. The initial static basis is $B_0 = T_0, F_0, G_0, E_0$, where $F_0 = \{\}$, $G_0 = \{\}$ and

$$T_0 = \{\text{bool}, \text{int}, \text{real}, \text{string}, \text{char}, \text{word}, \text{list}, \text{ref}, \text{exn}\}$$

The members of T_0 are type names, not type constructors; for convenience we have used type-constructor identifiers to stand also for the type names which are bound to them in the initial static type environment TE_0 . Of these type names, `list` and `ref` have arity 1, the rest have arity 0; all except `exn` and `real` admit equality. Finally, $E_0 = (SE_0, TE_0, VE_0)$, where $SE_0 = \{\}$, while TE_0 and VE_0 are shown in Figures 24 and 25, respectively.

<i>tycon</i>	\mapsto	$(\theta, \{vid_1 \mapsto (\sigma_1, is_1), \dots, vid_n \mapsto (\sigma_n, is_n)\})$	$(n \geq 0)$
<code>unit</code>	\mapsto	$(\Lambda().\{\}, \{\})$	
<code>bool</code>	\mapsto	$(\text{bool}, \{\text{true} \mapsto (\text{bool}, c), \text{false} \mapsto (\text{bool}, c)\})$	
<code>int</code>	\mapsto	$(\text{int}, \{\})$	
<code>word</code>	\mapsto	$(\text{word}, \{\})$	
<code>real</code>	\mapsto	$(\text{real}, \{\})$	
<code>string</code>	\mapsto	$(\text{string}, \{\})$	
<code>char</code>	\mapsto	$(\text{char}, \{\})$	
<code>list</code>	\mapsto	$(\text{list}, \{\text{nil} \mapsto (\forall 'a . 'a \text{ list}, c),$ $:: \mapsto (\forall 'a . 'a * 'a \text{ list} \rightarrow 'a \text{ list}, c)\})$	
<code>ref</code>	\mapsto	$(\text{ref}, \{\text{ref} \mapsto (\forall 'a . 'a \rightarrow 'a \text{ ref}, c)\})$	
<code>exn</code>	\mapsto	$(\text{exn}, \{\})$	

Figure 24: Static TE_0

NONFIX	INFIX
$vid \mapsto (\sigma, is)$	$vid \mapsto (\sigma, is)$
ref $\mapsto (\forall 'a . 'a \rightarrow 'a \text{ ref}, c)$	Precedence 5, right associative :
nil $\mapsto (\forall 'a . 'a \text{ list}, c)$	$:: \mapsto (\forall 'a . 'a * 'a \text{ list} \rightarrow 'a \text{ list}, c)$
true $\mapsto (\text{bool}, c)$	Precedence 4, left associative :
false $\mapsto (\text{bool}, c)$	$= \mapsto (\forall 'a . ''a * ''a \rightarrow \text{bool}, v)$
Match $\mapsto (\text{exn}, e)$	Precedence 3, left associative :
Bind $\mapsto (\text{exn}, e)$	$:= \mapsto (\forall 'a . 'a \text{ ref} * 'a \rightarrow \{\}, v)$

Note: In type schemes we have taken the liberty of writing $ty_1 * ty_2$ in place of $\{1 \mapsto ty_1, 2 \mapsto ty_2\}$.

Figure 25: Static VE_0

D Appendix: The Initial Dynamic Basis

We shall indicate components of the initial basis by the subscript 0. The initial dynamic basis is $B_0 = F_0, G_0, E_0$, where $F_0 = \{\}$, $G_0 = \{\}$ and $E_0 = (SE_0, TE_0, VE_0)$, where $SE_0 = \{\}$, TE_0 is shown in Figure 26 and

$$VE_0 = \{= \mapsto (=, v), := \mapsto (:=, v), \text{Match} \mapsto (\text{Match}, e), \text{Bind} \mapsto (\text{Bind}, e), \\ \text{true} \mapsto (\text{true}, c), \text{false} \mapsto (\text{false}, c), \\ \text{nil} \mapsto (\text{nil}, c), :: \mapsto (::, c), \text{ref} \mapsto (\text{ref}, c)\}.$$

<i>tycon</i>	\mapsto	$\{vid_1 \mapsto (v_1, is_1), \dots, vid_n \mapsto (v_n, is_n)\} \quad (n \geq 0)$
<code>unit</code>	\mapsto	$\{\}$
<code>bool</code>	\mapsto	$\{\text{true} \mapsto (\text{true}, c), \text{false} \mapsto (\text{false}, c)\}$
<code>int</code>	\mapsto	$\{\}$
<code>word</code>	\mapsto	$\{\}$
<code>real</code>	\mapsto	$\{\}$
<code>string</code>	\mapsto	$\{\}$
<code>char</code>	\mapsto	$\{\}$
<code>list</code>	\mapsto	$\{\text{nil} \mapsto (\text{nil}, c), :: \mapsto (::, c)\}$
<code>ref</code>	\mapsto	$\{\text{ref} \mapsto (\text{ref}, c)\}$
<code>exn</code>	\mapsto	$\{\}$

Figure 26: Dynamic TE_0

E Overloading

Two forms of overloading are available:

- Certain special constants are overloaded. For example, `0w5` may have type `word` or some other type, depending on the surrounding program text;
- Certain operators are overloaded. For example, `+` may have type `int * int → int` or `real * real → real`, depending on the surrounding program text;

Programmers cannot define their own overloaded constants or operators.

Although a formal treatment of overloading is outside the scope of this document, we do give a complete list of the overloaded operators and of types with overloaded special constants. This list is consistent with the Basis Library[18].

Every overloaded constant and value identifier has among its types a *default type*, which is assigned to it, when the surrounding text does not resolve the overloading. For this purpose, the surrounding text is no larger than the smallest enclosing structure-level declaration; an implementation may require that a smaller context determines the type.

E.1 Overloaded special constants

Libraries may extend the set T_0 of Appendix C with additional type names. Thereafter, certain subsets of T_0 have a special significance; they are called *overloading classes* and they are:

<code>Int</code>	\supseteq	<code>{int}</code>
<code>Real</code>	\supseteq	<code>{real}</code>
<code>Word</code>	\supseteq	<code>{word}</code>
<code>String</code>	\supseteq	<code>{string}</code>
<code>Char</code>	\supseteq	<code>{char}</code>
<code>WordInt</code>	$=$	<code>Word</code> \cup <code>Int</code>
<code>RealInt</code>	$=$	<code>Real</code> \cup <code>Int</code>
<code>Num</code>	$=$	<code>Word</code> \cup <code>Real</code> \cup <code>Int</code>
<code>NumTxt</code>	$=$	<code>Word</code> \cup <code>Real</code> \cup <code>Int</code> \cup <code>String</code> \cup <code>Char</code>

Among these, the five first (`Int`, `Real`, `Word`, `String` and `Char`) are said to be *basic*; the remaining are said to be *composite*. The reason that the basic

NONFIX	INFIX
<i>var</i> \mapsto set of monotypes	<i>var</i> \mapsto set of monotypes
abs \mapsto realint \rightarrow realint	Precedence 7, left associative :
\sim \mapsto realint \rightarrow realint	div \mapsto wordint * wordint \rightarrow wordint
	mod \mapsto wordint * wordint \rightarrow wordint
	* \mapsto num * num \rightarrow num
	/ \mapsto Real * Real \rightarrow Real
	Precedence 6, left associative :
	+ \mapsto num * num \rightarrow num
	- \mapsto num * num \rightarrow num
	Precedence 4, left associative :
	< \mapsto numtxt * numtxt \rightarrow numtxt
	> \mapsto numtxt * numtxt \rightarrow numtxt
	<= \mapsto numtxt * numtxt \rightarrow numtxt
	>= \mapsto numtxt * numtxt \rightarrow numtxt

Figure 27: Overloaded identifiers

classes are specified using \supseteq rather than $=$ is that libraries may extend each of the basic overloading classes with further type names. Special constants are overloaded within each of the basic overloading classes. However, the basic overloading classes must be arranged so that every special constant can be assigned types from at most one of the basic overloading classes. For example, `0w5` may be assigned type `word`, or some other member of `Word`, depending on the surrounding text. If the surrounding text does not determine the type of the constant, a default type is used. The default types for the five sets are `int`, `real`, `word`, `string` and `char` respectively.

Once overloading resolution has determined the type of a special constant, it is a compile-time error if the constant does not make sense or does not denote a value within the machine representation chosen for the type. For example, an escape sequence of the form `\uxxxx` in a string constant of 8-bit characters only makes sense if `xxxx` denotes a number in the range $[0, 255]$.

E.2 Overloaded value identifiers

Overloaded identifiers all have identifier status `v`. An overloaded identifier may be re-bound with any status (`v`, `c` and `e`) but then it is not overloaded within the scope of the binding.

The overloaded identifiers are given in Figure 27. For example, the entry

$$\mathbf{abs} \mapsto \mathbf{realint} \rightarrow \mathbf{realint}$$

states that **abs** may assume one of the types $\{t \rightarrow t \mid t \in \mathbf{RealInt}\}$. In general, the same type name must be chosen throughout the entire type of the overloaded operator; thus **abs** does not have type $\mathbf{real} \rightarrow \mathbf{int}$.

The operator $/$ is overloaded on all members of **Real**, with default type $\mathbf{real} * \mathbf{real} \rightarrow \mathbf{real}$. The default type of any other identifier is that one of its types which contains the type name **int**. For example, the program `fun double(x) = x + x;` declares a function of type $\mathbf{int} * \mathbf{int} \rightarrow \mathbf{int}$, while `fun double(x:real) = x + x;` declares a function of type $\mathbf{real} * \mathbf{real} \rightarrow \mathbf{real}$.

The dynamic semantics of the overloaded operators is defined in [18].

F Appendix: The Development of ML

This Appendix records the main stages in the development of ML, and the people principally involved. The main emphasis is upon the design of the language; there is also a section devoted to implementation. On the other hand, no attempt is made to record work on applications of the language.

Origins

ML and its semantic description have evolved over a period of about twenty years. It is a fusion of many ideas from many people; in this appendix we try to record and to acknowledge the important precursors of its ideas, the important influences upon it, and the important contributions to its design, implementation and semantic description.

ML, which stands for *meta language*, was conceived as a medium for finding and performing proofs in a formal logical system. This application was the focus of the initial design effort, by Robin Milner in collaboration first with Malcolm Newey and Lockwood Morris, then with Michael Gordon and Christopher Wadsworth [20]. The intended application to proof affected the design considerably. Higher order functions in full generality seemed necessary for programming proof tactics and strategies, and also a robust type system (see below). At the same time, imperative features were important for practical reasons; no-one had experience of large useful programs written in a pure functional style. In particular, an exception-raising mechanism was highly desirable for the natural presentation of tactics.

The full definition of this first version of ML was included in a book [19] which describes LCF, the proof system which ML was designed to support. The details of how the proof application exerted an influence on design is reported by Milner [39]. Other early influences were the applicative languages already in use in Artificial Intelligence, principally LISP [36], ISWIM [28] and POP2 [10].

Polymorphic types

The polymorphic type discipline and the associated type-assignment algorithm were prompted by the need for security; it is vital to know that when a program

produces an object which it claims to be a theorem, then it is indeed a theorem. A type discipline provides the security, but a polymorphic discipline also permits considerable flexibility.

The key ideas of the type discipline were evolved in combinatory logic by Haskell Curry and Roger Hindley, who arrived at different but equivalent algorithms for computing principal type schemes. Curry's [14] algorithm was by equation-solving; Hindley [26] used the unification algorithm of Alan Robinson [48] and also presented the precursor of our type inference system. James Morris [43] independently gave an equation-solving algorithm very similar to Curry's. The idea of an algorithm for finding principal type schemes is very natural and may well have been known earlier. Roger Hindley has pointed out that Carew Meredith's inference rule for propositional logic called Condensed Detachment, defined in the early 1950s, clearly suggests that he knew such an algorithm [37].

Milner [38], during the design of ML, rediscovered principal types and their calculation by unification, for a language (slightly richer than combinatory logic) containing local declarations. He and Damas [15] presented the ML type inference systems following Hindley's style. Damas [16], using ideas from Michael Gordon, also devised the first mathematical treatment of polymorphism in the presence of references and assignment. Tofte [54] produced a different scheme employing so-called *imperative types*, which was adopted in the original version of the language. This approach has been superseded in the present language by a simpler scheme, suggested by Tofte [54], Andrew Wright [57], and Xavier Leroy [29], according to which polymorphic bindings are restricted to non-expansive expressions.

Refinement of the Core Language

Two movements led to the re-design of ML. One was the work of Rod Burstall and his group on specifications, crystallised in the specification language CLEAR [11] and in the functional programming language HOPE [12]; the latter was for expressing executable specifications. The outcome of this work which is relevant here was twofold. First, there were elegant programming features in HOPE, particularly pattern matching and clausal function definitions; second, there were ideas on modular construction of specifications, using signatures in the interfaces. A smaller but significant movement was by Luca Cardelli, who extended the data-type repertoire in ML by adding named records and variant types.

In 1983, Milner (prompted by Bernard Sufrin) wrote the first draft of a standard form of ML attempting to unite these ideas; over the next three years it evolved into the Standard ML core language. Notable here was the harmony found among polymorphism, HOPE patterns and Cardelli records, and the nice generalisations of ML exceptions due to ideas from Alan Mycroft, Brian Monahan and Don Sannella. A simple stream-based I/O mechanism was developed from ideas of Cardelli by Milner and Harper. The Standard ML core language is described in detail in a composite report [23] which also contains a description of the I/O mechanism and MacQueen’s proposal for program modules (see later for discussion of this). Since then only few changes to the core language have occurred. Milner proposed equality types, and these were added, together with a few minor adjustments [40]. The last development before the 1990 Definition was in the exception mechanism, by MacQueen using an idea from Burstall [3]; it harmonized the ideas of exception and data type construction.

Modules

Besides contributory ideas to the core language, HOPE [12] contained a simple notion of program module. The most important and original feature of ML modules, however, stems from the work on parameterised specifications in CLEAR [11]. MacQueen, who was a member of Burstall’s group at the time, designed [34] a new parametric module feature for HOPE inspired by the CLEAR work. He later extended the parameterisation ideas by a novel method of specifying sharing of components among the structure parameters of a functor, and produced a draft design which accommodated features already present in ML – in particular the polymorphic type system. This design was discussed in detail at Edinburgh, leading to MacQueen’s first report on modules [23].

Thereafter, the design came under close scrutiny through a draft operational static semantics and prototype implementation of it by Harper, through Kevin Mitchell’s implementation of the evaluation, through a denotational semantics written by Don Sannella, and then through further work on operational semantics by Harper, Milner, and Tofte. (More is said about this in the later section on Semantics.) In all of this work the central ideas withstood scrutiny, while it also became clear that there were gaps in the design and ambiguities in interpretation. (An example of a gap was the inability to specify sharing between a functor argument structure and its

result structure; an example of an ambiguity was the question of whether sharing exists in a structure over and above what is specified in the signature expression which accompanies its declaration.)

Much discussion ensued; it was possible for a wider group to comment on modules through using Harper's prototype implementation, while Harper, Milner and Tofte gained understanding during development of this semantics. In parallel, Sannella and Tarlecki explored the implications of modules for the methodology of program development [49]. Tofte, in his thesis [53], proved several technical properties of modules in a skeletal language, which generated considerable confidence in this design. A key point in this development was the proof of the existence of principal signatures, and, in the careful distinction between the notion of *enrichment* of structures, which allows more polymorphism and more components, and *realisation* which allows more sharing.

At a meeting in Edinburgh in 1987 a choice of two designs was presented, hinging upon whether or not a functor application should coerce its actual argument to its argument signature. The meeting chose coercion, and thereafter the production of Section 5 of this report – the static semantics of modules – was a matter of detailed care. That section is undoubtedly the most original and demanding part of this semantics, just as the ideas of MacQueen upon which it is based are the most far-reaching extension to the original design of ML.

Considerable experience was gained in implementing, programming with, and teaching the language during the nearly ten years since the definition was first published. Based on this experience a number of design decisions were revisited at a meeting of the authors in Cambridge at the end of 1995. At this meeting it was decided to make several modest, but significant, changes to the language in order to simplify the semantics and to correct some shortcomings that had come to light. The most important of these changes was the replacement of the imperative type discipline by the so-called value restriction (discussed above), the elimination of structure sharing as a separate concept from type sharing, and the introduction of the closely connected mechanisms of opaque signature matching and type abbreviations in signatures. An important impetus for these changes to the modules language was the work of Leroy [30], and Harper and Lillibridge [21] on the type-theoretic interpretation of modules (described below).

Implementation

The first implementation of ML was by Malcolm Newey, Lockwood Morris and Robin Milner in 1974, for the DEC10. Later Mike Gordon and Chris Wadsworth joined; their work was mainly in specialising ML towards machine-assisted reasoning. Around 1980 Luca Cardelli implemented a version on VAX; his work was later extended by Alan Mycroft, Kevin Mitchell and John Scott. This version contained one or two new data-type features, and was based upon the *Functional Abstract Machine (FAM)*, a virtual machine which has been a considerable stimulus to later implementation. By providing a reasonably efficient implementation, this work enabled the language to be taught to students; this, in turn, prompted the idea that it could become a useful general purpose language.

In Gothenburg, an implementation was developed by Lennart Augustsson and Thomas Johnsson in 1982, using lazy evaluation rather than call-by-value; the result was called *Lazy ML* and is described in [6]. This work is part of continuing research in many places on implementation of lazy evaluation in pure functional languages. But for ML, which includes exceptions and assignment, the emphasis has been mainly upon strict evaluation (call-by-value).

In Cambridge, in the early 1980s, Larry Paulson made considerable improvements to the Edinburgh ML compiler, as part of his wider programme of improving *Edinburgh LCF* to become *Cambridge LCF* [45]. This system has supported larger proofs than the Edinburgh system, and with greater convenience; in particular, the compiled ML code ran four to five times faster.

Around the same time Gérard Huet at INRIA (Versailles) adapted ML to Maclisp on Multics, again for use in machine-assisted proof. There was close collaboration between INRIA and Cambridge in this period. ML has undergone a separate development in the group at INRIA on the *CAML* language [13]. Work on *CAML* included the development of several extensions to the core language, notably updatable fields in record types, values with dynamic types, support for lazy evaluation, and handling of embedded languages with user-defined syntax. It did not, however, include modules.

The first implementation of the Standard ML core language was by Mitchell, Mycroft and Scott at Edinburgh, around 1984. The prototype implementation of modules, before that part of the language settled down, was done in 1985-6; Mitchell dealt with evaluation, while Harper tackled the elaboration (or ‘signature checking’) which raised problems of a kind not previously en-

countered. Harper's implementation employed a form of unification that was later adopted in the static semantics of modules.

At around the same time the *Poly/ML* implementation began with a suggestion from Mike Gordon that an interesting application of Matthews' Poly language would be to implement Standard ML. Important experience was gained through Matthews' early implementation of the core language, followed by several versions of the modules language as they were devised. *Poly/ML* features arbitrary precision arithmetic, a process package, and a windowing system. Considerable experience has been gained with the compiler, notably by Larry Paulson at Cambridge and by Abstract Hardware Limited (AHL).

The *Standard ML of New Jersey (SML/NJ)* system has been in active development since 1986 [5, 2]. Initially started by David MacQueen at Bell Laboratories and Andrew Appel at Princeton University, the project has also benefited from significant contributions by Matthias Blume, Emden Gansner, Lal George, John Reppy and Zhong Shao. *SML/NJ* is a robust and complete environment for Standard ML that supports the implementation of large software systems and generates efficient code for a number of different hardware and software platforms. *SML/NJ* also serves as a laboratory for compiler research: in implementations of module systems for ML; code optimization based on continuation-passing style; efficient pattern matching; and very fast heap allocation and garbage collection. Dozens of researchers have contributed to the development of the compiler, in such areas as efficient closure representations, first-class continuations, type-directed compilation, concurrent programming, portable code generators, separate compilation, and register allocation. *SML/NJ* has also been widely used to explore extending SML with concurrency features.

In 1989, Mads Tofte, Nick Rothwell and David N. Turner started work on the *ML Kit Compiler* in Edinburgh. The *ML Kit* is a direct translation of the 1990 Definition into a collection of Standard ML modules, emphasis being on clarity rather than efficiency. During 1992 and 1993, Version 1 of the *ML Kit* was completed, mostly through the work of Nick Rothwell at Edinburgh and Lars Birkedal at DIKU[9]. In 1994, region inference was added to the *ML Kit*, by Mads Tofte. Lars Birkedal wrote a region-based C-code generator and a runtime system in C. In 1995, Martin Elsmann and Niels Hallenberg extended this work to generate native code for the HP PA-RISC architecture.

Harlequin Ltd. began the implementation of a commercial compiler in 1990. The *MLWorks* system is a fully-featured graphical programming en-

vironment, including an interactive debugger, inspector, browser, extensive profiling facilities, separate compilation and delivery, a foreign-language interface, and libraries for threads and windowing systems.

Caml Light, a lightweight reimplementaion of *CAML* released in 1991, added a simple module system in the style of Modula-2, targeted towards separate compilation of modules: structures and signatures are identified with files, functors and multiple views of a structure are not supported. These were added in the *Caml Special Light* implementation in 1995, while preserving the support for separate compilation. *Caml Special Light* and the present version of Standard ML share several important simplifications, such as the value restriction on polymorphism, type definitions in signatures, and the lack of support for structure sharing. The static semantics for *Caml Special Light* is based on the type-theoretic properties of dependent function types (functor signatures) and manifest types (type definitions in signatures) [30].

Moscow ML is an implementation of core Standard ML, created in 1994 by Sergei Romanenko in Moscow and Peter Sestoft in Copenhagen. The *Caml Light* system was used to implement the dynamic semantics, and the ML Kit guided the implementation of the static semantics. The result is a compact and robust implementation, suitable for teaching.

The *TIL (Typed Intermediate Languages)* compiler developed at Carnegie Mellon University by Greg Morrisett, David Tarditi, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee demonstrates the use of types in compilation. All but the last few stages of *TIL* are expressed as type-directed and type-preserving transforms. Types are used at run time to support unboxed, untagged data representations and natural calling conventions in the presence of variable types and garbage collection. *TIL* employs a wide variety of conventional functional language optimizations found in other SML compilers, as well as a set of loop-oriented optimizations. A description of the compiler and an analysis of its performance appears in [52].

Other currently active implementations are by Michael Hedlund at the Rutherford-Appleton Laboratory, by Robert Duncan, Simon Nichols and Aaron Sloman at the University of Sussex (*POPLOG*) and by Malcolm Newey and his group at the Australian National University.

Semantics

The description of the first version of ML [19] was informal, and in an operational style; around the same time a denotational semantics was written,

but never published, by Mike Gordon and Robin Milner. Meanwhile structured operational semantics, presented as an inference system, was gaining credence as a tractable medium. This originates with the reduction rules of λ -calculus, but was developed more widely through the work of Plotkin [47], and also by Milner. This was at first only used for dynamic semantics, but later the benefit of using inference systems for both static and dynamic semantics became apparent. This advantage was realised when Gilles Kahn and his group at INRIA were able to execute early versions of both forms of semantics for the ML core language using their Typol system [17]. The static and dynamic semantics of the core language reached a final form mostly through work by Tofte and Milner.

The modules of ML presented little difficulty as far as dynamic semantics is concerned, but the static semantics of modules was a concerted effort by several people. MacQueen's original informal description [23] was the starting point; Sannella wrote a denotational semantics for several versions, which showed that several issues had not been settled by the informal description. Robert Harper, while writing the first implementation of modules, made the first draft of the static semantics. Harper's version made clear the importance of structure names; work by Milner and Tofte introduced further ideas including realisation; thereafter a concerted effort by all three led to several suggestions for modification of the language, and a small range of alternative interpretations; these were assessed in discussion with MacQueen, and more widely with the principal users of the language, and an agreed form was reached.

Concurrently with the formulation of the Definition of Standard ML, Harper and Mitchell took up the challenge adumbrated by MacQueen [33] to find a type-theoretic interpretation of Standard ML [25]. This work led to the formulation of the XML language, an explicitly-typed λ -calculus that captured many aspects of Standard ML. Although incomplete, their approach formed the basis for a number of subsequent studies, including the work of Harper and Lillibridge [21] and Leroy [30] on the type-theoretic interpretation of modules. This work influenced the decision to revise the language, and culminated in a type-theoretic interpretation of the present language by Harper and Stone [51]. The TIL/ML compiler (described above) is based directly on this interpretation.

There is no doubt that the interaction between design and semantic description of modules has been one of the most striking phases in the entire language development, leading (in the opinion of those involved) to a high

degree of confidence both in the language and in the semantics.

Program Libraries

During 1989-1991, Dave Berry produced the first program library for Standard ML[7, 8]. The *SML/NJ* system is distributed with a rich library organised by Emden Gansner and John Reppy; this library was the starting point for the SML Basis Library . The *SML Basis Library*[18] has been developed over the past three years in a partnership between the SML/NJ effort, *MLWorks*, and *Moscow ML*. The resulting library is a much improved and extended replacement of the initial basis defined in the 1990 Definition of Standard ML.

G Appendix: What is New?

This appendix gives an overview of how the present Definition differs from the 1990 Definition of Standard ML[42]. For the purpose of this appendix, we write SML '90 for the language defined by the 1990 Definition and SML '97 for the present language. For each major change, we give its rationale and an overview of its practical implications. Also, the index may be used for locating changes.

G.1 Type Abbreviations in Signatures

There are cases of type sharing which cannot be expressed in SML '90 signatures although they arise in structures. For example, there is no SML '90 signature which precisely describes the relationship between `s` and `t` in

```
structure a =  
  struct  
    datatype s = C  
    type t = s * s  
  end
```

In SML '97, one can write *type abbreviations* in signatures, e.g.,

```
signature A =  
  sig  
    type s  
    type t = s * s  
  end
```

The need for type abbreviations in signatures was clear when SML '90 was defined. However, type abbreviations were not included since, in the presence of both structure sharing and type abbreviations, principal signatures do not exist[41] – and the SML '90 Definition depended strongly upon the notion of principal signature. Subsequently, Harper's and Lillibridge's work on translucent sums[22] and Leroy's work on modules[30] showed that, in the absence of structure sharing and certain other features of the SML '90 signatures, type abbreviations in signatures are possible. Type abbreviations in signatures were implemented by David MacQueen in *SML/NJ* 0.93 and by Xavier Leroy in *Caml Special Light*[31].

In SML '97, structure sharing has been removed (see Section G.3 below). Type abbreviations are not included directly, but they arise as a derived form, as follows. First, a new form of signature expression is allowed:

$$sigexp \text{ where type } tyvarseq \text{ longtycon} = ty$$

Here *longtycon* has to be specified by *sigexp*. The type expression *ty* may refer to type constructors which are present in the basis in which the whole signature expression is elaborated, but not to type constructors specified in *sigexp*.

The effect of the `where type` is, roughly speaking, to instantiate *longtycon* to *ty*. For example, the following sequence of declarations is legal:

```
signature SIG1 = sig type t; val x: t end;
signature SIG2 = SIG1 where type t = int*int;
structure S1: SIG1 = struct type t = real; val x = 1.0 end;
structure S2: SIG2 = struct type t = int*int; val x = (5, 7) end;
```

Next, a type abbreviation is a derived form. For example, `type u = t*t` is equivalent to `include sig type u end where type u = t*t`. In SML '97 it is allowed to include an arbitrary signature expression, not just a signature identifier.

G.2 Opaque Signature Matching

In imposing a signature on a structure, one often wants the types of the resulting structure to be “abstract” in order to hide their implementation. (Signature matching in SML '90 hides components, but does not hide type sharing.) MacQueen originally suggested and implemented an **abstraction** declaration for this purpose[32]. In the Commentary[41] it was pointed out that the issue is the semantics of matching. SML '97 provides two kinds of matching, as new forms of structure expression:

$$strexp : sigexp$$

$$strexp :> sigexp$$

The first (`:`) is the SML '90 signature matching; the second (`:>`) is *opaque matching*. Opaque matching can be applied to the result structure of a functor; thus it is more general than MacQueen's **abstraction** declaration. In CAML Special Light, all signature matching is opaque.

With opaque matching, types in the resulting structure will be abstract, to precisely the degree expressed in *sigexp*. Thus

```
signature Sig =
  sig
    type t = int
    val x: t
    type u
    val y: u
  end;
structure S1:> Sig =
  struct type t = int
        val x = 3
        type u = real
        val y = 3.0
  end
val r = S1.x + 1
```

is legal, but a subsequent declaration `val s = S1.y + 1.5` will fail to elaborate. Similarly, consider the functor declaration:

```
functor Dict(type t; val leq: t*t->int):>
  sig type u = t*t
      type 'a dict
    end =
  struct
    type u = t*t
    type 'a dict = (t * 'a) list
  end
```

When applied, `Dict` will propagate the identity of the type `t` from argument to result, but it will produce a fresh `dict` type upon each application.

Types which are specified as “abstract” in a opaque functor result signature give rise to generation of fresh type names upon each application of the functor, even if the functor body is a constant structure. For example, after the elaboration of

```
structure A = struct type t = int end
functor f():> sig type t end = A
structure B = f()
      and C = f();
```

the two types `B.t` and `C.t` are different.

G.3 Sharing

Structure sharing is a key idea in MacQueen’s original Modules design[32]. The theoretical aspects of structure sharing have been the subject of considerable research attention[24, 53, 1, 55, 35]. However, judging from experience, structure sharing is not often used in its full generality, namely to ensure identity of values. Furthermore, experience from teaching suggests that the structure sharing concept is somewhat hard to grasp. Finally, the semantic accounts of structure sharing that have been proposed are rather complicated.

The static semantics of SML ’97 has no notion of structure sharing. However, SML ’97 does provide a weaker form of structure sharing constraints, in which structure sharing is regarded as a derived form, equivalent to a collection of type sharing constraints.

G.3.1 Type Sharing

In SML ’90, a type sharing constraint `sharing type longtycon1 = ... = longtyconn` was an admissible form of specification. In SML ’97 such a constraint does not stand by itself as a specification, but may be used to qualify a specification. Thus there is a new form of specification, which we shall call a *qualified* specification:

$$spec \text{ sharing type } longtycon_1 = \dots = longtycon_n$$

Here the long type constructors have to be specified by *spec*. The type constructors may have been specified by `type`, `eqtype` or `datatype` specifications, or indirectly through signature identifiers and `include`. In order for the specification to be legal, all the type constructors must denote flexible type names. More precisely, let B be the basis in which the qualified specification is elaborated. Let us say that a type name t is *rigid (in B)* if $t \in T$ of B and that t is *flexible (in B)* otherwise. For example `int` is rigid in the initial basis and every datatype declaration introduces additional rigid type names into the basis. For the qualified specification to elaborate in basis B , it is required that each $longtycon_i$ denotes a type name which is flexible in B . In particular, no $longtycon_i$ may denote a type function which is not also a type name (e.g., a $longtycon$ must not denote $\Lambda().s * s$).

For example, the two signature expressions

```

sig                                sig
  type s                            type s
  type t                            datatype t = C
  sharing type s = t                sharing type s = t
end                                  end

```

are both legal. By contrast, the signature expressions

```

sig                                sig
  type s                            type s = int
  type t = s*s                       datatype t = C
  sharing type s = t                sharing type s = t
end                                  end

```

are both illegal.

G.3.2 The equality attribute of specified types

If *spec* `sharing type longtycon1 = ... = longtyconn` elaborates successfully, then all *n* type constructors will thereafter denote the same type name. This type name will admit equality, if *spec* associates an equality type name with one of the type constructors. Thus

```

eqtype t
type u
sharing type t = u

```

is legal and both `t` and `u` are equality types after the sharing qualification. The mechanism for inferring equality attributes for datatype specifications is the same as for inferring equality attributes for datatype declarations. Thus the specification

```

datatype answer = YES | NO
datatype 'a option = Some of 'a | None

```

specifies two equality types. Every specification of the form `datatype datdesc` introduces one type name for each type constructor described by *datdesc*. The equality attribute of such a type name is determined at the point where the specification occurs. Thus, in

```

type s
datatype t = C of s

```

the type name associated with `t` will not admit equality, even if `s` later is instantiated to an equality type. Type names associated with datatype specifications can be instantiated to other type names by subsequent `type sharing` or `where type` qualifications. In this case, no effort is made to ban type environments that do not respect equality. For example,

```

sig
  eqtype s
  datatype t = C of int -> int
  sharing type s = t
end

```

is legal in SML '97, even though it cannot be matched by any real structure.

G.3.3 Structure Sharing

For convenience, structure sharing constraints are provided, but only as a shorthand for type sharing constraints. There is a derived form of specification

$$spec \text{ sharing } longstrid_1 = \dots = longstrid_k \quad (k \geq 2)$$

Here *spec* must specify $longstrid_1, \dots, longstrid_k$. The equivalent form consists of *spec* qualified by all the type sharing constraints

$$sharing \text{ type } longstrid_i.longtycon = longstrid_j.longtycon$$

($1 \leq i < j \leq k$) such that both $longstrid_i.longtycon$ and $longstrid_j.longtycon$ are specified by *spec*.

In SML '90, structure sharing constraints are transitive, but in SML '97 they are not. For example,

```

structure A: sig type t end
structure B: sig end
structure C: sig type t end
sharing A=B=C

```

induces type sharing on `t`, whereas

```

structure A: sig type t end
structure B: sig end
structure C: sig type t end
sharing A=B sharing B=C

```

induces no type sharing. Thus a structure sharing constraint in some cases induces less sharing in SML '97 than in SML '90.

Next, SML '97 does not allow structure sharing equations which refer to “external” structures. For example, the program

```

structure A= struct end;
signature SIG = sig structure B : sig end
                 sharing A = B
end;

```

is not legal in SML '97, because the sharing constraint now only qualifies the specification `structure B: sig end`, which does not specify `A`. Thus not all legal SML '90 signatures are legal in SML '97.

The removal of structure sharing has a dramatic simplifying effect on the semantics. Most importantly, the elaboration rules can be made monogenic (i.e., “deterministic”), up to renaming of new type names. The need for the notion of principal signature (and even equality-principal signature) disappears. The notions of structure name, structure consistency and well-formed signature are no longer required. The notion of cover can be deleted. Only one kind of realisation, namely type realisation, remains. The notion of type-explication has been removed, since it can be proved that signatures automatically are type-explicit in the revised language.

G.4 Value Polymorphism

The imperative types of SML '90 were somewhat subtle, and they propagated into signatures in an unpleasant way. Experiments on existing code suggest that the power of imperative types is rarely used fully and that *value polymorphism*, which can in fact be seen as a restriction of the imperative type discipline, usually suffices[57]. With value polymorphism, there is only one kind of type variable. The definition of non-expansive expressions (see G.13 below) is relaxed to admit more expressions. In a declaration

```
val x = exp
```

the variable x will only be given a non-trivial polymorphic type scheme (i.e., a type scheme which is not also a type) if *exp* is non-expansive. This applies even if there is no application of `ref` in the entire program.

Example: in the declaration `val x = [] @ []`, x can be assigned type `'a list`, but not the type scheme $\forall 'a. 'a \text{ list}$ (since `[] @ []` is an expansive expression). Consequently, `(1::x, true::x)` will not elaborate in the scope of the declaration. Also, if the declaration appears at top level, the compiler may refuse elaboration due to a top-level free type variable (see G.8). Thus the top-level phrase `[] @ []` may fail, since it abbreviates `val it = [] @ []`. But of course it will not fail if a monotype is explicitly ascribed, e.g. `[] @ []:int list`.

On the other hand, in `fun f() = [] @ []` (or `val f = fn () => [] @ []`), f can be assigned type scheme $\forall 'a. \text{unit} \rightarrow 'a \text{ list}$ so that, for example, `(1::f(), true::f())` elaborates. This transformation (η -conversion) often gives the desired polymorphism. But beware that η -conversion can change the meaning of the program, if *exp* does not terminate or has side-effects.

G.5 Identifier Status

The 1990 Definition treated identifier status informally (in Section 2.4); a fuller definition was given in the Commentary[41, Appendix B]. However, some problems with the handling of exception constructors remained[27, Sect. 10.3].

In the present document, we have collapsed the three identifier classes `Var`, `ExCon` and `Con` into a single class, `VID`, of *value identifiers*. The semantic objects *VE* previously called variable environments are replaced by *value environments*. A value environment maps value identifiers to pairs of the form (o, is) , where o is some semantic object and is is an *identifier status* ($is \in \{v, c, e\}$) indicating whether the identifier should be regarded as a value variable (**v**), a value constructor (**c**) or an exception constructor (**e**). These changes have been carried out both in the static and in the dynamic semantics, for both Core and Modules. Thus the assignment of identifier status is incorporated formally in the present Definition.

The definition of enrichment has been modified to allow an identifier that has been specified as a value to be matched by a value constructor or an exception constructor. However, a specification of a value or exception constructor must be matched by a value or exception constructor, respectively.

Thus, the status descriptor says more than just what the lexical status

of the identifier is — it is a statement about the value in the corresponding dynamic environment: if the status of *id* in the static environment is *c*, then the value in a matching dynamic environment must be a value constructor. Similarly, if the status of *id* in the static environment is *e*, then the value in a matching dynamic environment must be an exception name. If the status of *id* is just *v*, however, the corresponding value in the dynamic environment can be any kind of value (of the appropriate type), including a value constructor and an exception name.

The exception environment (*EE*) has been deleted from the semantics, since it is no longer required for the definition of enrichment. Also, the constructor environment *CE* in the static semantics has been replaced by a value environment in which every identifier has status *c*.

The new handling of identifier status admits some `val rec` declarations that were illegal in SML '90 (see the comment to Rule 26).

G.6 Replication of Datatypes

SML '97 allows *datatype replication*, i.e. declarations and specifications of the form

```
datatype tycon ==- datatype longtycon
```

When elaborated, this binds type constructor *tycon* to the entire type structure (value constructors included) to which *longtycon* is bound in the context. Datatype replication does not generate a new datatype: the original and the replicated datatype share.

Here is an example of a use of the new construct:

```
signature MYBOOL =
sig
  type bool
  val xor: bool * bool -> bool
end;
structure MyBool: MYBOOL =
struct
  datatype bool = datatype bool (* from the initial basis *)
  fun xor(true, false) = true
    | xor(false, true) = true
    | xor _ = false
end;
```

```
val x = MyBool.xor(true, false);
```

Here `MyBool.xor(true, false)` evaluates to `true`. Note the use of transparent signature matching; had opaque matching been used instead, the declaration of `x` would not have elaborated.

A datatype replication implicitly introduces the value constructors of *longtycon* into the current scope. This is significant for signature matching. For example, the following program is legal:

```
datatype t0 = C;
structure A : sig type t val C: t end =
  struct
    datatype t = datatype t0
  end;
```

Note that `C` is specified as a value in the signature; the datatype replication copies the value environment of `t0` into the structure and that is why the structure contains the required `C` value.

To make it possible for datatype replication to copy value environments associated with type constructors, the dynamic semantics has been modified so that environments now contain a *TE* component (see Figure 13, page 42). Further, in the dynamic semantics of modules, the \downarrow operation, which is used for cutting down structures when they are matched against signatures, has been extended to cover the *TE* component (see page 54). In the above example, the value environment assigned to `A.t` will be empty, signifying that the type has no value constructors. Had the signature instead been

```
sig datatype t val C: t end
```

then the signature matching would have assigned `A.t` a value environment with domain $\{C\}$, indicating that `A.t` has value constructor `C`.

When the datatype replication is used as a specification, *longtycon* can refer to a datatype which has been introduced either by declaration or by specification. Here is an example of the former:

```
datatype t = C | D;
signature SIG =
sig
  datatype t = datatype t (* replication is not recursive! *)
  val f: t -> t
end
```

G.7 Local Datatypes

This change is concerned with expressions of the form `let dec in exp end` in which *dec* contains a `datatype` declaration. Let us refer to such a datatype declaration as a *local* datatype declaration. There are two reasons why changes to the handling of local datatype declarations are necessary.

The first is that the rule given for elaboration of `let`-expressions in the 1990 Definition is unsound^[27]; the problem has to do with the ability to export type names of locally declared datatypes out of scope.

The second is that the static semantics relies on the following invariant about all contexts, *C*, which arise in elaboration from the initial basis:

$$\text{tynames } C \subseteq T \text{ of } C$$

This invariant is used, for example, in the rule for elaborating datatype declarations, where type names are picked “fresh” with respect to *T* of *C*. As pointed out by Kahrs, the second premise of rule 16 in the 1990 Definition violates the above invariant.

To solve the first problem, the rule for elaborating `let`-expressions (rule 4 in the present document) has been provided with a side-condition which prevents the type of *exp* from containing type names generated by *dec*. For example,

```
let datatype t = C in C end
```

was legal SML '90 but is not legal SML '97.

To solve the second problem, a side-condition has been added in the rule for matches and the rule for `val rec` (rules 14 and 26 of the present document). As a consequence, again fewer programs elaborate. For example, the expression

```
fn x => let datatype t = C
        val _ = if true then x else C
      in 5
      end
```

is not legal SML '97, although it was legal SML '90.

G.8 Principal Environments

In SML '90, the elaboration rule which allows any *dec* to appear as a *strdec* is

$$\frac{C \text{ of } B \vdash dec \Rightarrow E \quad E \text{ principal for } dec \text{ in } (C \text{ of } B)}{B \vdash dec \Rightarrow E}$$

The side-condition forces the type scheme in *E* to be as general as possible. However, this side-condition would be undesirably restrictive in SML '97, since the new definition of the `Clos` operation admits less polymorphism than the one used in SML '90. For example, neither

```
val f = (fn x => x)(fn x => x)
structure A = struct end
val y = f 7
```

(where the presence of the `structure` declaration forces each `val` declaration to be parsed as a *strdec*), nor

```
structure A: sig val f: int -> int end =
  struct
    val f = (fn x => x)(fn x => x)
  end
```

would be legal in SML '97, if the side-condition were enforced. (A type-checker may at first infer the type `'a → 'a` from the declaration of `f`, but since `(fn x => x)(fn x => x)` is expansive, the generalisation to $\forall 'a. 'a \rightarrow 'a$ is not allowed.) By dropping the side-condition, it becomes possible to have the textual context of a structure-level declaration constrain free type variables to monotypes. Thus both the above examples can be elaborated.

Rather than lifting the notion of principal environments to the modules level, we have chosen to drop the requirement of principality. Since the notion of principal environments is no longer used in the rules, even the definition of principal environments has been removed. In practice, however, type checkers still have to infer types that are as general as possible, since implementations should not reject programs for which successful elaboration is possible.

In order to avoid reporting free type variables to users, rules 87 and 89 require that the environment to which a *topdec* elaborates must not contain free type variables. It is possible to satisfy this side-condition by replacing such type variables by arbitrary monotypes; however, implementers may instead choose to refuse elaboration in such situations.

G.9 Consistency and Admissibility

The primary purpose of consistency in SML '90 was to allow a very simple elaboration rule for structure sharing. A secondary purpose was to ban any signature which, because it specifies a datatype in inconsistent ways (e.g. with different constructors), can never be matched. With the removal of structure sharing, the primary purpose of consistency has gone away. In our experience, the secondary purpose has turned out not to be very significant in practice. Textual copying of datatype specifications in different signatures is best avoided, since changes in the datatype will have to be done several places. In practice, it is better to specify a datatype in one signature and then access it elsewhere using structure specifications or `include`. In SML '90 one could specify sharing between a datatype specification and an external (i.e., declared) datatype, and a consistency check was useful in this case. But in SML '97 this form of sharing is not allowed, so there remains no strong reason for preserving consistency; therefore it has been dropped.

In SML '90, admissibility was imposed partly to ensure the existence of principal signatures (which are no longer needed) and partly to ban certain unmatchable signatures. In SML '90, admissibility was the conjunction of well-formedness, cycle-freedom and consistency. Cycle-freedom is no longer relevant, since there is no structure sharing. We have already discussed consistency. Well-formedness of signatures is no longer relevant, but the notion of well-formed type structures is still relevant. It turns out that well-formedness only needs to be checked in one place (in rule 64). Otherwise, well-formedness is preserved by the rules (in a sense which can be made precise). Thus one can avoid a global well-formedness requirement and dispense with admissibility. This we have done.

G.10 Special Constants

The class of special constants has been extended with `word` and `char` constants and with hexadecimal notation. Also, there are additional escape sequences in strings and support for UNICODE characters. See Section 2.2.

G.11 Comments

A clarification concerning unmatched comment brackets was presented in the Commentary; subsequently, Stefan Kahrs discovered a problem with de-

manding that an unmatched `*`) be reported by the compiler. In SML '97, we therefore simply demand that an unmatched `*` must be reported by the compiler.

G.12 Infix Operators

The rules for associativity of infix operators at the same level of precedence have been modified, to avoid confusion between right- and left-associative operators with the same binding precedence (see Section 2.6).

G.13 Non-expansive Expressions

The class of non-expansive expressions (Section 4.7) has been extended, to compensate for the loss of polymorphism which value polymorphism entails.

G.14 Rebinding of built-in identifiers

In SML '97, no *datbind*, *valbind* or *exbind* may bind `true`, `false`, `nil`, `::` or `ref` and no *datbind* or *exbind* may bind `it` (Section 2.9). Similarly, no *datdesc*, *valdesc* or *exdesc* may describe `true`, `false`, `nil`, `::` or `ref` and no *datdesc* or *exdesc* may describe `it` (Section 3.5). These changes are made in order to fix the meaning of derived forms and to avoid ambiguity in the handling of `ref` in the dynamic semantics of the Core.

G.15 Grammar for Modules

There are several new derived forms for modules, see Appendix A (Figures 18 and 19). The grammar for *topdec* has been modified, so that there is no longer any need to put semicolons at the end of signature and functor declarations. Empty and sequential signature and functor declarations have been removed, as they no longer serve any purpose. SML '97 has neither functor signature expressions nor functor specifications, since they could not occur in programs and did not gain wide acceptance.

G.16 Closure Restrictions

Section 3.6 of the 1990 Definition has been deleted.

G.17 Specifications

`open` and `local` specifications have been criticised on the grounds of programming methodology[4]. Also, they are no longer needed for defining the derived forms for functors and they conflict with a desire to have all signatures be type-explicit.

SML '97 therefore admits neither `open` nor `local` in specifications. Moreover, sequential specifications must not specify the same identifier twice. As a consequence, the definition of type-explicitness has been removed: type-explicitness is automatically preserved by elaboration (if one starts in the initial basis) so there is no need to impose type-explicitness explicitly.

G.18 Scope of Explicit Type Variables

A binding construct for explicit type variables has been introduced at `val` and `fun` (see Figure 21). For example, one can declare the polymorphic identity function by

```
fun 'a id(x:'a) = x
```

There is no requirement that all explicit type variables be bound by this binding construct. For those that are not, the scope rules of the 1990 Definition apply. The explicit binding construct has no impact on the dynamic semantics. In particular, there are no explicit type abstractions or applications in the dynamic semantics.

G.19 The Initial Basis

To achieve a clean interface to the new Standard ML Basis Library[18], the initial basis (Appendices C and D) has been cut down to a bare minimum. The present Definition only provides what is necessary in order to define the derived forms and special constants of type `int`, `real`, `word`, `char` and `string`. The following identifiers are no longer defined in the initial basis: `<>`, `^`, `!`, `@`, `Abs`, `arctan`, `chr`, `Chr`, `close_in`, `close_out`, `cos`, `Diff`, `Div`, `end_of_stream`, `exp`, `Exp`, `explode`, `floor`, `Floor`, `implode`, `input`, `instream`, `Interrupt`, `Io`, `ln`, `Ln`, `lookahead`, `map`, `Mod`, `Neg`, `not`, `real` (the coercion function), `rev`, `sin`, `size`, `sqrt`, `Sqrt`, `std_in`, `std_out`, `Sum`, `output`, `outstream`, `Prod`, `Quot`. The corresponding basic values have also been deleted.

G.20 Overloading

The Standard ML Basis Library[18] rests on an overloading scheme for special constants and pre-defined identifiers. We have adopted this scheme (see Appendix E).

G.21 Reals

`real` is no longer an equality type and real constants are no longer allowed in patterns. The Basis Library provides IEEE equality operations on reals.

References

- [1] Maria Virginia Aponte. Extending record typing to type parametric modules with sharing. In *Proc. of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 465–478. ACM Press, January 1993.
- [2] A. W. Appel and D. B. MacQueen. Standard ML of New Jersey. In *Programming Language Implementation and Logic Programming*, volume 528 of *Lecture Notes in Computer Science*, pages 1–26, New York, N.Y., August 1991. Springer-Verlag.
- [3] Andrew Appel, David MacQueen, Robin Milner, and Mads Tofte. Unifying exceptions with constructors in Standard ML. LFCs Report Series ECS-LFCS-88-55, Laboratory for Foundations of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, U.K., June 1988.
- [4] Andrew W. Appel. A critique of Standard ML. *Journal of Functional Programming*, 3(4):391–429, October 1993.
- [5] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*. ACM, Springer-Verlag, Sept 1987.
- [6] Lennart Augustsson and Thomas Johnsson. Lazy ML user’s manual. Technical report, Department of Computer Science, Chalmers University of Technology, 1987.
- [7] Dave Berry. The Edinburgh SML Library. Technical Report ECS-LFCS-91-148, Laboratory for Foundations of Computer Science, Department of Computer Science, Edinburgh University, April 1991.
- [8] Dave Berry. Lessons from the design of a Standard ML library. *Journal of Functional Programming*, 3(4):527–552, October 1993.
- [9] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit (Version 1). Technical Report DIKU-report 93/14, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, 1993.

- [10] R. M. Burstall and R. Popplestone. POP-2 reference manual. In Dale and Michie, editors, *Machine Intelligence 2*. Oliver and Boyd, 1968.
- [11] Rod Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *Proc. Fifth Int'l Joint Conf. on Artificial Intelligence*, pages 1045–1058, 1977.
- [12] Rod Burstall, David MacQueen, and Donald Sannella. HOPE: An experimental applicative language. In *Proc. 1980 LISP Conference*, pages 136–143, Stanford, California, 1980. Stanford University.
- [13] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. *Science of Computer Programming*, 8, May 1987.
- [14] H. B. Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23:83–92, 1969.
- [15] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Proc. Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
- [16] Luis Manuel Martins Damas. *Type Assignment in Programming Languages*. PhD thesis, Edinburgh University, 1985.
- [17] Thierry Despeyroux. Executable specifications of static semantics. In Gilles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer Verlag, June 1984.
- [18] E.R. Gansner and J.H. Reppy (eds.). The Standard ML Basis Library reference manual. (In preparation).
- [19] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.
- [20] M.J.C. Gordon, R. Milner, L. Morris, M.C. Newey, and C.P. Wadsworth. A metalanguage for interactive proof in LCF. In *Proc. Fifth ACM Symposium on Principles of Programming Languages*, Tucson, AZ, 1978.

- [21] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. Twenty-First ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.
- [22] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137. ACM Press, January 1994.
- [23] Robert Harper, David MacQueen, and Robin Milner. Standard ML. Technical Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Edinburgh University, March 1986.
- [24] Robert Harper, Robin Milner, and Mads Tofte. A type discipline for program modules. In *Proc. Int'l Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, pages 308–319. Springer-Verlag, Mar. 1987. Lecture Notes in Computer Science, Vol. 250.
- [25] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Trans. on Prog. Lang. and Sys.*, 15(2):211–252, April 1993.
- [26] J. Roger Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–40, 1969.
- [27] Stefan Kahrs. Mistakes and ambiguities in the Definition of Standard ML. Technical Report ECS-LFCS-93-257, Dept. of Computer Science, University of Edinburgh, 1993.
- [28] Peter J. Landin. The next 700 programming languages. *Comm. ACM*, 9(3):57–164, 1966.
- [29] Xavier Leroy. Polymorphism by name. In *Proc. Twentieth ACM Symposium on Principles of Programming Languages*, January 1993.
- [30] Xavier Leroy. Manifest types, modules and separate compilation. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 109–122. ACM Press, January 1994.

- [31] Xavier Leroy. The Caml Special Light system. Software and documentation available on the Web, <http://pauillac.inria.fr/csl/>, 1995.
- [32] D. MacQueen. Modules for Standard ML. In *Conf. Rec. of the 1984 ACM Symp. on LISP and Functional Programming*, pages 198–207, Aug. 1984.
- [33] David MacQueen. Using dependent types to express modular structure. In *Proc. Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.
- [34] David. B. MacQueen. Structures and parameterisation in a typed functional language. In *Proc. Symposium on Functional Programming and Computer Architecture*, Aspinas, Sweden, 1981.
- [35] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sannella, editor, *Proceedings of the 5th European Symposium on Programming (ESOP)*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423. Springer-Verlag, 1994.
- [36] John McCarthy. *LISP 1.5 Programmer's Manual*. MIT Press, 1956.
- [37] D. Meredith. In memoriam Carew Arthur Meredith. *Notre Dame Journal of Formal Logic*, 18:513–516, 1977.
- [38] Robin Milner. A theory of type polymorphism in programming languages. *J. Computer and Systems Sciences*, 17:348–375, 1978.
- [39] Robin Milner. How ML evolved. *Polymorphism: The ML/LCF/Hope Newsletter*, 1(1), 1983.
- [40] Robin Milner. Changes to the Standard ML core language. Technical Report ECS-LFCS-87-33, Laboratory for Foundations of Computer Science, Edinburgh University, 1987.
- [41] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.
- [42] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

- [43] James H. Morris. *Lambda Calculus Models of Programming Languages*. PhD thesis, MIT, 1968.
- [44] Colin Myers, Chris Clack, and Ellen Poon. *Programming with Standard ML*. Prentice Hall, 1993.
- [45] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with LCF*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.
- [46] Lawrence C. Paulson. *ML for the Working Programmer (2nd edition)*. Cambridge University Press, 1996.
- [47] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Computer Science Department, Aarhus University, 1981.
- [48] John A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [49] Donald Sannella and Andrzej Tarlecki. Program specification and development in Standard ML. In *Proc. Twelfth ACM Symposium on Principles of Programming Languages*, New Orleans, 1985.
- [50] Ryan Stansifer. *ML Primer*. Prentice Hall, 1992.
- [51] Chris Stone and Robert Harper. A type-theoretic account of Standard ML 1996. Technical Report CMU-CS-96-136, School of Computer Science, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, May 1996.
- [52] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN Symposium on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.
- [53] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, Edinburgh University, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, May 1988. Available as Technical Report CST-52-88.

- [54] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1), November 1990.
- [55] Mads Tofte. Principal signatures for higher-order program modules. *Journal of Functional Programming*, 4(3):285–335, July 1994.
- [56] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, 1994.
- [57] Andrew Wright. Simple imperative polymorphism. *Journal of Lisp and Symbolic Computation*, 8(4):343–355, December 1995.

