The Definition of Standard ML

Robin Milner Mads Tofte
Robert Harper
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

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, particular 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.

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 fourteen years as a fusion of many ideas from many people. This evolution is described in some detail in Appendix E 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 strong research interest at Edinburgh and elsewhere, throughout these fourteen 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 *signatures*. 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 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 $P_1, \ldots, P_n$.

The form of the Definition[1]

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 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. Some expository books have already been written, and this Definition will be useful as a background reference for their readers. We have also become convinced, while writing the 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 will not be answered

[1]The Definition has evolved through a sequence of three previous versions, circulated as Technical Reports. For those who have followed the sequence, we should point out that the treatment of *equality types* and of *admissibility* has been slightly modified in this publication to meet the claim for principal signatures. The changes are mainly in Sections 4.9, 5.5 and 5.13 and in the inference rules 19, 20, 29 and 65.

by pedagogic texts either. So we are writing a Commentary on the Definition which will assist people in reading it, and which will serve as a bridge between the Definition and other texts.

Edinburgh
August 1989

Contents

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 just 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:

	Core	*Modules*	*Programs*
Syntax	Section 2	Section 3	Section 8
Static Semantics	Section 4	Section 5	
Dynamic Semantics	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 all predefined identifiers.

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

$$A \vdash phrase \Rightarrow A'$$

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. This theory, particularly the theory of types and signatures, will benefit from a more pedagogic treatment in other publications; the treatment here is probably the minimum required to understand the meaning of the rules.

The robustness of the semantics depends upon theorems. Usually these have been proven, but the proof is not included. In two cases, however, they are presented as "claims" rather than theorems; these are the claim of principal environments in Section 4.12, and the claim of principal signatures in Section 5.13. We need further confirmation of our detailed proofs of these claims, before asserting them as theorems.

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   do   datatype   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* is any non-empty sequence of digits, possibly preceded by a negation symbol (`~`). A *real constant* is an integer constant, possibly followed by a point (`.`) and one or more digits, possibly followed by an exponent symbol `E` and an integer constant; 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 256 characters (numbered 0 to 255) such that the characters with numbers 0 to 127 coincide with the ASCII character set. 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:

`\n`	A single character interpreted by the system as end-of-line.
`\t`	Tab.
`\^`c	The control character c, where c may be any character with number 64–95. The number of `\^`c is 64 less than the number of c.
`\`*ddd*	The single character with number *ddd* (3 decimal digits denoting an integer in the interval $[0, 255]$).
`\"`	`"`
`\\`	`\`
`\`$f \cdot\cdot f$`\`	This sequence is ignored, where $f \cdot\cdot f$ stands for a sequence of one or more formatting characters.

Var	(value variables)	long
Con	(value constructors)	long
ExCon	(exception constructors)	long
TyVar	(type variables)	
TyCon	(type constructors)	long
Lab	(record labels)	
StrId	(structure identifiers)	long

Figure 1: Identifiers

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.

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

2.3 Comments

A *comment* is any character sequence within comment brackets `(* *)` in which comment brackets are properly nested. An unmatched comment bracket should be detected by the compiler.

2.4 Identifiers

The classes of *identifiers* for the Core are shown in Figure 1. We use *var*, *tyvar* to range over Var, 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:

longx	::=	x	identifier
		$strid_1.\cdots.strid_n.x$	qualified identifier ($n \geq 1$)

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 subclass ImpTyVar of TyVar, the *imperative* type variables, consists of those which start with one or two primes followed by an underbar. The complement AppTyVar = TyVar \ ImpTyVar consists of the *applicative* type variables. The other six classes (Var, Con, ExCon, TyCon, Lab and StrId) 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 22). The class Lab is extended to include the *numeric* labels `1` `2` `3` $\cdots$, i.e. any numeral not starting with `0`.

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

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, and must be within the scope of the type binding or datatype binding which introduced it.

4. Elsewhere, *id* is an exception constructor if it occurs in the scope of an exception binding which introduces it as such, or a value constructor if it occurs in the scope of a datatype binding which introduced it as such; otherwise it is a value variable.

It follows from the last rule that no value declaration can make a "hole" in the scope of a value or exception constructor by introducing the same identifier as a variable; this is because, in the scope of the declaration which introduces *id* as a value or exception constructor, any occurrence of *id* in a

pattern is interpreted as the constructor and not as the binding occurrence of a new variable.

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 Infixed 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 *var*, a *con* or an *excon* within the scope (see below) of the directive. (Note that qualified identifiers never have infix status.) If *id* has infix status, then "exp_1 *id* exp_2" (resp. "pat_1 *id* pat_2") may occur – in parentheses if necessary – wherever the application "*id*`{1=`exp_1`,2=`exp_2`}`" or its derived form "*id*`(`exp_1`,`exp_2`)`" (resp "*id*`(`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-infixed. The only required use of `op` is in prefixing a non-infixed occurrence of an identifier *id* 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 ($n \geq 1$):

$$\texttt{infix}\ \langle d \rangle\ id_1 \cdots id_n$$

$$\texttt{infixr}\ \langle d \rangle\ id_1 \cdots id_n$$

$$\texttt{nonfix}\ id_1 \cdots id_n$$

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; association is always to the left for different operators of the same precedence. 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

`let` *dec* `in` $\cdots$ `end`

`local` *dec* `in` $\cdots$ `end`

then the scope of *dir* does not extend beyond the phrase. Further scope limitations are imposed for Modules.

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:

xseq	::=	x	(singleton sequence)
			(empty sequence)
		$(x_1, \cdots, x_n)$	(sequence, $n \geq 1$)

  (Note that the "$\cdots$" used here, meaning syntactic iteration, 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 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.

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvar*	value variable
		⟨`op`⟩*longcon*	value constructor
		⟨`op`⟩*longexcon*	exception constructor
		`{` ⟨*exprow*⟩ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		exp_1 *id* exp_2	infixed application
		exp `:` *ty*	typed (L)
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	
dec	::=	`val` *valbind*	value declaration
		`type` *typbind*	type declaration
		`datatype` *datbind*	datatype declaration
		`abstype` *datbind* `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \cdots longstrid_n$	open declaration ($n \geq 1$)
			empty declaration
		dec_1 ⟨;⟩ dec_2	sequential declaration
		`infix` ⟨*d*⟩ $id_1 \cdots id_n$	infix (L) directive
		`infixr` ⟨*d*⟩ $id_1 \cdots id_n$	infix (R) directive
		`nonfix` $id_1 \cdots id_n$	nonfix directive
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
typbind	::=	*tyvarseq tycon* `=` *ty* ⟨`and` *typbind*⟩	
datbind	::=	*tyvarseq tycon* `=` *conbind* ⟨`and` *datbind*⟩	
conbind	::=	⟨`op`⟩*con* ⟨`of` *ty*⟩ ⟨ `|` *conbind*⟩	
exbind	::=	⟨`op`⟩*excon* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		⟨`op`⟩*excon* `=` ⟨`op`⟩*longexcon* ⟨`and` *exbind*⟩	

Figure 3: Grammar: Expressions, Matches, Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		⟨`op`⟩*var*	variable
		⟨`op`⟩*longcon*	constant
		⟨`op`⟩*longexcon*	exception constant
		`{` ⟨*patrow*⟩ `}`	record
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow*⟩	pattern row
pat		*atpat*	atomic
		⟨`op`⟩*longcon atpat*	value construction
		⟨`op`⟩*longexcon atpat*	exception construction
		pat_1 *con* pat_2	infixed value construction
		pat_1 *excon* pat_2	infixed exception construction
		pat `:` *ty*	typed
		⟨`op`⟩*var*⟨`:` *ty*⟩ `as` *pat*	layered
ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record type expression
		tyseq longtycon	type construction
		ty `->` *ty′*	function type expression (R)
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow*⟩	type-expression row

Figure 4: Grammar: Patterns and Type expressions

2.9 Syntactic Restrictions

- No pattern may contain the same *var* twice. No expression row, pattern row or type row may bind the same *lab* twice.
- No binding *valbind*, *typbind*, *datbind* or *exbind* may bind the same identifier twice; this applies also to value constructors within a *datbind*.
- In the left side *tyvarseq tycon* of any *typbind* or *datbind*, *tyvarseq* must not contain the same *tyvar* twice. Any *tyvar* occurring within the right side must occur in *tyvarseq*.

- For each value binding *pat* `=` *exp* within `rec`, *exp* must be of the form `fn` *match*, possibly constrained by one or more type expressions. The derived form of function-value binding given in Appendix A, page 80, necessarily obeys this restriction.

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 concern functors and 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
```

3.2 Identifiers

The additional syntax classes for Modules are SigId (signature identifiers) and FunId (functor identifiers); they may be either alphanumeric – not starting with a prime – or symbolic. 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 Infixed 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` $\cdots$ `end`

`local` *strdec* `in` $\cdots$ `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 generative structure expression – in particular, to such an expression occurring as a functor body. A more liberal scheme (which is under consideration) would allow fixity directives to appear also as specifications, so that fixity may be dictated by a signature expression; furthermore, it would allow an `open` or `include` construction to restore the fixity which prevailed in the structures being opened, or in the signatures being included. This scheme is not adopted at present.

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 grammat-

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
SharEq	sharing equations
FunDec	functor declarations
FunBind	functor bindings
FunSigExp	functor signature expressions
FunSpec	functor specifications
FunDesc	functor descriptions
TopDec	top-level declarations

Figure 5: Modules Phrase Classes

ical rules are shown in Figures 6, 7 and 8.

It should be noted that functor specifications (FunSpec) cannot occur in programs; neither can the associated functor descriptions (FunDesc) and functor signature expressions (FunSigExp). The purpose of a *funspec* is to specify the static attributes (i.e. functor signature) of one or more functors. This will be useful, in fact essential, for separate compilation of functors. If, for example, a functor g refers to another functor f then — in order to compile g in the absence of the declaration of f — at least the specification of f (i.e. its functor signature) must be available. At present there is no special grammatical form for a separately compilable "chunk" of text – which we may like to call call a *module* – containing a *fundec* together with a *funspec*

specifying its global references. However, below in the semantics for Modules it is defined when a declared functor matches a functor signature specified for it. This determines exactly those functor environments (containing declared functors such as f) into which the separately compiled "chunk" containing the declaration of g may be loaded.

strexp	::=	`struct` *strdec* `end`	generative
		longstrid	structure identifier
		funid `(` *strexp* `)`	functor application
		`let` *strdec* `in` *strexp* `end`	local declaration
strdec	::=	*dec*	declaration
		`structure` *strbind*	structure
		`local` $strdec_1$ `in` $strdec_2$ `end`	local
			empty
		$strdec_1$ ⟨`;`⟩ $strdec_2$	sequential
strbind	::=	*strid* ⟨`:` *sigexp*⟩ `=` *strexp* ⟨`and` *strbind*⟩	
sigexp	::=	`sig` *spec* `end`	generative
		sigid	signature identifier
sigdec	::=	`signature` *sigbind*	single
			empty
		$sigdec_1$ ⟨`;`⟩ $sigdec_2$	sequential
sigbind	::=	*sigid* `=` *sigexp* ⟨`and` *sigbind*⟩	

Figure 6: Grammar: Structure and Signature Expressions

3.5 Syntactic Restrictions

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

spec	::=	`val` *valdesc*	value
		`type` *typdesc*	type
		`eqtype` *typdesc*	eqtype
		`datatype` *datdesc*	datatype
		`exception` *exdesc*	exception
		`structure` *strdesc*	structure
		`sharing` *shareq*	sharing
		`local` $spec_1$ `in` $spec_2$ `end`	local
		`open` $longstrid_1 \cdots longstrid_n$	open ($n \geq 1$)
		`include` $sigid_1 \cdots sigid_n$	include ($n \geq 1$)
			empty
		$spec_1$ $\langle$`;`$\rangle$ $spec_2$	sequential
valdesc	::=	*var* `:` *ty* $\langle$`and` *valdesc*$\rangle$	
typdesc	::=	*tyvarseq tycon* $\langle$`and` *typdesc*$\rangle$	
datdesc	::=	*tyvarseq tycon* `=` *condesc* $\langle$`and` *datdesc*$\rangle$	
condesc	::=	*con* $\langle$`of` *ty*$\rangle$ $\langle$ `|` *condesc*$\rangle$	
exdesc	::=	*excon* $\langle$`of` *ty*$\rangle$ $\langle$`and` *exdesc*$\rangle$	
strdesc	::=	*strid* `:` *sigexp* $\langle$`and` *strdesc*$\rangle$	
shareq	::=	$longstrid_1$ `=` $\cdots$ `=` $longstrid_n$	structure sharing ($n \geq 2$)
		`type` $longtycon_1$ `=` $\cdots$ `=` $longtycon_n$	type sharing ($n \geq 2$)
		$shareq_1$ `and` $shareq_2$	multiple

Figure 7: Grammar: Specifications

fundec	::=	`functor` *funbind*	single
			empty
		fundec$_1$ ⟨`;`⟩ *fundec*$_2$	sequence
funbind	::=	*funid* `(` *strid* `:` *sigexp* `)` ⟨`:` *sigexp*′⟩ `=` *strexp* ⟨`and` *funbind*⟩	functor binding
funsigexp	::=	`(` *strid* `:` *sigexp* `)` `:` *sigexp*′	functor signature expression
funspec	::=	`functor` *fundesc*	functor specification
			empty
		funspec$_1$ ⟨`;`⟩ *funspec*$_2$	sequence
fundesc	::=	*funid funsigexp* ⟨`and` *fundesc*⟩	
topdec	::=	*strdec*	structure-level declaration
		sigdec	signature declaration
		fundec	functor declaration

Note: No *topdec* may contain, as an initial segment, a shorter top-level declaration followed by a semicolon.

Figure 8: Grammar: Functors and Top-level Declarations

3.6 Closure Restrictions

The semantics presented in later sections requires no restriction on reference to non-local identifiers. For example, it allows a signature expression to refer to external signature identifiers and (via `sharing` or `open`) to external structure identifiers; it also allows a functor to refer to external identifiers of any kind.

However, implementers who want to provide a simple facility for separate compilation may want to impose the following restrictions (ignoring references to identifiers bound in the initial basis B_0, which may occur anywhere):

1. In any signature binding *sigid* `=` *sigexp* , the only non-local references in *sigexp* are to signature identifiers.

2. In any functor description *funid* `(` *strid* `:` *sigexp* `)` `:` *sigexp*′ , the only non-local references in *sigexp* and *sigexp*′ are to signature identifiers, except that *sigexp*′ may refer to *strid* and its components.

3. In any functor binding *funid* `(` *strid* `:` *sigexp* `)` $\langle$`:` *sigexp*$'\rangle$ `=` *strexp* , the only non-local references in *sigexp*, *sigexp*$'$ and *strexp* are to functor and signature identifiers, except that both *sigexp*$'$ and *strexp* may refer to *strid* and its components.

In the last two cases the final qualification allows, for example, sharing constraints to be specified between functor argument and result. (For a completely precise definition of these closure restrictions, see the comments to rules 66 (page 48), 91 (page 52) and 96 (page 52) in the static semantics of modules, Section 5.)

The significance of these restrictions is that they may ease separate compilation; this may be seen as follows. If one takes a *module* to be a sequence of signature declarations, functor specifications and functor declarations satisfying the above restrictions then the elaboration of a module can be made to depend on the initial static basis alone (in particular, it will not rely on structures outside the module). Moreover, the elaboration of a module cannot create new free structure or type names, so name consistency (as defined in Section 5.2, page 39) is automatically preserved across separately compiled modules. On the other hand, imposing these restrictions may force the programmer to write many more sharing equations than is needed if functors and signature expressions can refer to free structures.

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 structure names. 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. Structure names play an active role only in the Modules semantics; they enter the Core semantics only because they appear in structure environments, which (in turn) are needed in the Core semantics only to determine the values of long identifiers. 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 *tyvar*	$\in$	TyVar	type variables
t	$\in$	TyName	type names
m	$\in$	StrName	structure names

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). Independently hereof, each α possesses a boolean attribute, the *imperative* attribute, which determines whether it is imperative, i.e. whether it is a member of ImpTyVar (defined on page 5) or not.

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 $\text{TyName}^{(k)}$.

With each special constant *scon* we associate a type name type(*scon*) which is either `int`, `real` or `string` as indicated by Section 2.2.

4.2 Compound Objects

When A and B are sets $\mathrm{Fin}\, A$ 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 $\mathrm{Dom}\, f$ and $\mathrm{Ran}\, f$. A finite map will often be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\},\ k \geq 0$; 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 $\mathrm{Dom}\, f \cup \mathrm{Dom}\, g$ and values

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

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, tynames A and tyvars A denote respectively the set of type names and the set of type variables occurring free in A. Moreover, imptyvars A and apptyvars A denote respectively the set of imperative type variables and the set of applicative type variables occurring free in A.

4.3 Projection, Injection and Modification

Projection: We often need to select components of tuples – for example, the variable-environment component of a context. In such cases we rely on variable names to indicate which component is selected. For instance "*VE* of *E*" means "the variable-environment component of *E*" and "*m* of *S*" means "the structure name of *S*".

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 $C(tycon)$ means $(TE \text{ of } C)tycon$.

A particular case needs mention: $C(con)$ is taken to stand for $(VE \text{ of } C)con$; similarly, $C(excon)$ is taken to stand for $(VE \text{ of } C)excon$. The type

$$
\begin{array}{rcl}
\tau & \in & \mathrm{Type} = \mathrm{TyVar} \cup \mathrm{RecType} \cup \mathrm{FunType} \cup \mathrm{ConsType} \\
(\tau_1, \cdots, \tau_k) \text{ or } \tau^{(k)} & \in & \mathrm{Type}^k \\
(\alpha_1, \cdots, \alpha_k) \text{ or } \alpha^{(k)} & \in & \mathrm{TyVar}^k \\
\varrho & \in & \mathrm{RecType} = \mathrm{Lab} \xrightarrow{\text{fin}} \mathrm{Type} \\
\tau \to \tau' & \in & \mathrm{FunType} = \mathrm{Type} \times \mathrm{Type} \\
 & & \mathrm{ConsType} = \cup_{k \geq 0} \mathrm{ConsType}^{(k)} \\
\tau^{(k)} t & \in & \mathrm{ConsType}^{(k)} = \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
\theta \text{ or } \Lambda\alpha^{(k)}.\tau & \in & \mathrm{TypeFcn} = \cup_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
\sigma \text{ or } \forall\alpha^{(k)}.\tau & \in & \mathrm{TypeScheme} = \cup_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
S \text{ or } (m, E) & \in & \mathrm{Str} = \mathrm{StrName} \times \mathrm{Env} \\
(\theta, CE) & \in & \mathrm{TyStr} = \mathrm{TypeFcn} \times \mathrm{ConEnv} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\text{fin}} \mathrm{Str} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{\text{fin}} \mathrm{TyStr} \\
CE & \in & \mathrm{ConEnv} = \mathrm{Con} \xrightarrow{\text{fin}} \mathrm{TypeScheme} \\
VE & \in & \mathrm{VarEnv} = (\mathrm{Var} \cup \mathrm{Con} \cup \mathrm{ExCon}) \xrightarrow{\text{fin}} \mathrm{TypeScheme} \\
EE & \in & \mathrm{ExConEnv} = \mathrm{ExCon} \xrightarrow{\text{fin}} \mathrm{Type} \\
E \text{ or } (SE, TE, VE, EE) & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{VarEnv} \times \mathrm{ExConEnv} \\
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
U & \in & \mathrm{TyVarSet} = \mathrm{Fin}(\mathrm{TyVar}) \\
C \textit{ or } T, U, E & \in & \mathrm{Context} = \mathrm{TyNameSet} \times \mathrm{TyVarSet} \times \mathrm{Env}
\end{array}
$$

Figure 10: Compound Semantic Objects

scheme of a value constructor is held in *VE* as well as in *TE* (where it will be recorded within a *CE*); similarly, the type of an exception constructor is held in *VE* as well as in *EE*. Thus the re-binding of a constructor of either kind is given proper effect by accessing it in *VE*, rather than in *TE* or in *EE*.

Finally, environments may be applied to long identifiers. For instance if $longcon = strid_1.\cdots.strid_k.con$ then $E(longcon)$ means

$$(VE \text{ of } (SE \text{ of } \cdots (SE \text{ of } (SE \text{ of } E) strid_1) strid_2 \cdots) strid_k) con.$$

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

A type function $\theta = \Lambda\alpha^{(k)}.\tau$ has arity k; it must be *closed* – i.e. $\text{tyvars}(\tau) \subseteq \alpha^{(k)}$ – and 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, $\Lambda\alpha.\alpha \to \alpha$ and $\Lambda\beta.\beta \to \beta$ are equal type functions even if α admits equality but β does not. Similarly, the imperative attribute has no significance in the bound variable of a type function. If t has arity k, then we write t to mean $\Lambda\alpha^{(k)}.\alpha^{(k)} t$ (eta-conversion); thus TyName $\subseteq$ 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

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

A type is *imperative* if all type variables occurring in it are imperative.

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, and τ_i is imperative if α_i is imperative. 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 $\forall\alpha.\alpha \to \alpha$ and $\forall\beta.\beta \to \beta$ are only equal if either both α and β admit equality, or neither does. Similarly, the imperative attribute of a bound type variable of a type scheme *is* significant. 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. In the modules, a description of a value, type, or datatype may contain explicit type variables whose scope is that description. However, 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 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, an occurrence of α in a value declaration `val` *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 *scoped at* a particular occurrence O of `val` *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 occurrence O.

Hence, associated with every occurrence of a value declaration there is a set U of the explicit type variables that are scoped at that occurrence. One

may think of each occurrence of `val` as being implicitly decorated with such a set, for instance:

```
val{} x =   (let val{'a} Id1:'a->'a = fn z=>z in Id1 Id1 end,
             let val{'a} Id2:'a->'a = fn z=>z in Id2 Id2 end)

val{'a} x = (let val{} Id:'a->'a = fn z=>z in Id Id end,
             fn z=> z:'a)
```

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. Any variable, constructor and `fn` expression, possibly constrained by one or more type expressions, is non-expansive; all other expressions are said to be expansive. 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 $\mathrm{Clos}_A(\tau)$, the *closure* of τ with respect to A, is the type scheme $\forall\alpha^{(k)}.\tau$, where $\alpha^{(k)} = \mathrm{tyvars}(\tau) \setminus \mathrm{tyvars}\, A$. Commonly, A will be a context C. We abbreviate the *total* closure $\mathrm{Clos}_{\{\}}(\tau)$ to $\mathrm{Clos}(\tau)$. If the range of a variable environment VE contains only types (rather than arbitrary type schemes) we set

$$\mathrm{Clos}_A VE = \{id \mapsto \mathrm{Clos}_A(\tau)\ ;\ VE(id) = \tau\}$$

with a similar definition for $\mathrm{Clos}_A CE$.

Closing a variable 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 $var \in$

Dom VE there is a unique *pat* `=` *exp* in *valbind* which binds *var*. If $VE(var) = \tau$, let $\mathrm{Clos}_{C,valbind} VE(var) = \forall \alpha^{(k)}.\tau$, where

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

Notice that the form of *valbind* does not affect the binding of applicative type variables, only the binding of imperative type variables.

4.9 Type Structures and Type Environments

A type structure (θ, CE) is *well-formed* if either $CE = \{\}$, or θ is a type name t. (The latter case arises, with $CE \neq \{\}$, in `datatype` declarations.) All type structures occurring in elaborations are assumed to be well-formed.

A type structure (t, CE) is said to *respect equality* if, whenever t admits equality, then either $t =$ `ref` (see Appendix C) or, for each $CE(con)$ of the form $\forall \alpha^{(k)}.(\tau \to \alpha^{(k)}t)$, the type function $\Lambda\alpha^{(k)}.\tau$ also admits equality. (This ensures that the equality predicate `=` will be applicable to a constructed value (con, 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, CE) occurs in TE for some $CE \neq \{\}$. 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

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

where no CE_i is the empty map, and for any E we define $\mathrm{Abs}(TE, E)$ to be the environment obtained from E and TE as follows. First, let $\mathrm{Abs}(TE)$ be the type environment $\{tycon_i \mapsto (t_i, \{\})\ ;\ 1 \leq i \leq k\}$ in which all constructor environments CE_i have been replaced by the empty map. Let $t'_1, \cdots, t'_k$ be new distinct type names none of which admit equality. Then $\mathrm{Abs}(TE, E)$ is the result of simultaneously substituting t'_i for t_i, $1 \leq i \leq k$, throughout $\mathrm{Abs}(TE) + E$. (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

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

where A is usually an environment or 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 or environment) 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 $B = (M, T), F, G, E$ 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 $T, U, E \vdash \mathit{phrase} \Rightarrow A$ such that tynames $E \subseteq T$, and let S′ be a sentence $T', U', E' \vdash \mathit{phrase}' \Rightarrow A'$ occurring in a proof of S; then also tynames $E' \subseteq T'$.

Atomic Expressions $\boxed{C \vdash \mathit{atexp} \Rightarrow \tau}$

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

$$\frac{C(\mathit{longvar}) \succ \tau}{C \vdash \mathit{longvar} \Rightarrow \tau} \quad (2)$$

$$\frac{C(longcon) \succ \tau}{C \vdash longcon \Rightarrow \tau} \tag{3}$$

$$\frac{C(longexcon) = \tau}{C \vdash longexcon \Rightarrow \tau} \tag{4}$$

$$\frac{\langle C \vdash exprow \Rightarrow \varrho \rangle}{C \vdash \texttt{\{}\ \langle exprow \rangle\ \texttt{\}} \Rightarrow \{\}\langle +\ \varrho \rangle \text{ in Type}} \tag{5}$$

$$\frac{C \vdash dec \Rightarrow E \qquad C \oplus E \vdash exp \Rightarrow \tau}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{6}$$

$$\frac{C \vdash exp \Rightarrow \tau}{C \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow \tau} \tag{7}$$

Comments:

(2),(3) The instantiation of type schemes allows different occurrences of a single *longvar* or *longcon* to assume different types.

(6) 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.

Expression Rows

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

$$\frac{C \vdash exp \Rightarrow \tau \qquad \langle C \vdash exprow \Rightarrow \varrho \rangle}{C \vdash lab\ \texttt{=}\ exp\ \langle\ \texttt{,}\ exprow \rangle \Rightarrow \{lab \mapsto \tau\}\langle +\ \varrho \rangle} \tag{8}$$

Expressions

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

$$\frac{C \vdash atexp \Rightarrow \tau}{C \vdash atexp \Rightarrow \tau} \tag{9}$$

$$\frac{C \vdash exp \Rightarrow \tau' \to \tau \qquad C \vdash atexp \Rightarrow \tau'}{C \vdash exp\ atexp \Rightarrow \tau} \tag{10}$$

$$\frac{C \vdash exp \Rightarrow \tau \qquad C \vdash ty \Rightarrow \tau}{C \vdash exp\ \texttt{:}\ ty \Rightarrow \tau} \tag{11}$$

$$\frac{C \vdash exp \Rightarrow \tau \qquad C \vdash match \Rightarrow \texttt{exn} \to \tau}{C \vdash exp\ \texttt{handle}\ match \Rightarrow \tau} \tag{12}$$

$$\frac{C \vdash exp \Rightarrow \texttt{exn}}{C \vdash \texttt{raise}\ exp \Rightarrow \tau} \tag{13}$$

$$\frac{C \vdash match \Rightarrow \tau}{C \vdash \texttt{fn}\ match \Rightarrow \tau} \tag{14}$$

Comments:

(9) The relational symbol $\vdash$ is overloaded for all syntactic classes (here atomic expressions and expressions).

(11) 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 `(fn x=>x):'a->'b`. The effect of type variables in an explicitly typed expression is to indicate exactly the degree of polymorphism present in the expression.

(13) Note that τ does not occur in the premise; thus a `raise` expression has "arbitrary" type.

Matches

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

$$\frac{C \vdash mrule \Rightarrow \tau \qquad \langle C \vdash match \Rightarrow \tau \rangle}{C \vdash mrule\ \langle\ \texttt{|}\ match \rangle \Rightarrow \tau} \tag{15}$$

Match Rules

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

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C + VE \vdash exp \Rightarrow \tau'}{C \vdash pat\ \texttt{=>}\ exp\ \Rightarrow \tau \to \tau'} \tag{16}$$

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 11).

Declarations

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

$$\frac{C + U \vdash valbind \Rightarrow VE \qquad VE' = \mathrm{Clos}_{C,valbind} VE \qquad U \cap \mathrm{tyvars}\, VE' = \emptyset}{C \vdash \mathtt{val}_U\ valbind \Rightarrow VE' \text{ in Env}} \tag{17}$$

$$\frac{C \vdash typbind \Rightarrow TE}{C \vdash \mathtt{type}\ typbind \Rightarrow TE \text{ in Env}} \tag{18}$$

$$\frac{\begin{array}{c} C \oplus TE \vdash datbind \Rightarrow VE, TE \qquad \forall (t, CE) \in \mathrm{Ran}\, TE,\ t \notin (T \text{ of } C) \\ TE \text{ maximises equality} \end{array}}{C \vdash \mathtt{datatype}\ datbind \Rightarrow (VE, TE) \text{ in Env}} \tag{19}$$

$$\frac{\begin{array}{cc} C \oplus TE \vdash datbind \Rightarrow VE, TE & \forall (t, CE) \in \mathrm{Ran}\, TE,\ t \notin (T \text{ of } C) \\ C \oplus (VE, TE) \vdash dec \Rightarrow E & TE \text{ maximises equality} \end{array}}{C \vdash \mathtt{abstype}\ datbind\ \mathtt{with}\ dec\ \mathtt{end} \Rightarrow \mathrm{Abs}(TE, E)} \tag{20}$$

$$\frac{C \vdash exbind \Rightarrow EE \quad VE = EE}{C \vdash \mathtt{exception}\ exbind \Rightarrow (VE, EE) \text{ in Env}} \tag{21}$$

$$\frac{C \vdash dec_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash dec_2 \Rightarrow E_2}{C \vdash \mathtt{local}\ dec_1\ \mathtt{in}\ dec_2\ \mathtt{end} \Rightarrow E_2} \tag{22}$$

$$\frac{C(longstrid_1) = (m_1, E_1) \quad \cdots \quad C(longstrid_n) = (m_n, E_n)}{C \vdash \mathtt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n} \tag{23}$$

$$\frac{}{C \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{24}$$

$$\frac{C \vdash dec_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash dec_2 \Rightarrow E_2}{C \vdash dec_1\ \langle ; \rangle\ dec_2 \Rightarrow E_1 + E_2} \tag{25}$$

Comments:

(17) Here *VE* will contain types rather than general type schemes. The closure of *VE* is exactly what allows variables to be used polymorphically, via rule 2.

Moreover, *U* is the set of explicit type variables scoped at this particular occurrence of `val` *valbind*, cf. Section 4.6, page 25. The side-condition

on U ensures that these explicit type variables are bound by the closure operation. On the other hand, no *other* explicit type variable occurring free in VE will become bound, since it must be in U of C, and is therefore excluded from closure by the definition of the closure operation (Section 4.8, page 26) since U of $C \subseteq \text{tyvars}\, C$.

(19),(20) 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.

(20) The Abs operation was defined in Section 4.9, page 27.

(21) No closure operation is used here, since EE maps exception names to types rather than to general type schemes. Note that EE is also recorded in the VarEnv component of the resulting environment (see Section 4.3, page 22).

Value Bindings

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

$$\frac{C \vdash \mathit{pat} \Rightarrow (VE, \tau) \qquad C \vdash \mathit{exp} \Rightarrow \tau \qquad \langle C \vdash \mathit{valbind} \Rightarrow VE' \rangle}{C \vdash \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle \texttt{and}\ \mathit{valbind} \rangle \Rightarrow VE\ \langle + VE' \rangle} \tag{26}$$

$$\frac{C + VE \vdash \mathit{valbind} \Rightarrow VE}{C \vdash \texttt{rec}\ \mathit{valbind} \Rightarrow VE} \tag{27}$$

Comments:

(26) When the option is present we have $\text{Dom}\, VE \cap \text{Dom}\, VE' = \emptyset$ by the syntactic restrictions.

(27) Modifying C by VE on the left captures the recursive nature of the binding. From rule 26 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 ascribed the same type.

Type Bindings

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

$$\frac{\mathit{tyvarseq} = \alpha^{(k)} \qquad C \vdash \mathit{ty} \Rightarrow \tau \qquad \langle C \vdash \mathit{typbind} \Rightarrow TE \rangle}{\begin{array}{l} C \vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{typbind} \rangle \Rightarrow \\ \qquad \{\mathit{tycon} \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\}\ \langle + TE \rangle \end{array}} \tag{28}$$

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

Data Type Bindings

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

$$\frac{\begin{array}{c} tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)}t \vdash conbind \Rightarrow CE \\ \langle C \vdash datbind \Rightarrow VE, TE \qquad \forall (t', CE) \in \mathrm{Ran}\, TE, t \neq t' \rangle \end{array}}{\begin{array}{l} C \vdash tyvarseq\ tycon\ \texttt{=}\ conbind\ \langle \texttt{and}\ datbind \rangle \Rightarrow \\ \qquad \mathrm{Clos}CE\langle + VE \rangle,\ \{tycon \mapsto (t, \mathrm{Clos}CE)\}\ \langle + TE \rangle \end{array}} \tag{29}$$

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

Constructor Bindings

$\boxed{C, \tau \vdash conbind \Rightarrow CE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash conbind \Rightarrow CE \rangle\rangle}{\begin{array}{l} C, \tau \vdash con\ \langle \texttt{of}\ ty \rangle\ \langle\langle\ \texttt{|}\ conbind \rangle\rangle \Rightarrow \\ \qquad \{con \mapsto \tau\}\ \langle + \{con \mapsto \tau' \to \tau\}\ \rangle\ \langle\langle + CE \rangle\rangle \end{array}} \tag{30}$$

Comment: By the syntactic restrictions $con \notin \mathrm{Dom}\, CE$.

Exception Bindings

$\boxed{C \vdash exbind \Rightarrow EE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau \quad \tau \text{ is imperative} \rangle \qquad \langle\langle C \vdash exbind \Rightarrow EE \rangle\rangle}{\begin{array}{l} C \vdash excon\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{and}\ exbind \rangle\rangle \Rightarrow \\ \qquad \{excon \mapsto \texttt{exn}\}\ \langle + \{excon \mapsto \tau \to \texttt{exn}\}\ \rangle\ \langle\langle + EE \rangle\rangle \end{array}} \tag{31}$$

$$\frac{C(longexcon) = \tau \qquad \langle C \vdash exbind \Rightarrow EE \rangle}{C \vdash excon\ \texttt{=}\ longexcon\ \langle \texttt{and}\ exbind \rangle \Rightarrow \{excon \mapsto \tau\}\ \langle + EE \rangle} \tag{32}$$

Comments:

(31) Notice that τ must not contain any applicative type variables.

(31),(32) There is a unique EE, for each C and $exbind$, such that $C \vdash exbind \Rightarrow EE$.

Atomic Patterns

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

$$\frac{}{C \vdash \_ \Rightarrow (\{\}, \tau)} \tag{33}$$

$$\frac{}{C \vdash scon \Rightarrow (\{\}, \text{type}(scon))} \tag{34}$$

$$\frac{}{C \vdash var \Rightarrow (\{var \mapsto \tau\}, \tau)} \tag{35}$$

$$\frac{C(longcon) \succ \tau^{(k)}t}{C \vdash longcon \Rightarrow (\{\}, \tau^{(k)}t)} \tag{36}$$

$$\frac{C(longexcon) = \texttt{exn}}{C \vdash longexcon \Rightarrow (\{\}, \texttt{exn})} \tag{37}$$

$$\frac{\langle C \vdash patrow \Rightarrow (VE, \varrho)\rangle}{C \vdash \texttt{\{}\ \langle patrow\rangle\ \texttt{\}} \Rightarrow (\ \{\}\langle + VE\rangle,\ \{\}\langle + \varrho\rangle \text{ in Type }\ )} \tag{38}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau)}{C \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow (VE, \tau)} \tag{39}$$

Comments:

(35) Note that *var* can assume a type, not a general type scheme.

Pattern Rows

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

$$\frac{}{C \vdash \ldots \Rightarrow (\{\}, \varrho)} \tag{40}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad \langle C \vdash patrow \Rightarrow (VE', \varrho) \qquad lab \notin \text{Dom}\,\varrho\rangle}{C \vdash lab\ \texttt{=}\ pat\ \langle\ \texttt{,}\ patrow\rangle \Rightarrow (VE\langle + VE'\rangle,\ \{lab \mapsto \tau\}\langle + \varrho\rangle)} \tag{41}$$

Comment:

(41) By the syntactic restrictions, $\text{Dom}\,VE \cap \text{Dom}\,VE' = \emptyset$.

Patterns

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

$$\frac{C \vdash atpat \Rightarrow (VE, \tau)}{C \vdash atpat \Rightarrow (VE, \tau)} \tag{42}$$

$$\frac{C(longcon) \succ \tau' \to \tau \qquad C \vdash atpat \Rightarrow (VE, \tau')}{C \vdash longcon\ atpat \Rightarrow (VE, \tau)} \tag{43}$$

$$\frac{C(longexcon) = \tau \to \texttt{exn} \qquad C \vdash atpat \Rightarrow (VE, \tau)}{C \vdash longexcon\ atpat \Rightarrow (VE, \texttt{exn})} \tag{44}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C \vdash ty \Rightarrow \tau}{C \vdash pat\ \texttt{:}\ ty \Rightarrow (VE, \tau)} \tag{45}$$

$$\frac{\begin{array}{c} C \vdash var \Rightarrow (VE, \tau) \qquad \langle C \vdash ty \Rightarrow \tau \rangle \\ C \vdash pat \Rightarrow (VE', \tau) \end{array}}{C \vdash var \langle \texttt{:}\ ty \rangle\ \texttt{as}\ pat \Rightarrow (VE + VE', \tau)} \tag{46}$$

Comments:

(46) By the syntactic restrictions, $\mathrm{Dom}\, VE \cap \mathrm{Dom}\, VE' = \emptyset$.

Type Expressions

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

$$\frac{tyvar = \alpha}{C \vdash tyvar \Rightarrow \alpha} \tag{47}$$

$$\frac{\langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash \texttt{\{}\ \langle tyrow \rangle\ \texttt{\}} \Rightarrow \{\}\langle + \varrho \rangle \text{ in Type}} \tag{48}$$

$$\frac{\begin{array}{c} tyseq = ty_1 \cdots ty_k \qquad C \vdash ty_i \Rightarrow \tau_i\ (1 \le i \le k) \\ C(longtycon) = (\theta, CE) \end{array}}{C \vdash tyseq\ longtycon \Rightarrow \tau^{(k)}\theta} \tag{49}$$

$$\frac{C \vdash ty \Rightarrow \tau \qquad C \vdash ty' \Rightarrow \tau'}{C \vdash ty\ \texttt{->}\ ty' \Rightarrow \tau \to \tau'} \tag{50}$$

$$\frac{C \vdash ty \Rightarrow \tau}{C \vdash \texttt{(}\ ty\ \texttt{)} \Rightarrow \tau} \tag{51}$$

Comments:

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

Type-expression Rows $\boxed{C \vdash tyrow \Rightarrow \varrho}$

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash lab \ : \ ty \ \langle \ , \ tyrow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \ \varrho \rangle} \tag{52}$$

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

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 $\{lab_1 = pat_1, \cdots, lab_m = pat_m, \ldots\}$ the program context must determine uniquely the domain $\{lab_1, \cdots, lab_n\}$ of its record type, where $m \leq n$; thus, the context must determine the labels $\{lab_{m+1}, \cdots, lab_n\}$ of the fields to be matched by the wildcard. For this purpose, an explicit type constraint may be needed. This restriction is necessary to ensure the existence of principal type schemes.

2. In a match of the form pat_1 `=>` exp_1 `|` $\cdots$ `|` $\mathtt{pat}_n$ `=>` exp_n the pattern sequence $pat_1, \ldots, pat_n$ 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 `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 binding $var\ atpat_1 \cdots atpat_n \langle : ty \rangle$ `=` exp (consisting of one clause only), each separate $atpat_i$ should be exhaustive by itself.

4.12 Principal Environments

The notion of *enrichment*, $E \succ E'$, between environments $E = (SE, TE, VE, EE)$ and $E' = (SE', TE', VE', EE')$ is defined in Section 5.11. For the present section, $E \succ E'$ may be taken to mean $SE = SE' = \{\}$, $TE = TE'$, $EE = EE'$, $\mathrm{Dom}\,VE = \mathrm{Dom}\,VE'$ and, for each $id \in \mathrm{Dom}\,VE$, $VE(id) \succ VE'(id)$.

Let C be a context, and suppose that $C \vdash dec \Rightarrow E$ according to the preceding Inference Rules. Then E is *principal* (for *dec* in the context C)

if, for all E' for which $C \vdash dec \Rightarrow E'$, we have $E \succ E'$. We claim that if *dec* elaborates to any environment in C then it elaborates to a principal environment in C. Strictly, we must allow for the possibility that type names and imperative type variables which do not occur in C are chosen differently for E and E'. The stated claim is therefore made up to such variation.

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.

$$\begin{array}{rcl} M & \in & \text{StrNameSet} = \text{Fin}(\text{StrName}) \\ N \text{ or } (M,T) & \in & \text{NameSet} = \text{StrNameSet} \times \text{TyNameSet} \\ \Sigma \text{ or } (N)S & \in & \text{Sig} = \text{NameSet} \times \text{Str} \\ \Phi \text{ or } (N)(S,(N')S') & \in & \text{FunSig} = \text{NameSet} \times (\text{Str} \times \text{Sig}) \\ G & \in & \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Sig} \\ F & \in & \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunSig} \\ B \text{ or } N,F,G,E & \in & \text{Basis} = \text{NameSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env} \end{array}$$

Figure 11: Further Compound Semantic Objects

The prefix (N), in signatures and functor signatures, binds both type names and structure names. We shall always consider a set N of names as partitioned into a pair (M,T) of sets of the two kinds of name.

It is sometimes convenient to work with an arbitrary semantic object A, or assembly A of such objects. As with the function tynames, strnames(A) and names(A) denote respectively the set of structure names and the set of names occurring free in A.

Certain operations require a change of bound names in semantic objects; see for example Section 5.7. When bound type names are changed, we demand that all of their attributes (i.e. imperative, equality and arity) are preserved.

For any structure $S = (m,(SE,TE,VE,EE))$ we call m the *structure name* or *name* of S; also, the *proper substructures* of S are the members of $\operatorname{Ran} SE$ and their proper substructures. The *substructures* of S are S itself and its proper substructures. The structures *occurring in* an object or assembly A are the structures and substructures from which it is built.

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 N of B to include the type names and structure names of E (or of SE say). We therefore define $B \oplus SE$, for example, to mean $B + (\text{names}\, SE, SE)$.

5.2 Consistency

A set of type structures is said to be *consistent* if, for all (θ_1, CE_1) and (θ_2, CE_2) in the set, if $\theta_1 = \theta_2$ then

$$CE_1 = \{\} \text{ or } CE_2 = \{\} \text{ or } \text{Dom}\, CE_1 = \text{Dom}\, CE_2$$

A semantic object A or assembly A of objects is said to be *consistent* if (after changing bound names to make all nameset prefixes in A disjoint) for all S_1 and S_2 occurring in A and for every *longstrid* and every *longtycon*

1. If m of $S_1 = m$ of S_2, and both $S_1(longstrid)$ and $S_2(longstrid)$ exist, then
$$m \text{ of } S_1(longstrid) \;=\; m \text{ of } S_2(longstrid)$$

2. If m of $S_1 = m$ of S_2, and both $S_1(longtycon)$ and $S_2(longtycon)$ exist, then
$$\theta \text{ of } S_1(longtycon) \;=\; \theta \text{ of } S_2(longtycon)$$

3. The set of all type structures in A is consistent

As an example, a functor signature $(N)(S, (N')S')$ is consistent if, assuming first that $N \cap N' = \emptyset$, the assembly $A = \{S, S'\}$ is consistent.

We may loosely say that two structures S_1 and S_2 are consistent if $\{S_1, S_2\}$ is consistent, but must remember that this is stronger than the assertion that S_1 is consistent and S_2 is consistent.

Note that if A is a consistent assembly and $A' \subset A$ then A' is also a consistent assembly.

5.3 Well-formedness

A signature $(N)S$ is *well-formed* if $N \subseteq \text{names}\, S$, and also, whenever (m, E) is a substructure of S and $m \notin N$, then $N \cap (\text{names}\, E) = \emptyset$. A functor signature $(N)(S, (N')S')$ is *well-formed* if $(N)S$ and $(N')S'$ are well-formed,

and also, whenever (m', E') is a substructure of S' and $m' \notin N \cup N'$, then $(N \cup N') \cap (\text{names}\, E') = \emptyset$.

An object or assembly A is *well-formed* if every type environment, signature and functor signature occurring in A is well-formed.

5.4 Cycle-freedom

An object or assembly A is *cycle-free* if it contains no cycle of structure names; that is, there is no sequence

$$m_0, \cdots, m_{k-1}, m_k = m_0 \quad (k > 0)$$

of structure names such that, for each i $(0 \leq i < k)$ some structure with name m_i occurring in A has a proper substructure with name m_{i+1}.

5.5 Admissibility

An object or assembly A is *admissible* if it is consistent, well-formed and cycle-free. Henceforth it is assumed that all objects mentioned are admissible. We also require that

1. In every sentence $A \vdash \mathit{phrase} \Rightarrow A'$ inferred by the rules given in Section 5.14, the assembly $\{A, A'\}$ is admissible.

2. In the special case of a sentence $B \vdash \mathit{sigexp} \Rightarrow S$, we further require that the assembly consisting of all semantic objects occurring in the entire inference of this sentence be admissible. This is important for the definition of principal signatures in Section 5.13.

In our semantic definition we have not undertaken to indicate how admissibility should be checked in an implementation.

5.6 Type Realisation

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

The *support* $\text{Supp}\, \varphi_{\text{Ty}}$ of a type realisation φ_{Ty} is the set of type names t for which $\varphi_{\text{Ty}}(t) \neq t$.

5.7 Realisation

A *realisation* is a function φ of names, partitioned into a type realisation $\varphi_{\mathrm{Ty}} : \mathrm{TyName} \to \mathrm{TypeFcn}$ and a function $\varphi_{\mathrm{Str}} : \mathrm{StrName} \to \mathrm{StrName}$. The *support* $\mathrm{Supp}\,\varphi$ of a realisation φ is the set of names n for which $\varphi(n) \neq n$. The *yield* $\mathrm{Yield}\,\varphi$ of a realisation φ is the set of names which occur in some $\varphi(n)$ for which $n \in \mathrm{Supp}\,\varphi$.

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

$$N \cap (\mathrm{Supp}\,\varphi \cup \mathrm{Yield}\,\varphi) = \emptyset\ .$$

5.8 Type Explication

A signature $(N)S$ is *type-explicit* if, whenever $t \in N$ and occurs free in S, then some substructure of S contains a type environment TE such that $TE(tycon) = (t, CE)$ for some $tycon$ and some CE.

5.9 Signature Instantiation

A structure S_2 *is an instance of* a signature $\Sigma_1 = (N_1)S_1$, written $\Sigma_1 \geq S_2$, if there exists a realisation φ such that $\varphi(S_1) = S_2$ and $\mathrm{Supp}\,\varphi \subseteq N_1$. (Note that if Σ_1 is type-explicit then there is at most one such φ.) A signature $\Sigma_2 = (N_2)S_2$ *is an instance of* $\Sigma_1 = (N_1)S_1$, written $\Sigma_1 \geq \Sigma_2$, if $\Sigma_1 \geq S_2$ and $N_2 \cap (\mathrm{names}\,\Sigma_1) = \emptyset$. It can be shown that $\Sigma_1 \geq \Sigma_2$ iff, for all S, whenever $\Sigma_2 \geq S$ then $\Sigma_1 \geq S$.

5.10 Functor Signature Instantiation

A pair $(S, (N')S')$ is called a *functor instance*. Given $\Phi = (N_1)(S_1, (N_1')S_1')$, a functor instance $(S_2, (N_2')S_2')$ is an *instance* of Φ, written $\Phi \geq (S_2, (N_2')S_2')$, if there exists a realisation φ such that $\varphi(S_1, (N_1')S_1') = (S_2, (N_2')S_2')$ and $\mathrm{Supp}\,\varphi \subseteq N_1$.

5.11 Enrichment

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

A structure $S_1 = (m_1, E_1)$ *enriches* another structure $S_2 = (m_2, E_2)$, written $S_1 \succ S_2$, if

1. $m_1 = m_2$
2. $E_1 \succ E_2$

An environment $E_1 = (SE_1, TE_1, VE_1, EE_1)$ *enriches* another environment $E_2 = (SE_2, TE_2, VE_2, EE_2)$, written $E_1 \succ E_2$, if

1. $\mathrm{Dom}\, SE_1 \supseteq \mathrm{Dom}\, SE_2$, and $SE_1(strid) \succ SE_2(strid)$ for all $strid \in \mathrm{Dom}\, SE_2$
2. $\mathrm{Dom}\, TE_1 \supseteq \mathrm{Dom}\, TE_2$, and $TE_1(tycon) \succ TE_2(tycon)$ for all $tycon \in \mathrm{Dom}\, TE_2$
3. $\mathrm{Dom}\, VE_1 \supseteq \mathrm{Dom}\, VE_2$, and $VE_1(id) \succ VE_2(id)$ for all $id \in \mathrm{Dom}\, VE_2$
4. $\mathrm{Dom}\, EE_1 \supseteq \mathrm{Dom}\, EE_2$, and $EE_1(excon) = EE_2(excon)$ for all $excon \in \mathrm{Dom}\, EE_2$

Finally, a type structure (θ_1, CE_1) *enriches* another type structure (θ_2, CE_2), written $(\theta_1, CE_1) \succ (\theta_2, CE_2)$, if

1. $\theta_1 = \theta_2$
2. Either $CE_1 = CE_2$ or $CE_2 = \{\}$

5.12 Signature Matching

A structure S *matches* a signature Σ_1 if there exists a structure S^- such that $\Sigma_1 \geq S^- \prec S$. Thus matching is a combination of instantiation and enrichment. There is at most one such S^-, given Σ_1 and S. Moreover, writing $\Sigma_1 = (N_1)S_1$, if $\Sigma_1 \geq S^-$ then there exists a realisation φ with $\mathrm{Supp}\, \varphi \subseteq N_1$ and $\varphi(S_1) = S^-$. We shall then say that S matches Σ_1 *via* φ. (Note that if Σ_1 is type-explicit then φ is uniquely determined by Σ_1 and S.)

A signature Σ_2 *matches* a signature Σ_1 if for all structures S, if S matches Σ_2 then S matches Σ_1. It can be shown that $\Sigma_2 = (N_2)S_2$ matches $\Sigma_1 = (N_1)S_1$ if and only if there exists a realisation φ with $\mathrm{Supp}\,\varphi \subseteq N_1$ and $\varphi(S_1) \prec S_2$ and $N_2 \cap \mathrm{names}\,\Sigma_1 = \emptyset$.

5.13 Principal Signatures

The definitions in this section concern the elaboration of signature expressions; more precisely they concern inferences of sentences of the form $B \vdash \mathit{sigexp} \Rightarrow S$, where S is a structure and B is a basis. Recall, from Section 5.5, that the assembly of all semantic objects in such an inference must be admissible.

For any basis B and any structure S, we say that B *covers* S if for every substructure (m, E) of S such that $m \in N$ of B:

1. For every structure identifier $\mathit{strid} \in \mathrm{Dom}\,E$, B contains a substructure (m, E') with m free and $\mathit{strid} \in \mathrm{Dom}\,E'$

2. For every type constructor $\mathit{tycon} \in \mathrm{Dom}\,E$, B contains a substructure (m, E') with m free and $\mathit{tycon} \in \mathrm{Dom}\,E'$

(This condition is not a consequence of consistency of $\{B, S\}$; informally, it states that if S shares a substructure with B, then S mentions no more components of the substructure than B does.)

We say that a signature $(N)S$ is *principal for sigexp in* B if, choosing N so that $(N \text{ of } B) \cap N = \emptyset$,

1. B covers S

2. $B \vdash \mathit{sigexp} \Rightarrow S$

3. Whenever $B \vdash \mathit{sigexp} \Rightarrow S'$, then $(N)S \geq S'$

We claim that if *sigexp* elaborates in B to some structure covered by B, then it possesses a principal signature in B.

Analogous to the definition given for type environments in Section 4.9, we say that a semantic object A *respects equality* if every type environment occurring in A respects equality.

Now let us assume that *sigexp* possesses a principal signature $\Sigma_0 = (N_0)S_0$ in B. We wish to define, in terms of Σ_0, another signature Σ which provides

more information about the equality attributes of structures which will match Σ_0. To this end, let T_0 be the set of type names $t \in N_0$ which do not admit equality, and such that (t, CE) occurs in S_0 for some $CE \neq \{\}$. Then we say Σ is *equality-principal for sigexp in B* if

1. Σ respects equality
2. Σ is obtained from Σ_0 just by making as many members of T_0 admit equality as possible, subject to 1. above

It is easy to show that, if any such Σ exists, it is determined uniquely by Σ_0; moreover, Σ exists if Σ_0 itself respects equality.

5.14 Inference Rules

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

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

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 $B = N, F, G, E$ in which a *topdec* is elaborated has the property that $\mathrm{names}\, F \cup \mathrm{names}\, G \cup \mathrm{names}\, E \subseteq N$. This is not the case for bases in which signature expressions and specifications are elaborated, but the following Theorem can be proved:

> Let S be an inferred sentence $B \vdash \mathit{topdec} \Rightarrow B'$ in which B satisfies the above condition. Then B' also satisfies the condition.
>
> Moreover, if S′ is a sentence of the form $B'' \vdash \mathit{phrase} \Rightarrow A$ occurring in a proof of S, where *phrase* is either a structure expression or a structure-level declaration, then B'' also satisfies the condition.
>
> Finally, if $T, U, E \vdash \mathit{phrase} \Rightarrow A$ occurs in a proof of S, where *phrase* is a phrase of the Core, then $\mathrm{tynames}\, E \subseteq T$.

Structure Expressions $\boxed{B \vdash \mathit{strexp} \Rightarrow S}$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad m \notin (N \text{ of } B) \cup \mathrm{names}\, E}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow (m, E)} \tag{53}$$

$$\frac{B(\mathit{longstrid}) = S}{B \vdash \mathit{longstrid} \Rightarrow S} \tag{54}$$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow S \\ B(\mathit{funid}) \geq (S'', (N')S')\ ,\ S \succ S'' \\ (N \text{ of } B) \cap N' = \emptyset \end{array}}{B \vdash \mathit{funid}\ (\ \mathit{strexp}\ )\ \Rightarrow S'} \tag{55}$$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad B \oplus E \vdash \mathit{strexp} \Rightarrow S}{B \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow S} \tag{56}$$

Comments:

(53) The side condition ensures that each generative structure expression receives a new name. If the expression occurs in a functor body the structure name will be bound by (N') in rule 99; this will ensure that for each application of the functor, by rule 55, a new distinct name will be chosen for the structure generated.

(55) The side condition $(N \text{of} B) \cap N' = \emptyset$ can always be satisfied by renaming bound names in $(N')S'$ thus ensuring that the generated structures receive new names.

Let $B(\mathit{funid}) = (N)(S_f, (N')S'_f)$. Assuming that $(N)S_f$ is type-explicit, the realisation φ for which $\varphi(S_f, (N')S'_f) = (S'', (N')S')$ is uniquely determined by S, since $S \succ S''$ can only hold if the type names and structure names in S and S'' agree. Recall that enrichment $\succ$ allows more components and more polymorphism, while instantiation $\geq$ does not.

Sharing between argument and result specified in the declaration of the functor *funid* is represented by the occurrence of the same name in both S_f and S'_f, and this repeated occurrence is preserved by φ, yielding sharing between the argument structure S and the result structure S' of this functor application.

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

Structure-level Declarations $\boxed{B \vdash \mathit{strdec} \Rightarrow E}$

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E \quad E \text{ principal for } \mathit{dec} \text{ in } (C \text{ of } B)}{B \vdash \mathit{dec} \Rightarrow E} \tag{57}$$

$$\frac{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}}{B \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow \mathit{SE} \text{ in Env}} \tag{58}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2} \tag{59}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\} \text{ in Env}} \tag{60}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \mathit{strdec}_1\ \langle ; \rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2} \tag{61}$$

Comments:

(57) The side condition ensures that all type schemes in E are as general as possible.

Structure Bindings

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

$$\frac{B \vdash \mathit{strexp} \Rightarrow S \qquad \langle B \vdash \mathit{sigexp} \Rightarrow \Sigma\ ,\ \Sigma \geq S' \prec S \rangle \qquad \langle\langle B + \text{names}\, S \vdash \mathit{strbind} \Rightarrow \mathit{SE} \rangle\rangle}{B \vdash \mathit{strid}\ \langle \texttt{:}\ \mathit{sigexp} \rangle\ \texttt{=}\ \mathit{strexp}\ \langle\langle \texttt{and}\ \mathit{strbind} \rangle\rangle \Rightarrow \{\mathit{strid} \mapsto S\langle' \rangle\}\ \langle\langle + \mathit{SE} \rangle\rangle} \tag{62}$$

Comment: If present, *sigexp* has the effect of restricting the view which *strid* provides of S while retaining sharing of names. The notation $S\langle' \rangle$ means S', if the first option is present, and S if not.

Signature Expressions

$\boxed{B \vdash \mathit{sigexp} \Rightarrow S}$

$$\frac{B \vdash \mathit{spec} \Rightarrow E}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow (m, E)} \tag{63}$$

$$\frac{B(\mathit{sigid}) \geq S}{B \vdash \mathit{sigid} \Rightarrow S} \tag{64}$$

Comments:

(63) In contrast to rule 53, m is not here required to be new. The name m may be chosen to achieve the sharing required in rule 88, or to achieve the enrichment side conditions of rule 62 or 99. The choice of m must result in an admissible object.

(64) The instance S of $B(\mathit{sigid})$ is not determined by this rule, but – as in rule 63 – the instance may be chosen to achieve sharing properties or enrichment conditions.

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

$$\frac{B \vdash \mathit{sigexp} \Rightarrow S \quad (N)S \text{ equality-principal for } \mathit{sigexp} \text{ in } B \quad (N)S \text{ type-explicit}}{B \vdash \mathit{sigexp} \Rightarrow (N)S} \tag{65}$$

Comment: A signature expression *sigexp* which is an immediate constituent of a structure binding, a signature binding, a functor binding or a functor signature is elaborated to an equality-principal and type-explicit signature, see rules 62, 69, 95 and 99. By contrast, signature expressions occurring in structure descriptions are elaborated to structures using the liberal rules 63 and 64, see rule 87, so that names can be chosen to achieve sharing, when necessary.

Signature Declarations

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

$$\frac{B \vdash \mathit{sigbind} \Rightarrow G}{B \vdash \texttt{signature}\ \mathit{sigbind} \Rightarrow G} \tag{66}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\}} \tag{67}$$

$$\frac{B \vdash \mathit{sigdec}_1 \Rightarrow G_1 \qquad B + G_1 \vdash \mathit{sigdec}_2 \Rightarrow G_2}{B \vdash \mathit{sigdec}_1\ \langle\texttt{;}\rangle\ \mathit{sigdec}_2 \Rightarrow G_1 + G_2} \tag{68}$$

Comments:

(66) The first closure restriction of Section 3.6 can be enforced by replacing the B in the premise by $B_0 + G$ of B.

(68) A signature declaration does not create any new structures or types; hence the use of $+$ instead of $\oplus$.

Signature Bindings

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

$$\frac{B \vdash \mathit{sigexp} \Rightarrow \Sigma \qquad \langle B \vdash \mathit{sigbind} \Rightarrow G\rangle}{B \vdash \mathit{sigid}\ \texttt{=}\ \mathit{sigexp}\ \langle\texttt{and}\ \mathit{sigbind}\rangle \Rightarrow \{\mathit{sigid} \mapsto \Sigma\}\ \langle + G\rangle} \tag{69}$$

Comment: The condition that Σ be equality-principal, implicit in the first premise, ensures that the signature found is as general as possible given the sharing constraints present in *sigexp*.

Specifications

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

$$\frac{C \text{ of } B \vdash \mathit{valdesc} \Rightarrow VE}{B \vdash \texttt{val}\ \mathit{valdesc} \Rightarrow \mathrm{Clos}VE \text{ in Env}} \tag{70}$$

$$\frac{C \text{ of } B \vdash \mathit{typdesc} \Rightarrow TE}{B \vdash \texttt{type}\ \mathit{typdesc} \Rightarrow TE \text{ in Env}} \tag{71}$$

$$\frac{C \text{ of } B \vdash \mathit{typdesc} \Rightarrow TE \qquad \forall(\theta, CE) \in \mathrm{Ran}\, TE,\ \theta \text{ admits equality}}{B \vdash \texttt{eqtype}\ \mathit{typdesc} \Rightarrow TE \text{ in Env}} \tag{72}$$

$$\frac{C \text{ of } B + TE \vdash \mathit{datdesc} \Rightarrow VE, TE}{B \vdash \texttt{datatype}\ \mathit{datdesc} \Rightarrow (VE, TE) \text{ in Env}} \tag{73}$$

$$\frac{C \text{ of } B \vdash \mathit{exdesc} \Rightarrow EE \quad VE = EE}{B \vdash \texttt{exception}\ \mathit{exdesc} \Rightarrow (VE, EE) \text{ in Env}} \tag{74}$$

$$\frac{B \vdash \mathit{strdesc} \Rightarrow SE}{B \vdash \texttt{structure}\ \mathit{strdesc} \Rightarrow SE \text{ in Env}} \tag{75}$$

$$\frac{B \vdash \mathit{shareq} \Rightarrow \{\}}{B \vdash \texttt{sharing}\ \mathit{shareq} \Rightarrow \{\} \text{ in Env}} \tag{76}$$

$$\frac{B \vdash \mathit{spec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{spec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{spec}_1\ \texttt{in}\ \mathit{spec}_2\ \texttt{end} \Rightarrow E_2} \tag{77}$$

$$\frac{B(\mathit{longstrid}_1) = (m_1, E_1) \quad \cdots \quad B(\mathit{longstrid}_n) = (m_n, E_n)}{B \vdash \texttt{open}\ \mathit{longstrid}_1 \cdots \mathit{longstrid}_n \Rightarrow E_1 + \cdots + E_n} \tag{78}$$

$$\frac{B(\mathit{sigid}_1) \geq (m_1, E_1) \quad \cdots \quad B(\mathit{sigid}_n) \geq (m_n, E_n)}{B \vdash \texttt{include}\ \mathit{sigid}_1 \cdots \mathit{sigid}_n \Rightarrow E_1 + \cdots + E_n} \tag{79}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{80}$$

$$\frac{B \vdash \mathit{spec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{spec}_2 \Rightarrow E_2}{B \vdash \mathit{spec}_1\ \langle ; \rangle\ \mathit{spec}_2 \Rightarrow E_1 + E_2} \tag{81}$$

Comments:

(70) VE is determined by B and *valdesc*.

(71)–(73) The type functions in TE may be chosen to achieve the sharing hypothesis of rule 89 or the enrichment conditions of rules 62 and 99. In particular, the type names in TE in rule 73 need not be new. Also, in rule 71 the type functions in TE may admit equality.

(74) EE is determined by B and *exdesc* and contains monotypes only.

(79) The names m_i in the instances may be chosen to achieve sharing or enrichment conditions.

Value Descriptions $\boxed{C \vdash valdesc \Rightarrow VE}$

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash valdesc \Rightarrow VE \rangle}{C \vdash var \,:\, ty\ \langle \texttt{and}\ valdesc \rangle \Rightarrow \{var \mapsto \tau\}\ \langle + VE \rangle} \tag{82}$$

Type Descriptions $\boxed{C \vdash typdesc \Rightarrow TE}$

$$\frac{tyvarseq = \alpha^{(k)} \qquad \langle C \vdash typdesc \Rightarrow TE \rangle \qquad \text{arity}\,\theta = k}{C \vdash tyvarseq\ tycon\ \langle \texttt{and}\ typdesc \rangle \Rightarrow \{tycon \mapsto (\theta, \{\})\}\ \langle + TE \rangle} \tag{83}$$

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

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

$$\frac{tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)} t \vdash condesc \Rightarrow CE \qquad \langle C \vdash datdesc \Rightarrow VE, TE \rangle}{\begin{array}{l} C \vdash tyvarseq\ tycon\ \texttt{=}\ condesc\ \langle \texttt{and}\ datdesc \rangle \Rightarrow \\ \quad \text{Clos}CE\langle + VE \rangle,\ \{tycon \mapsto (t, \text{Clos}CE)\}\ \langle + TE \rangle \end{array}} \tag{84}$$

Constructor Descriptions $\boxed{C, \tau \vdash condesc \Rightarrow CE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash condesc \Rightarrow CE \rangle\rangle}{\begin{array}{l} C, \tau \vdash con\ \langle \texttt{of}\ ty \rangle\ \langle\langle\ \texttt{|}\ condesc \rangle\rangle \Rightarrow \\ \quad \{con \mapsto \tau\}\ \langle + \{con \mapsto \tau' \to \tau\}\ \rangle\ \langle\langle + CE \rangle\rangle \end{array}} \tag{85}$$

Exception Descriptions $\boxed{C \vdash exdesc \Rightarrow EE}$

$$\frac{\langle C \vdash ty \Rightarrow \tau \qquad \text{tyvars}(\tau) = \emptyset \rangle \qquad \langle\langle C \vdash exdesc \Rightarrow EE \rangle\rangle}{\begin{array}{l} C \vdash excon\ \langle \texttt{of}\ ty \rangle\ \ \langle\langle \texttt{and}\ exdesc \rangle\rangle \Rightarrow \\ \quad \{excon \mapsto \texttt{exn}\}\ \langle + \{excon \mapsto \tau \to \texttt{exn}\} \rangle\ \langle\langle + EE \rangle\rangle \end{array}} \tag{86}$$

Structure Descriptions $\boxed{B \vdash strdesc \Rightarrow SE}$

$$\frac{B \vdash sigexp \Rightarrow S \qquad \langle B \vdash strdesc \Rightarrow SE \rangle}{B \vdash strid\ :\ sigexp\ \langle \texttt{and}\ strdesc \rangle \Rightarrow \{strid \mapsto S\}\ \langle + SE \rangle} \tag{87}$$

Sharing Equations $\boxed{B \vdash shareq \Rightarrow \{\}}$

$$\frac{m \text{ of } B(longstrid_1) = \cdots = m \text{ of } B(longstrid_n)}{B \vdash longstrid_1\ \texttt{=}\ \cdots\ \texttt{=}\ longstrid_n\ \Rightarrow \{\}} \tag{88}$$

$$\frac{\theta \text{ of } B(longtycon_1) = \cdots = \theta \text{ of } B(longtycon_n)}{B \vdash \texttt{type}\ longtycon_1\ \texttt{=}\ \cdots\ \texttt{=}\ longtycon_n\ \Rightarrow \{\}} \tag{89}$$

$$\frac{B \vdash shareq_1 \Rightarrow \{\} \qquad B \vdash shareq_2 \Rightarrow \{\}}{B \vdash shareq_1\ \texttt{and}\ shareq_2 \Rightarrow \{\}} \tag{90}$$

Comments:

(88) By the definition of consistency the premise is weaker than $B(longstrid_1) = \cdots = B(longstrid_n)$. Two different structures with the same name may be thought of as representing different views. The requirement that B is consistent forces different views to be consistent.

(89) By the definition of consistency the premise is weaker than $B(longtycon_1) = \cdots = B(longtycon_n)$. A type structure with empty constructor environment may have the same type name as one with a non-empty constructor environment; the former could arise from a type description, and the latter from a datatype description. However, the requirement that B is consistent will prevent two type structures with constructor environments which have different non-empty domains from sharing the same type name.

Functor Specifications $\boxed{B \vdash \mathit{funspec} \Rightarrow F}$

$$\frac{B \vdash \mathit{fundesc} \Rightarrow F}{B \vdash \texttt{functor}\ \mathit{fundesc} \Rightarrow F} \tag{91}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\}} \tag{92}$$

$$\frac{B \vdash \mathit{funspec}_1 \Rightarrow F_1 \qquad B + F_1 \vdash \mathit{funspec}_2 \Rightarrow F_2}{B \vdash \mathit{funspec}_1\ \langle \texttt{;} \rangle\ \mathit{funspec}_2 \Rightarrow F_1 + F_2} \tag{93}$$

Comments:

(91) The second closure restriction of Section 3.6 can be enforced by replacing the B in the premise by $B_0 + G$ of B.

Functor Descriptions $\boxed{B \vdash \mathit{fundesc} \Rightarrow F}$

$$\frac{B \vdash \mathit{funsigexp} \Rightarrow \Phi \qquad \langle B \vdash \mathit{fundesc} \Rightarrow F \rangle}{B \vdash \mathit{funid}\ \mathit{funsigexp}\ \langle \texttt{and}\ \mathit{fundesc} \rangle \Rightarrow \{\mathit{funid} \mapsto \Phi\} \langle + F \rangle} \tag{94}$$

Functor Signature Expressions $\boxed{B \vdash \mathit{funsigexp} \Rightarrow \Phi}$

$$\frac{B \vdash \mathit{sigexp} \Rightarrow (N)S \qquad B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow (N')S'}{B \vdash \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{)}\ \texttt{:}\ \mathit{sigexp}' \Rightarrow (N)(S, (N')S')} \tag{95}$$

Comment: The signatures $(N)S$ and $(N')S'$ are equality-principal and type-explicit, see rule 65.

Functor Declarations $\boxed{B \vdash \mathit{fundec} \Rightarrow F}$

$$\frac{B \vdash \mathit{funbind} \Rightarrow F}{B \vdash \texttt{functor}\ \mathit{funbind} \Rightarrow F} \tag{96}$$

$$\frac{}{B \vdash \quad \Rightarrow \{\}} \tag{97}$$

$$\frac{B \vdash \mathit{fundec}_1 \Rightarrow F_1 \qquad B + F_1 \vdash \mathit{fundec}_2 \Rightarrow F_2}{B \vdash \mathit{fundec}_1\ \langle \texttt{;} \rangle\ \mathit{fundec}_2 \Rightarrow F_1 + F_2} \tag{98}$$

Comments:

(96) The third closure restriction of Section 3.6 can be enforced by replacing the B in the premise by $B_0 + (G \text{ of } B) + (F \text{ of } B)$.

Functor Bindings

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

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow (N)S \qquad B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{strexp} \Rightarrow S' \\ \langle B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow \Sigma',\ \Sigma' \geq S'' \prec S' \rangle \\ N' = \text{names}\, S' \setminus ((N \text{ of } B) \cup N) \\ \langle\langle B \vdash \mathit{funbind} \Rightarrow F \rangle\rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{)}\ \langle \texttt{:}\ \mathit{sigexp}' \rangle\ \texttt{=}\ \mathit{strexp}\ \langle\langle \texttt{and}\ \mathit{funbind} \rangle\rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, (N')S'\langle' \rangle)\}\ \langle\langle + F \rangle\rangle \end{array}} \tag{99}$$

Comment: The requirement that $(N)S$ be equality-principal, implicit in the first premise, forces $(N)S$ to be as general as possible given the sharing constraints in *sigexp*. The requirement that $(N)S$ be type-explicit ensures that there is at most one realisation via which an actual argument can match $(N)S$. Since $\oplus$ is used, any structure name m and type name t in S acts like a constant in the functor body; in particular, it ensures that further names generated during elaboration of the body are distinct from m and t. The set N' is chosen such that every name free in $(N)S$ or $(N)(S, (N')S')$ is free in B.

Top-level Declarations

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

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \quad \text{imptyvars}\, E = \emptyset}{B \vdash \mathit{strdec} \Rightarrow (\text{names}\, E, E) \text{ in Basis}} \tag{100}$$

$$\frac{B \vdash \mathit{sigdec} \Rightarrow G \quad \text{imptyvars}\, G = \emptyset}{B \vdash \mathit{sigdec} \Rightarrow (\text{names}\, G, G) \text{ in Basis}} \tag{101}$$

$$\frac{B \vdash \mathit{fundec} \Rightarrow F \quad \text{imptyvars}\, F = \emptyset}{B \vdash \mathit{fundec} \Rightarrow (\text{names}\, F, F) \text{ in Basis}} \tag{102}$$

Comments:

(100)–(102) The side conditions ensure that no free imperative type variables enter the basis.

5.15 Functor Signature Matching

As pointed out in Section 3.4 on the grammar for Modules, there is no phrase class whose elaboration requires matching one functor signature to another functor signature. But a precise definition of this matching is needed, since a functor g may only be separately compiled in the presence of specification of any functor f to which g refers, and then a real functor f must match this specification. In the case, then, that f has been specified by a functor signature

$$\Phi_1 \;=\; (N_1)(S_1, (N_1')S_1')$$

and that later f is declared with functor signature

$$\Phi_2 \;=\; (N_2)(S_2, (N_2')S_2')$$

the following matching rule will be employed:

A functor signature $\Phi_2 \;=\; (N_2)(S_2, (N_2')S_2')$ *matches* another functor signature, $\Phi_1 \;=\; (N_1)(S_1, (N_1')S_1')$, if there exists a realisation φ such that

1. $(N_1)S_1$ matches $(N_2)S_2$ via φ, and
2. $\varphi((N_2')S_2')$ matches $(N_1')S_1'$.

The first condition ensures that the real functor signature Φ_2 for f requires the argument *strexp* of any application $f(\textit{strexp})$ to have no more sharing, and no more richness, than was predicted by the specified signature Φ_1. The second condition ensures that the real functor signature Φ_2, instantiated to $(\varphi S_2, \varphi((N_2')S_2'))$, provides in the result of the application $f(\textit{strexp})$ no less sharing, and no less richness, than was predicted by the specified signature Φ_1.

6 Dynamic Semantics for the Core

6.1 Reduced Syntax

Since types are fully dealt with in the static semantics, the dynamic semantics ignores them. The Core syntax is therefore 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 exception bindings.
- Any declaration of the form "`type` *typbind*" or "`datatype` *datbind*" is replaced by the empty declaration.
- A declaration of the form "`abstype` *datbind* `with` *dec* `end`" is replaced by "*dec*".
- The Core phrase classes TypBind, DatBind, ConBind, 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 or real constant denotes a value according to normal mathematical conventions; each string constant denotes a sequence of ASCII characters as explained in

Section 2.2. The value denoted by *scon* is written 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, unlike value constructors which simply evaluate to themselves. This is to accommodate the generative nature of exception bindings; each evaluation of a declaration of a exception constructor binds it to a new unique name.

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 Con, Con $\times$ Val etc into Val. For records $r \in$ 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. An important point is that structure names m have no significance at all in the dynamic semantics; this explains why the object class Str = StrName $\times$ Env is absent here – for the dynamic semantics the concepts *structure* and *environment* coincide.

6.4 Basic Values

The basic values in BasVal are the values bound to predefined variables. These values are denoted by the identifiers to which they are bound in the initial dynamic basis (see Appendix D), and are as follows:

$$
\begin{array}{rcl}
v & \in & \mathrm{Val} = \{\texttt{:=}\} \cup \mathrm{SVal} \cup \mathrm{BasVal} \cup \mathrm{Con} \\
 & & \quad \cup (\mathrm{Con} \times \mathrm{Val}) \cup \mathrm{ExVal} \\
 & & \quad \cup \mathrm{Record} \cup \mathrm{Addr} \cup \mathrm{Closure} \\
r & \in & \mathrm{Record} = \mathrm{Lab} \xrightarrow{\text{fin}} \mathrm{Val} \\
e & \in & \mathrm{ExVal} = \mathrm{ExName} \cup (\mathrm{ExName} \times \mathrm{Val}) \\
[e] \text{ or } p & \in & \mathrm{Pack} = \mathrm{ExVal} \\
(match, E, VE) & \in & \mathrm{Closure} = \mathrm{Match} \times \mathrm{Env} \times \mathrm{VarEnv} \\
mem & \in & \mathrm{Mem} = \mathrm{Addr} \xrightarrow{\text{fin}} \mathrm{Val} \\
ens & \in & \mathrm{ExNameSet} = \mathrm{Fin}(\mathrm{ExName}) \\
(mem, ens) \text{ or } s & \in & \mathrm{State} = \mathrm{Mem} \times \mathrm{ExNameSet} \\
(SE, VE, EE) \text{ or } E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{VarEnv} \times \mathrm{ExConEnv} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\text{fin}} \mathrm{Env} \\
VE & \in & \mathrm{VarEnv} = \mathrm{Var} \xrightarrow{\text{fin}} \mathrm{Val} \\
EE & \in & \mathrm{ExConEnv} = \mathrm{ExCon} \xrightarrow{\text{fin}} \mathrm{ExName}
\end{array}
$$

Figure 13: Compound Semantic Objects

```
abs  floor  real  sqrt  sin  cos  arctan  exp  ln
   size  chr  ord  explode  implode  div  mod
      ~  /  *  +  -  =  <>  <  >  <=  >=
std_in  std_out  open_in  open_out  close_in  close_out
        input  output  lookahead  end_of_stream
```

The meaning of basic values (almost all of which are functions) is represented by the function

$$\mathrm{APPLY} \ : \ \mathrm{BasVal} \times \mathrm{Val} \to \mathrm{Val} \cup \mathrm{Pack}$$

which is detailed in Appendix D.

6.5 Basic Exceptions

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

```
Abs  Ord  Chr   Div  Mod  Quot  Prod
Neg  Sum  Diff  Floor  Sqrt  Exp  Ln
Io   Match  Bind  Interrupt
```

The exceptions on the first two lines are raised by corresponding basic functions, where `~ / * + -` correspond respectively to `Neg Quot Prod Sum Diff`. The details are given in Appendix D. The exception $(\texttt{Io}, s)$, where s is a string, is raised by certain of the basic input/output functions, as detailed in Appendix D. 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. Finally, `Interrupt` is raised by external intervention.

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.

For each value binding *pat* `=` *exp* the compiler must issue a report (but still compile) if *either* pat is not exhaustive *or* pat contains no variable. This will (on both counts) detect a mistaken declaration like `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, these warnings should not be given when the binding is a component of a top-level declaration `val` *valbind*; e.g. `val x::l =` exp_1 `and y =` exp_2 is not faulted by the compiler at top level, but may of course generate a `Bind` exception.

6.6 Closures

The informal understanding of a *closure* $(match, E, VE)$ is as follows: when the 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 $\mathrm{Dom}\, VE$ of this third component contains those function 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 $E + \mathrm{Rec}\, VE$, where

$$\mathrm{Rec} \ : \ \mathrm{VarEnv} \to \mathrm{VarEnv$$

is defined as follows:

- $\mathrm{Dom}(\mathrm{Rec}\, VE) = \mathrm{Dom}\, VE$
- If $VE(var) \notin$ Closure, then $(\mathrm{Rec}\, VE)(var) = VE(var)$
- If $VE(var) = (match', E', VE')$ then $(\mathrm{Rec}\, VE)(var) = (match', E', VE)$

The effect is that, before application of $(match, E, VE)$ to v, the closure values in $\mathrm{Ran}\, VE$ 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 closure.

6.7 Inference Rules

The semantic rules allow sentences of the form

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

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

$$\frac{A_1 \vdash phrase_1 \Rightarrow A_1' \qquad A_2 \vdash phrase_2 \Rightarrow A_2' \quad \cdots \quad \cdots \quad A_n \vdash phrase_n \Rightarrow A_n'}{A \vdash phrase \Rightarrow A'}$$

then the full form is intended to be

$$\frac{s_0, A_1 \vdash phrase_1 \Rightarrow A_1', s_1 \qquad s_1, A_2 \vdash phrase_2 \Rightarrow A_2', s_2 \quad \cdots \quad \cdots \quad s_{n-1}, A_n \vdash phrase_n \Rightarrow A_n', s_n}{s_0, A \vdash phrase \Rightarrow A', s_n}$$

(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

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

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

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

where p' does not occur in the original rule.[2] 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 $\mathrm{Val} \cup \mathrm{Pack}$. We also allow x/FAIL to range over $X \cup \{\mathrm{FAIL}\}$ where x ranges over X; furthermore, we extend environment modification to allow for failure as follows:

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

Atomic Expressions

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

$$\frac{}{E \vdash \mathit{scon} \Rightarrow \mathrm{val}(\mathit{scon})} \tag{103}$$

$$\frac{E(\mathit{longvar}) = v}{E \vdash \mathit{longvar} \Rightarrow v} \tag{104}$$

[2] There is one exception to the exception convention; no extra rule is added for rule 119 which deals with handlers, since a handler is the only means by which propagation of an exception can be arrested.

$$\frac{longcon = strid_1.\cdots.strid_k.con}{E \vdash longcon \Rightarrow con} \tag{105}$$

$$\frac{E(longexcon) = en}{E \vdash longexcon \Rightarrow en} \tag{106}$$

$$\frac{\langle E \vdash exprow \Rightarrow r \rangle}{E \vdash \texttt{\{}\ \langle exprow \rangle\ \texttt{\}} \Rightarrow \{\}\langle + r \rangle \text{ in Val}} \tag{107}$$

$$\frac{E \vdash dec \Rightarrow E' \qquad E + E' \vdash exp \Rightarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow v} \tag{108}$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow v} \tag{109}$$

Comments:

(105) Value constructors denote themselves.

(106) Exception constructors are looked up in the exception environment component of E.

Expression Rows

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

$$\frac{E \vdash exp \Rightarrow v \qquad \langle E \vdash exprow \Rightarrow r \rangle}{E \vdash lab\ \texttt{=}\ exp\ \langle\ \texttt{,}\ exprow \rangle \Rightarrow \{lab \mapsto v\}\langle + r \rangle} \tag{110}$$

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

Expressions

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

$$\frac{E \vdash atexp \Rightarrow v}{E \vdash atexp \Rightarrow v} \tag{111}$$

$$\frac{E \vdash exp \Rightarrow con \qquad con \neq \texttt{ref} \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (con, v)} \tag{112}$$

$$\frac{E \vdash exp \Rightarrow en \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (en, v)} \tag{113}$$

$$\frac{s, E \vdash exp \Rightarrow \texttt{ref}\ , s' \qquad s', E \vdash atexp \Rightarrow v, s'' \qquad a \notin \mathrm{Dom}(mem\,\mathrm{of}\ s'')}{s, E \vdash exp\ atexp \Rightarrow a,\ s'' + \{a \mapsto v\}} \tag{114}$$

$$\frac{s, E \vdash exp \Rightarrow\ \texttt{:=}\ , s' \qquad s', E \vdash atexp \Rightarrow \{\texttt{1} \mapsto a,\ \texttt{2} \mapsto v\}, s''}{s, E \vdash exp\ atexp \Rightarrow \{\}\ \mathrm{in\ Val},\ s'' + \{a \mapsto v\}} \tag{115}$$

$$\frac{E \vdash exp \Rightarrow b \qquad E \vdash atexp \Rightarrow v \qquad \mathrm{APPLY}(b, v) = v'}{E \vdash exp\ atexp \Rightarrow v'} \tag{116}$$

$$\frac{\begin{array}{c}E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\,VE,\ v \vdash match \Rightarrow v'\end{array}}{E \vdash exp\ atexp \Rightarrow v'} \tag{117}$$

$$\frac{\begin{array}{c}E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\,VE,\ v \vdash match \Rightarrow \mathrm{FAIL}\end{array}}{E \vdash exp\ atexp \Rightarrow [\texttt{Match}]} \tag{118}$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \tag{119}$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \tag{120}$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow \mathrm{FAIL}}{E \vdash exp\ \texttt{handle}\ match \Rightarrow [e]} \tag{121}$$

$$\frac{E \vdash exp \Rightarrow e}{E \vdash \texttt{raise}\ exp \Rightarrow [e]} \tag{122}$$

$$\frac{}{E \vdash \texttt{fn}\ match \Rightarrow (match, E, \{\})} \tag{123}$$

Comments:

(114) The side condition ensures that a new address is chosen. There are no rules concerning disposal of inaccessible addresses ("garbage collection").

(112)–(118) 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.

(119) This is the only rule to which the exception convention does not apply. If the operator evaluates to a packet then rule 120 or rule 121 must be used.

(121) Packets that are not handled by the *match* propagate.

(123) The third component of the closure is empty because the match does not introduce new recursively defined values.

Matches

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

$$\frac{E, v \vdash mrule \Rightarrow v'}{E, v \vdash mrule\ \langle\ \texttt{|}\ match\rangle \Rightarrow v'} \tag{124}$$

$$\frac{E, v \vdash mrule \Rightarrow \text{FAIL}}{E, v \vdash mrule \Rightarrow \text{FAIL}} \tag{125}$$

$$\frac{E, v \vdash mrule \Rightarrow \text{FAIL} \qquad E, v \vdash match \Rightarrow v'/\text{FAIL}}{E \vdash mrule\ \texttt{|}\ match \Rightarrow v'/\text{FAIL}} \tag{126}$$

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

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

$$\frac{E, v \vdash pat \Rightarrow VE \qquad E + VE \vdash exp \Rightarrow v'}{E, v \vdash pat\ \texttt{=>}\ exp\ \Rightarrow v'} \tag{127}$$

$$\frac{E, v \vdash pat \Rightarrow \text{FAIL}}{E, v \vdash pat\ \texttt{=>}\ exp\ \Rightarrow \text{FAIL}} \tag{128}$$

Declarations

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

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \texttt{val}\ valbind \Rightarrow VE \text{ in Env}} \tag{129}$$

$$\frac{E \vdash exbind \Rightarrow EE}{E \vdash \texttt{exception}\ exbind \Rightarrow EE \text{ in Env}} \tag{130}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash \texttt{local}\ dec_1\ \texttt{in}\ dec_2\ \texttt{end} \Rightarrow E_2} \tag{131}$$

$$\frac{E(longstrid_1) = E_1 \quad \cdots \quad E(longstrid_k) = E_k}{E \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_k} \tag{132}$$

$$\frac{}{E \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{133}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash dec_1\ \langle;\rangle\ dec_2 \Rightarrow E_1 + E_2} \tag{134}$$

Value Bindings

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

$$\frac{E \vdash exp \Rightarrow v \qquad E, v \vdash pat \Rightarrow VE \qquad \langle E \vdash valbind \Rightarrow VE' \rangle}{E \vdash pat\ \texttt{=}\ exp\ \langle \texttt{and}\ valbind \rangle \Rightarrow VE\ \langle + VE' \rangle} \tag{135}$$

$$\frac{E \vdash exp \Rightarrow v \qquad E, v \vdash pat \Rightarrow \text{FAIL}}{E \vdash pat\ \texttt{=}\ exp\ \langle \texttt{and}\ valbind \rangle \Rightarrow [\texttt{Bind}]} \tag{136}$$

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \texttt{rec}\ valbind \Rightarrow \text{Rec}\, VE} \tag{137}$$

Exception Bindings

$\boxed{E \vdash exbind \Rightarrow EE/p}$

$$\frac{en \notin ens \text{ of } s \qquad s' = s + \{en\} \qquad \langle s', E \vdash exbind \Rightarrow EE, s'' \rangle}{s, E \vdash excon\ \langle \texttt{and}\ exbind \rangle \Rightarrow \{excon \mapsto en\} \langle + EE \rangle,\ s' \langle ' \rangle} \tag{138}$$

$$\frac{E(longexcon) = en \qquad \langle E \vdash exbind \Rightarrow EE \rangle}{E \vdash excon\ \texttt{=}\ longexcon\ \langle \texttt{and}\ exbind \rangle \Rightarrow \{excon \mapsto en\} \langle + EE \rangle} \tag{139}$$

Comments:

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

Atomic Patterns

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

$$\frac{}{E, v \vdash \_ \Rightarrow \{\}} \tag{140}$$

$$\frac{v = \text{val}(scon)}{E, v \vdash scon \Rightarrow \{\}} \tag{141}$$

$$\frac{v \neq \text{val}(scon)}{E, v \vdash scon \Rightarrow \text{FAIL}} \tag{142}$$

$$\frac{}{E, v \vdash var \Rightarrow \{var \mapsto v\}} \tag{143}$$

$$\frac{longcon = strid_1.\cdots.strid_k.con \qquad v = con}{E, v \vdash longcon \Rightarrow \{\}} \tag{144}$$

$$\frac{longcon = strid_1.\cdots.strid_k.con \qquad v \neq con}{E, v \vdash longcon \Rightarrow \text{FAIL}} \tag{145}$$

$$\frac{E(longexcon) = v}{E, v \vdash longexcon \Rightarrow \{\}} \tag{146}$$

$$\frac{E(longexcon) \neq v}{E, v \vdash longexcon \Rightarrow \text{FAIL}} \tag{147}$$

$$\frac{v = \{\}\langle +r\rangle \text{ in Val} \qquad \langle E, r \vdash patrow \Rightarrow VE/\text{FAIL}\rangle}{E, v \vdash \texttt{\{}\ \langle patrow\rangle\ \texttt{\}} \Rightarrow \{\}\langle +VE/\text{FAIL}\rangle} \tag{148}$$

$$\frac{E, v \vdash pat \Rightarrow VE/\text{FAIL}}{E, v \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow VE/\text{FAIL}} \tag{149}$$

Comments:

(142),(145),(147) Any evaluation resulting in FAIL must do so because rule 142, rule 145, rule 147, rule 155, or rule 157 has been applied.

Pattern Rows

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

$$\frac{}{E, r \vdash \ldots \Rightarrow \{\}} \tag{150}$$

$$\frac{E, r(lab) \vdash pat \Rightarrow \text{FAIL}}{E, r \vdash lab \texttt{ = } pat \langle \texttt{ , } patrow \rangle \Rightarrow \text{FAIL}} \tag{151}$$

$$\frac{E, r(lab) \vdash pat \Rightarrow VE \qquad \langle E, r \vdash patrow \Rightarrow VE'/\text{FAIL} \rangle}{E, r \vdash lab \texttt{ = } pat \langle \texttt{ , } patrow \rangle \Rightarrow VE \langle + VE'/\text{FAIL} \rangle} \tag{152}$$

Comments:

(151),(152) For well-typed programs *lab* will be in the domain of r.

Patterns

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

$$\frac{E, v \vdash atpat \Rightarrow VE/\text{FAIL}}{E, v \vdash atpat \Rightarrow VE/\text{FAIL}} \tag{153}$$

$$\frac{longcon = strid_1.\cdots.strid_k.con \neq \texttt{ref} \qquad v = (con, v') \quad E, v' \vdash atpat \Rightarrow VE/\text{FAIL}}{E, v \vdash longcon\ atpat \Rightarrow VE/\text{FAIL}} \tag{154}$$

$$\frac{longcon = strid_1.\cdots.strid_k.con \neq \texttt{ref} \qquad v \notin \{con\} \times \text{Val}}{E, v \vdash longcon\ atpat \Rightarrow \text{FAIL}} \tag{155}$$

$$\frac{E(longexcon) = en \qquad v = (en, v') \quad E, v' \vdash atpat \Rightarrow VE/\text{FAIL}}{E, v \vdash longexcon\ atpat \Rightarrow VE/\text{FAIL}} \tag{156}$$

$$\frac{E(longexcon) = en \qquad v \notin \{en\} \times \text{Val}}{E, v \vdash longexcon\ atpat \Rightarrow \text{FAIL}} \tag{157}$$

$$\frac{s(a) = v \qquad s, E, v \vdash atpat \Rightarrow VE/\text{FAIL}, s}{s, E, a \vdash \texttt{ref}\ atpat \Rightarrow VE/\text{FAIL}, s} \tag{158}$$

$$\frac{E, v \vdash pat \Rightarrow VE/\text{FAIL}}{E, v \vdash var\ \texttt{as}\ pat \Rightarrow \{var \mapsto v\} + VE/\text{FAIL}} \tag{159}$$

Comments:

(155),(157) Any evaluation resulting in FAIL must do so because rule 142, rule 145, rule 147, rule 155, or rule 157 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. Unlike types, it cannot ignore them completely; the reason is that an explicit signature ascription plays the role of restricting the "view" of a structure - that is, restricting the domains of its component environments. However, the types and the sharing properties of structures and signatures are irrelevant to dynamic evaluation; 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 exception descriptions.
- Any specification of the form "`type` *typdesc*", "`eqtype` *typdesc*", "`datatype` *datdesc*" or "`sharing` *shareq*" is replaced by the empty specification.
- The Modules phrase classes TypDesc, DatDesc, ConDesc and SharEq are 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* $I \in \text{Int}$

$$
\begin{array}{rcl}
(\mathit{strid} : I, \mathit{strexp}\langle : I' \rangle, B) & \in & \text{FunctorClosure} \\
 & & \quad = (\text{StrId} \times \text{Int}) \times (\text{StrExp}\langle \times \text{Int} \rangle) \times \text{Basis} \\
(\mathit{IE}, \mathit{vars}, \mathit{excons}) \text{ or } I & \in & \text{Int} = \text{IntEnv} \times \text{Fin(Var)} \times \text{Fin(ExCon)} \\
\mathit{IE} & \in & \text{IntEnv} = \text{StrId} \xrightarrow{\text{fin}} \text{Int} \\
G & \in & \text{SigEnv} = \text{SigId} \xrightarrow{\text{fin}} \text{Int} \\
F & \in & \text{FunEnv} = \text{FunId} \xrightarrow{\text{fin}} \text{FunctorClosure} \\
(F, G, E) \text{ or } B & \in & \text{Basis} = \text{FunEnv} \times \text{SigEnv} \times \text{Env} \\
(G, \mathit{IE}) \text{ or } \mathit{IB} & \in & \text{IntBasis} = \text{SigEnv} \times \text{IntEnv}
\end{array}
$$

Figure 14: Compound Semantic Objects

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 "`open`") are represented only by their interfaces. To extract an interface from a dynamic environment we define the operation

$$\text{Inter} \ : \ \text{Env} \to \text{Int}$$

as follows:

$$\text{Inter}(SE, VE, EE) \ = \ (IE, \text{Dom}\, VE, \text{Dom}\, EE)$$

where

$$IE \ = \ \{strid \mapsto \text{Inter}\, E \ ; \ SE(strid) = E\} \ .$$

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

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

A further operation

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

is required, to cut down an environment E to a given interface I, representing the effect of an explicit signature ascription. It is defined as follows:

$$(SE, VE, EE) \downarrow (IE, vars, excons) \ = \ (SE', VE', EE')$$

where

$$SE' \ = \ \{strid \mapsto E \downarrow I \ ; \ SE(strid) = E \text{ and } IE(strid) = I\}$$

and (taking $\downarrow$ now to mean restriction of a function domain)

$$VE' = VE \downarrow vars, \ EE' = EE \downarrow excons.$$

It is important to note that an interface is also a projection of the *static* value Σ of a signature expression; it is obtained by omitting structure names m and type environments TE, and replacing each variable environment VE and each exception environment EE by its domain. 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

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

to be inferred, where A is either a basis or an interface basis 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 *strexp*, *strdec*, *strbind* or *topdec*.

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

$$\frac{B \vdash \mathit{strdec} \Rightarrow E}{B \vdash \texttt{struct}\ \mathit{strdec}\ \texttt{end} \Rightarrow E} \tag{160}$$

$$\frac{B(\mathit{longstrid}) = E}{B \vdash \mathit{longstrid} \Rightarrow E} \tag{161}$$

$$\frac{\begin{array}{c} B(\mathit{funid}) = (\mathit{strid} : I, \mathit{strexp}'\langle : I'\rangle, B') \\ B \vdash \mathit{strexp} \Rightarrow E \qquad B' + \{\mathit{strid} \mapsto E \downarrow I\} \vdash \mathit{strexp}' \Rightarrow E' \end{array}}{B \vdash \mathit{funid}\ (\ \mathit{strexp}\ )\ \Rightarrow E'\langle \downarrow I'\rangle} \tag{162}$$

$$\frac{B \vdash \mathit{strdec} \Rightarrow E \qquad B + E \vdash \mathit{strexp} \Rightarrow E'}{B \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow E'} \tag{163}$$

Comments:

(162) Before the evaluation of the functor body *strexp*′, the actual argument E is cut down by the formal parameter interface I, so that any opening of *strid* resulting from the evaluation of *strexp*′ will produce no more components than anticipated during the static elaboration.

Structure-level Declarations $\boxed{B \vdash \mathit{strdec} \Rightarrow E/p}$

$$\frac{E \text{ of } B \vdash \mathit{dec} \Rightarrow E'}{B \vdash \mathit{dec} \Rightarrow E'} \tag{164}$$

$$\frac{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}}{B \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow \mathit{SE} \text{ in Env}} \tag{165}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2} \tag{166}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{167}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \mathit{strdec}_1\ \langle ; \rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2} \tag{168}$$

Structure Bindings $\boxed{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}/p}$

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow E \qquad \langle \text{Inter}\, B \vdash \mathit{sigexp} \Rightarrow I \rangle \\ \langle\langle B \vdash \mathit{strbind} \Rightarrow \mathit{SE} \rangle\rangle \end{array}}{\begin{array}{l} B \vdash \mathit{strid}\ \langle \texttt{:}\ \mathit{sigexp} \rangle\ \texttt{=}\ \mathit{strexp}\ \langle\langle \texttt{and}\ \mathit{strbind} \rangle\rangle \Rightarrow \\ \qquad \{\mathit{strid} \mapsto E\langle\downarrow I\rangle\}\ \langle\langle + \mathit{SE} \rangle\rangle \end{array}} \tag{169}$$

Comment: As in the static semantics, when present, *sigexp* constrains the "view" of the structure. The restriction must be done in the dynamic semantics to ensure that any dynamic opening of the structure produces no more components than anticipated during the static elaboration.

Signature Expressions $\boxed{\mathit{IB} \vdash \mathit{sigexp} \Rightarrow I}$

$$\frac{\mathit{IB} \vdash \mathit{spec} \Rightarrow I}{\mathit{IB} \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow I} \tag{170}$$

$$\frac{\mathit{IB}(\mathit{sigid}) = I}{\mathit{IB} \vdash \mathit{sigid} \Rightarrow I} \tag{171}$$

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

$$\frac{IB \vdash sigbind \Rightarrow G}{IB \vdash \texttt{signature}\ sigbind \Rightarrow G} \tag{172}$$

$$\frac{}{IB \vdash \quad \Rightarrow \{\}} \tag{173}$$

$$\frac{IB \vdash sigdec_1 \Rightarrow G_1 \qquad IB + G_1 \vdash sigdec_2 \Rightarrow G_2}{IB \vdash sigdec_1\ \langle\texttt{;}\rangle\ sigdec_2 \Rightarrow G_1 + G_2} \tag{174}$$

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

$$\frac{IB \vdash sigexp \Rightarrow I \qquad \langle IB \vdash sigbind \Rightarrow G\rangle}{IB \vdash sigid\ \texttt{=}\ sigexp\ \langle\texttt{and}\ sigbind\rangle \Rightarrow \{sigid \mapsto I\}\ \langle + G\rangle} \tag{175}$$

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

$$\frac{\vdash valdesc \Rightarrow vars}{IB \vdash \texttt{val}\ valdesc \Rightarrow vars \text{ in Int}} \tag{176}$$

$$\frac{\vdash exdesc \Rightarrow excons}{IB \vdash \texttt{exception}\ exdesc \Rightarrow excons \text{ in Int}} \tag{177}$$

$$\frac{IB \vdash strdesc \Rightarrow IE}{IB \vdash \texttt{structure}\ strdesc \Rightarrow IE \text{ in Int}} \tag{178}$$

$$\frac{IB \vdash spec_1 \Rightarrow I_1 \qquad IB + IE \text{ of } I_1 \vdash spec_2 \Rightarrow I_2}{IB \vdash \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow I_2} \tag{179}$$

$$\frac{IB(longstrid_1) = I_1 \quad \cdots \quad IB(longstrid_n) = I_n}{IB \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow I_1 + \cdots + I_n} \tag{180}$$

$$\frac{IB(sigid_1) = I_1 \quad \cdots \quad IB(sigid_n) = I_n}{IB \vdash \texttt{include}\ sigid_1 \cdots sigid_n \Rightarrow I_1 + \cdots + I_n} \tag{181}$$

$$\frac{}{IB \vdash \quad \Rightarrow \{\} \text{ in Int}} \tag{182}$$

$$\frac{IB \vdash spec_1 \Rightarrow I_1 \qquad IB + IE \text{ of } I_1 \vdash spec_2 \Rightarrow I_2}{IB \vdash spec_1\ \langle\texttt{;}\rangle\ spec_2 \Rightarrow I_1 + I_2} \tag{183}$$

Comments:

(179),(183) Note that *vars* of I_1 and *excons* of I_1 are not needed for the evaluation of $spec_2$.

Value Descriptions

$\boxed{\vdash valdesc \Rightarrow vars}$

$$\frac{\langle \vdash valdesc \Rightarrow vars \rangle}{\vdash var\ \langle \texttt{and}\ valdesc \rangle \Rightarrow \{var\}\ \langle \cup\ vars \rangle} \tag{184}$$

Exception Descriptions

$\boxed{\vdash exdesc \Rightarrow excons}$

$$\frac{\langle \vdash exdesc \Rightarrow excons \rangle}{\vdash excon\ \langle exdesc \rangle \Rightarrow \{excon\}\ \langle \cup\ excons \rangle} \tag{185}$$

Structure Descriptions

$\boxed{IB \vdash strdesc \Rightarrow IE}$

$$\frac{IB \vdash sigexp \Rightarrow I \qquad \langle IB \vdash strdesc \Rightarrow IE \rangle}{IB \vdash strid\ :\ sigexp\ \langle \texttt{and}\ strdesc \rangle \Rightarrow \{strid \mapsto I\}\ \langle +\ IE \rangle} \tag{186}$$

Functor Bindings

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

$$\frac{\begin{array}{c}\mathrm{Inter}\, B \vdash sigexp \Rightarrow I \qquad \langle \mathrm{Inter}\, B + \{strid \mapsto I\} \vdash sigexp' \Rightarrow I' \rangle \\ \langle \langle B \vdash funbind \Rightarrow F \rangle \rangle\end{array}}{\begin{array}{r}B \vdash funid\ (\ strid\ :\ sigexp\ )\ \langle :\ sigexp' \rangle\ \texttt{=}\ strexp\ \langle \langle \texttt{and}\ funbind \rangle \rangle \Rightarrow \\ \{funid \mapsto (strid : I, strexp \langle : I' \rangle, B)\}\ \langle \langle +\ F \rangle \rangle\end{array}} \tag{187}$$

Functor Declarations

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

$$\frac{B \vdash funbind \Rightarrow F}{B \vdash \texttt{functor}\ funbind \Rightarrow F} \tag{188}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\}} \tag{189}$$

$$\frac{B \vdash fundec_1 \Rightarrow F_1 \qquad B + F_1 \vdash fundec_2 \Rightarrow F_2}{B \vdash fundec_1\ \langle ; \rangle\ fundec_2 \Rightarrow F_1 + F_2} \tag{190}$$

Top-level Declarations

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

$$\frac{B \vdash \mathit{strdec} \Rightarrow E}{B \vdash \mathit{strdec} \Rightarrow E \text{ in Basis}} \tag{191}$$

$$\frac{\text{Inter}\, B \vdash \mathit{sigdec} \Rightarrow G}{B \vdash \mathit{sigdec} \Rightarrow G \text{ in Basis}} \tag{192}$$

$$\frac{B \vdash \mathit{fundec} \Rightarrow F}{B \vdash \mathit{fundec} \Rightarrow F \text{ in Basis}} \tag{193}$$

8 Programs

The phrase class Program of programs is defined as follows

$$\mathit{program} ::= \mathit{topdec} \ ; \ \langle \mathit{program} \rangle$$

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 19. 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

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

Further, we shall use $\vdash_{\text{STAT}}$ for elaboration as defined in Section 5, and $\vdash_{\text{DYN}}$ 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

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

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 or external intervention) 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 $\ldots \vdash_{\text{STAT}} \mathit{topdec} \not\Rightarrow$. (This covers also the case in which a user interrupts the elaboration.)

Programs

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

$$\frac{B_{\text{STAT}} \text{ of } B \vdash_{\text{STAT}} \mathit{topdec} \not\Rightarrow \qquad \langle s, B \vdash \mathit{program} \Rightarrow B', s' \rangle}{s, B \vdash \mathit{topdec}\ ;\ \langle \mathit{program} \rangle \Rightarrow B\langle' \rangle, s\langle' \rangle} \tag{194}$$

$$\frac{\begin{array}{l} B_{\text{STAT}} \text{ of } B \vdash_{\text{STAT}} \mathit{topdec} \Rightarrow B^{(1)}_{\text{STAT}} \\ s, B_{\text{DYN}} \text{ of } B \vdash_{\text{DYN}} \mathit{topdec} \Rightarrow p, s' \qquad \langle s', B \vdash \mathit{program} \Rightarrow B', s'' \rangle \end{array}}{s, B \vdash \mathit{topdec}\ ;\ \langle \mathit{program} \rangle \Rightarrow B\langle' \rangle, s'\langle' \rangle} \tag{195}$$

$$\frac{\begin{array}{l} B_{\text{STAT}} \text{ of } B \vdash_{\text{STAT}} \mathit{topdec} \Rightarrow B^{(1)}_{\text{STAT}} \\ s, B_{\text{DYN}} \text{ of } B \vdash_{\text{DYN}} \mathit{topdec} \Rightarrow B^{(1)}_{\text{DYN}}, s' \quad B' = B \oplus (B^{(1)}_{\text{STAT}}, B^{(1)}_{\text{DYN}}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle s', B' \vdash \mathit{program} \Rightarrow B'', s'' \rangle \end{array}}{s, B \vdash \mathit{topdec}\ ;\ \langle \mathit{program} \rangle \Rightarrow B'\langle' \rangle, s'\langle' \rangle} \tag{196}$$

Comments:

(194) A failing elaboration has no effect whatever.

(195) 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

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

2. Replace the definition of structure-level declarations by

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

3. Omit the `open` declaration from the syntax class of declarations *dec*

4. Restrict the long identifier classes to identifiers, i.e. omit qualified identifiers.

This means that several components of a basis, for example the signature and functor environments, are irrelevant to the execution of a core language program.

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

$$\text{Derived form} \implies \text{Equivalent form}$$

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 $\overline{n}$ 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` *fvalbind*. The mixed forms `val rec` *fvalbind*, `val` *fvalbind* and `fun` *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 infixed; see Figure 20 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

  $$tyvarseq_1\ tycon_1\ \texttt{=}ty_1\ \ \texttt{and}\ \cdots\ \texttt{and}\ \ tyvarseq_n\ tycon_n\ \texttt{=}ty_n$$

  then *datbind*′ is the result of simultaneous replacement (in *datbind*) of every type expression $tyseq_i\ tycon_i$ $(1 \leq i \leq n)$ 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. These forms are currently more experimental than the bare syntax of modules, but we recommend implementers to include them so that they can be tested in practice. In the derived forms for functor bindings and functor signature expressions,

Derived Form	Equivalent Form	
Expressions *exp*		
`()`	`{ }`	
`(`exp_1 `,` $\cdots$ `,` exp_n`)`	`{1=`exp_1`,` $\cdots$`,` $\overline{n}$`=`exp_n`}`	$(n \geq 2)$
`#` *lab*	`fn {`*lab*`=`*var*`,...} =>` *var*	(*var* new)
`case` *exp* `of` *match*	`(fn` *match*`)(`*exp*`)`	
`if` exp_1 `then` exp_2 `else` exp_3	`case` exp_1 `of true =>` exp_2 `| false =>` exp_3	
exp_1 `orelse` exp_2	`if` exp_1 `then true else` exp_2	
exp_1 `andalso` exp_2	`if` exp_1 `then` exp_2 `else false`	
`(`exp_1 `;` $\cdots$ `;` exp_n `;` *exp*`)`	`case` exp_1 `of (_) =>` $\cdots$ `case` exp_n `of (_) =>` *exp*	$(n \geq 1)$
`let` *dec* `in` exp_1 `;` $\cdots$ `;` exp_n `end`	`let` *dec* `in (`exp_1 `;` $\cdots$ `;` exp_n`) end`	$(n \geq 2)$
`while` exp_1 `do` exp_2	`let val rec` *var* `= fn () => if` exp_1 `then (`exp_2`;`*var*`()) else () in` *var*`() end`	(*var* new)
`[`exp_1 `,` $\cdots$ `,` exp_n`]`	exp_1 `::` $\cdots$ `::` exp_n `:: nil`	$(n \geq 0)$

Figure 15: Derived forms of Expressions

strid is a new structure identifier and the form of *sigexp*′ depends on the form of *sigexp* as follows. If *sigexp* is simply a signature identifier *sigid*, then *sigexp*′ is also *sigid*; otherwise *sigexp* must take the form `sig` $spec_1$ `end` , and then *sigexp*′ is `sig local open` *strid* `in` $spec_1$ `end end`.

Derived Form	Equivalent Form	
Patterns *pat*		
`()`	`{ }`	
`(`pat_1 `,` $\cdots$ `,` pat_n`)`	`{1=`pat_1`,` $\cdots$ `,` $\overline{n}$`=`pat_n`}`	$(n \geq 2)$
`[`pat_1 `,` $\cdots$ `,` pat_n`]`	pat_1 `::` $\cdots$ `::` pat_n `::` `nil`	$(n \geq 0)$
Pattern Rows *patrow*		
$id\langle$`:`$ty\rangle$ $\langle$`as` $pat\rangle$ $\langle$`,` $patrow\rangle$	id `=` $id\langle$`:`$ty\rangle$ $\langle$`as` $pat\rangle$ $\langle$`,` $patrow\rangle$	
Type Expressions *ty*		
ty_1 `*` $\cdots$ `*` ty_n	`{1:`ty_1`,` $\cdots$ `,` $\overline{n}$`:`ty_n`}`	$(n \geq 2)$

Figure 16: Derived forms of Patterns and Type Expressions

Derived Form	Equivalent Form
Function-value Bindings *fvalbind*	
$\langle$`op`$\rangle var$ $atpat_{11}\cdots atpat_{1n}\langle$`:`$ty\rangle$ `=` exp_1 `|` $\langle$`op`$\rangle var$ $atpat_{21}\cdots atpat_{2n}\langle$`:`$ty\rangle$ `=` exp_2 `|` $\cdots$ $\cdots$ `|` $\langle$`op`$\rangle var$ $atpat_{m1}\cdots atpat_{mn}\langle$`:`$ty\rangle$ `=` exp_m $\langle$`and` $fvalbind\rangle$	$\langle$`op`$\rangle var$ `=` `fn` var_1`=>` $\cdots$ `fn` var_n`=>` `case (`var_1`,` $\cdots$ `,` var_n`) of` $(atpat_{11},\cdots,atpat_{1n})$`=>`$exp_1\langle$`:`$ty\rangle$ `|`$(atpat_{21},\cdots,atpat_{2n})$`=>`$exp_2\langle$`:`$ty\rangle$ `|` $\cdots$ $\cdots$ `|`$(atpat_{m1},\cdots,atpat_{mn})$`=>`$exp_m\langle$`:`$ty\rangle$ $\langle$`and` $fvalbind\rangle$
$(m, n \geq 1;\ var_1, \cdots, var_n$ distinct and new)	
Declarations *dec*	
`fun` *fvalbind*	`val rec` *fvalbind*
`datatype` *datbind* `withtype` *typbind*	`datatype` *datbind′* `;` `type` *typbind*
`abstype` *datbind* `withtype` *typbind* `with` *dec* `end`	`abstype` *datbind′* `with type` *typbind* `;` *dec* `end`
(see note in text concerning *datbind′*)	

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

Derived Form	Equivalent Form

Structure Expressions *strexp*

funid `(` *strdec* `)`	*funid* `(` `struct` *strdec* `end` `)`

Functor Bindings *funbind*

funid `(` *spec* `)` ⟨`:` *sigexp*⟩ `=` *strexp* ⟨`and` *funbind*⟩	*funid* `(` *strid* `:` `sig` *spec* `end` `)` ⟨`:` *sigexp′*⟩ `=` `let` `open` *strid* `in` *strexp* `end` ⟨`and` *funbind*⟩

(*strid* new; see note in text concerning *sigexp′*)

Functor Signature Expressions *funsigexp*

`(` *spec* `)` `:` *sigexp*	`(` *strid* `:` `sig` *spec* `end` `)` `:` *sigexp′*

(*strid* new; see note in text concerning *sigexp′*)

Top-level Declarations *topdec*

exp	`val` `it` `=` *exp*

Figure 18: Derived forms of Functors and Top-level Declarations

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 Figure 18 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` $\cdots$ `do` $\cdots$ ", are now disallowed.

The grammatical rules are displayed in Figures 19, 20, 21 and 22. The grammatical conventions are exactly as in Section 2, namely:

- 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:

xseq	::=	x	(singleton sequence)
			(empty sequence)
		`(`x_1`,`$\cdots$`,`x_n`)`	(sequence, $n \geq 1$)

  (Note that the "$\cdots$" 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

  F_1: `if` *exp*$_1$ `then` *exp*$_2$ `else` *exp*$_3$
  F_2: *exp* `handle` *match*

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

$$\cdots\ \cdots\ \texttt{if}\ \cdots\ \texttt{then}\ \cdots\ \texttt{else}\ \textit{exp}\ \texttt{handle}\ \cdots\ \cdots$$

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

$$\cdots\ \cdots\ \texttt{if}\ \cdots\ \texttt{then}\ \cdots\ \texttt{else}\ (\textit{exp}\ \texttt{handle}\ \cdots)\ \cdots$$

not the form

$$\cdots\ (\cdots\ \texttt{if}\ \cdots\ \texttt{then}\ \cdots\ \texttt{else}\ \textit{exp})\ \texttt{handle}\ \cdots\ \cdots$$

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

$$\texttt{if}\ \cdots\ \texttt{then}\ \texttt{while}\ \cdots\ \texttt{do}\ \cdots\ \texttt{else}\ \texttt{while}\ \cdots\ \texttt{do}\ \cdots$$

is quite admissible, and will be parsed as

$$\texttt{if}\ \cdots\ \texttt{then}\ (\texttt{while}\ \cdots\ \texttt{do}\ \cdots)\ \texttt{else}\ (\texttt{while}\ \cdots\ \texttt{do}\ \cdots)$$

- 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*, $\cdots$) 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.

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvar*	value variable
		⟨`op`⟩*longcon*	value constructor
		⟨`op`⟩*longexcon*	exception constructor
		`{` ⟨*exprow*⟩ `}`	record
		`#` *lab*	record selector
		`()`	0-tuple
		(*exp*$_1$ `,` ⋯ `,` *exp*$_n$)	n-tuple, $n \geq 2$
		`[`*exp*$_1$ `,` ⋯ `,` *exp*$_n$`]`	list, $n \geq 0$
		(*exp*$_1$ `;` ⋯ `;` *exp*$_n$)	sequence, $n \geq 2$
		`let` *dec* `in` *exp*$_1$ `;` ⋯ `;` *exp*$_n$ `end`	local declaration, $n \geq 1$
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
appexp	::=	*atexp*	
		appexp atexp	application expression
infexp	::=	*appexp*	
		infexp$_1$ *id infexp*$_2$	infix expression
exp	::=	*infexp*	
		exp `:` *ty*	typed (L)
		exp$_1$ `andalso` *exp*$_2$	conjunction
		exp$_1$ `orelse` *exp*$_2$	disjunction
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`if` *exp*$_1$ `then` *exp*$_2$ `else` *exp*$_3$	conditional
		`while` *exp*$_1$ `do` *exp*$_2$	iteration
		`case` *exp* `of` *match*	case analysis
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	

Figure 19: Grammar: Expressions and Matches

dec	::=	`val` *valbind*	value declaration
		`fun` *fvalbind*	function declaration
		`type` *typbind*	type declaration
		`datatype` *datbind* ⟨`withtype` *typbind*⟩	datatype declaration
		`abstype` *datbind* ⟨`withtype` *typbind*⟩ `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \cdots longstrid_n$	open declaration, $n \geq 1$
			empty declaration
		dec_1 ⟨;⟩ dec_2	sequential declaration
		`infix` ⟨d⟩ $id_1 \cdots id_n$	infix (L) directive, $n \geq 1$
		`infixr` ⟨d⟩ $id_1 \cdots id_n$	infix (R) directive, $n \geq 1$
		`nonfix` $id_1 \cdots id_n$	nonfix directive, $n \geq 1$
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
fvalbind	::=	⟨`op`⟩*var* $atpat_{11} \cdots atpat_{1n}$⟨`:` *ty*⟩`=`$exp_1$	$m, n \geq 1$
		`|` ⟨`op`⟩*var* $atpat_{21} \cdots atpat_{2n}$⟨`:` *ty*⟩`=`$exp_2$	See also note below
		`|` ⋯ ⋯	
		`|` ⟨`op`⟩*var* $atpat_{m1} \cdots atpat_{mn}$⟨`:` *ty*⟩`=`$exp_m$ ⟨`and` *fvalbind*⟩	
typbind	::=	*tyvarseq tycon* `=` *ty* ⟨`and` *typbind*⟩	
datbind	::=	*tyvarseq tycon* `=` *conbind* ⟨`and` *datbind*⟩	
conbind	::=	⟨`op`⟩*con* ⟨`of` *ty*⟩ ⟨ `|` *conbind*⟩	
exbind	::=	⟨`op`⟩*excon* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		⟨`op`⟩*excon* `=` ⟨`op`⟩*longexcon* ⟨`and` *exbind*⟩	

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

Figure 20: Grammar: Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		⟨`op`⟩*var*	variable
		⟨`op`⟩*longcon*	constant
		⟨`op`⟩*longexcon*	exception constant
		`{` ⟨*patrow*⟩ `}`	record
		`()`	0-tuple
		(*pat*$_1$ `,` $\cdots$ `,` *pat*$_n$)	n-tuple, $n \geq 2$
		[*pat*$_1$ `,` $\cdots$ `,` *pat*$_n$]	list, $n \geq 0$
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow*⟩	pattern row
		id⟨`:` *ty*⟩ ⟨`as` *pat*⟩ ⟨`,` *patrow*⟩	label as variable
pat	::=	*atpat*	atomic
		⟨`op`⟩*longcon atpat*	value construction
		⟨`op`⟩*longexcon atpat*	exception construction
		pat$_1$ *con pat*$_2$	infixed value construction
		pat$_1$ *excon pat*$_2$	infixed exception construction
		pat `:` *ty*	typed
		⟨`op`⟩*var*⟨`:` *ty*⟩ `as` *pat*	layered

Figure 21: Grammar: Patterns

ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record type expression
		tyseq longtycon	type construction
		ty$_1$ `*` $\cdots$ `*` *ty*$_n$	tuple type, $n \geq 2$
		ty `->` *ty′*	function type expression
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow*⟩	type-expression row

Figure 22: Grammar: Type expressions

C Appendix: The Initial Static Basis

We shall indicate components of the initial basis by the subscript 0. The initial static basis is

$$B_0 \;=\; (M_0, T_0), F_0, G_0, E_0$$

where

- $M_0 \;=\; \emptyset$
- $T_0 \;=\; \{\texttt{bool}, \texttt{int}, \texttt{real}, \texttt{string}, \texttt{list}, \texttt{ref}, \texttt{exn}, \texttt{instream}, \texttt{outstream}\}$
- $F_0 \;=\; \{\}$
- $G_0 \;=\; \{\}$
- $E_0 \;=\; (SE_0, TE_0, VE_0, EE_0)$

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`, `instream` and `outstream` admit equality.

The components of E_0 are as follows:

- $SE_0 \;=\; \{\}$
- VE_0 is shown in Figures 23 and 24. Note that $\mathrm{Dom}\, VE_0$ contains those identifiers (`true`,`false`,`nil`, `::`) which are basic value constructors, for reasons discussed in Section 4.3. VE_0 also includes EE_0, for the same reasons.
- TE_0 is shown in Figure 25. Note that the type structures in TE_0 contain the type schemes of all basic value constructors.
- $\mathrm{Dom}\, EE_0 \;=\;$ BasExName , the set of basic exception names listed in Section 6.5. In each case the associated type is `exn` , except that $EE_0(\texttt{Io}) = \texttt{string} \to \texttt{exn}$.

NONFIX			INFIX		
var	$\mapsto$	σ	*var*	$\mapsto$	σ
map	$\mapsto$	$\forall$'a 'b. ('a $\to$ 'b) $\to$ 'a list $\to$ 'b list	Precedence 7 :		
rev	$\mapsto$	$\forall$'a. 'a list $\to$ 'a list	/	$\mapsto$	real $*$ real $\to$ real
not	$\mapsto$	bool $\to$ bool	div	$\mapsto$	int $*$ int $\to$ int
~	$\mapsto$	num $\to$ num	mod	$\mapsto$	int $*$ int $\to$ int
abs	$\mapsto$	num $\to$ num	*	$\mapsto$	num $*$ num $\to$ num
floor	$\mapsto$	real $\to$ int	Precedence 6 :		
real	$\mapsto$	int $\to$ real	+	$\mapsto$	num $*$ num $\to$ num
sqrt	$\mapsto$	real $\to$ real	-	$\mapsto$	num $*$ num $\to$ num
sin	$\mapsto$	real $\to$ real	^	$\mapsto$	string $*$ string $\to$ string
cos	$\mapsto$	real $\to$ real	Precedence 5 :		
arctan	$\mapsto$	real $\to$ real	::	$\mapsto$	$\forall$'a.'a $*$ 'a list $\to$ 'a list
exp	$\mapsto$	real $\to$ real	@	$\mapsto$	$\forall$'a. 'a list $*$ 'a list $\to$ 'a list
ln	$\mapsto$	real $\to$ real	Precedence 4 :		
size	$\mapsto$	string $\to$ int	=	$\mapsto$	$\forall$''a. ''a $*$ ''a $\to$ bool
chr	$\mapsto$	int $\to$ string	<>	$\mapsto$	$\forall$''a. ''a $*$ ''a $\to$ bool
ord	$\mapsto$	string $\to$ int	<	$\mapsto$	num $*$ num $\to$ bool
explode	$\mapsto$	string $\to$ string list	>	$\mapsto$	num $*$ num $\to$ bool
implode	$\mapsto$	string list $\to$ string	<=	$\mapsto$	num $*$ num $\to$ bool
!	$\mapsto$	$\forall$'a. 'a ref $\to$ 'a	>=	$\mapsto$	num $*$ num $\to$ bool
ref	$\mapsto$	$\forall$ '_a . '_a $\to$ '_a ref	Precedence 3 :		
true	$\mapsto$	bool	:=	$\mapsto$	$\forall$'a. 'a ref $*$ 'a $\to$ unit
false	$\mapsto$	bool	o	$\mapsto$	$\forall$'a 'b 'c. ('b $\to$ 'c) $*$ ('a $\to$ 'b) $\to$ ('a $\to$ 'c)
nil	$\mapsto$	$\forall$'a. 'a list			

Notes:

- In type schemes we have taken the liberty of writing $ty_1 * ty_2$ in place of $\{1 \mapsto ty_1, 2 \mapsto ty_2\}$.
- An identifier with type involving num stands for two functions – one in which num is replaced by int in its type, and another in which num is replaced by real in its type. Sometimes an explicit type constraint will be needed if the surrounding text does not determine the type of a particular occurrence of + (for example). For this purpose, the surrounding text is no larger than the enclosing top-level declaration; an implementation may require that a smaller context determines the type.

Figure 23: Static VE_0 (except for Input/Output and EE_0)

var	$\mapsto$	σ
std_in	$\mapsto$	instream
open_in	$\mapsto$	string $\to$ instream
input	$\mapsto$	instream $*$ int $\to$ string
lookahead	$\mapsto$	instream $\to$ string
close_in	$\mapsto$	instream $\to$ unit
end_of_stream	$\mapsto$	instream $\to$ bool
std_out	$\mapsto$	outstream
open_out	$\mapsto$	string $\to$ outstream
output	$\mapsto$	outstream $*$ string $\to$ unit
close_out	$\mapsto$	outstream $\to$ unit

Figure 24: Static VE_0 (Input/Output)

tycon	$\mapsto$	$\{\ \theta,$	$\{con_1 \mapsto \sigma_1, \ldots, con_n \mapsto \sigma_n\}\ \}$ $(n \geq 0)$
unit	$\mapsto$	$\{\ \Lambda().\{\},$	$\{\}\ \}$
bool	$\mapsto$	$\{\ $bool,	$\{$true $\mapsto$ bool, false $\mapsto$ bool$\}\ \}$
int	$\mapsto$	$\{\ $int,	$\{\}\ \}$
real	$\mapsto$	$\{\ $real,	$\{\}\ \}$
string	$\mapsto$	$\{\ $string,	$\{\}\ \}$
list	$\mapsto$	$\{\ $list,	$\{$nil $\mapsto \forall$'a . 'a list, +::+ $\mapsto \forall$'a . 'a $*$ 'a list $\to$ 'a list$\}\ \}$
ref	$\mapsto$	$\{\ $ref,	$\{$ref $\mapsto \forall$ '_a . '_a $\to$ '_a ref$\}\ \}$
exn	$\mapsto$	$\{\ $exn,	$\{\}\ \}$
instream	$\mapsto$	$\{\ $instream,	$\{\}\ \}$
outstream	$\mapsto$	$\{\ $outstream,	$\{\}\ \}$

Figure 25: Static TE_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

- $F_0 \;=\; \{\}$
- $G_0 \;=\; \{\}$
- $E_0 \;=\; E'_0 + E''_0$

E'_0 contains bindings of identifiers to the basic values BasVal and basic exception names BasExName; in fact $E'_0 \;=\; SE'_0, VE'_0, EE'_0$, where:

- $SE'_0 \;=\; \{\}$
- $VE'_0 \;=\; \{id \mapsto id \;;\; id \in \text{BasVal}\} \cup \{\texttt{:=} \mapsto \texttt{:=}\}$
- $EE'_0 \;=\; \{id \mapsto id \;;\; id \in \text{BasExName}\}$

Note that VE'_0 is the identity function on BasVal; this is because we have chosen to denote these values by the names of variables to which they are initially bound. The semantics of these basic values (most of which are functions) lies principally in their behaviour under APPLY, which we describe below. On the other hand the semantics of `:=` is provided by a special semantic rule, rule 115. Similarly, EE'_0 is the identity function on BasExName, the set of basic exception names, because we have also chosen these names to be just those exception constructors to which they are initially bound. These exceptions are raised by APPLY as described below.

E''_0 contains initial variable bindings which, unlike BasVal, are definable in ML; it is the result of evaluating the following declaration in the basis F_0, G_0, E'_0. For convenience, we have also included all basic infix directives in this declaration.

```
infix  3  o
infix  4  = <> < > <= >=
infix  5  @
infixr 5  ::
infix  6  + - ^
```

```
infix  7  div mod / *

fun (F o G)x = F(G x)

fun nil @ M = M
  | (x::L) @ M = x::(L @ M)

fun s ^ s' = implode((explode s) @ (explode s'))

fun map F nil = nil
  | map F (x::L) = (F x)::(map F L)

fun rev nil = nil
  | rev (x::L) = (rev L) @ [x]

fun not true = false
  | not false = true

fun ! (ref x) = x
```

We now describe the effect of APPLY upon each value $b \in$ BasVal. For special values, we shall normally use i, r, n, s to range over integers, reals, numbers (integer or real), strings respectively. We also take the liberty of abbreviating "APPLY(`abs`, r)" to "`abs`(r)", "APPLY(`mod`, $\{1 \mapsto i, 2 \mapsto d\}$)" to "$i$ `mod` d", etc. .

- `~`(n) returns the negation of n, or the packet [`Neg`] if the result is out of range.
- `abs`(n) returns the absolute value of n, or the packet [`Abs`] if the result is out of range.
- `floor`(r) returns the largest integer i not greater than r; it returns the packet [`Floor`] if i is out of range.
- `real`(i) returns the real value equal to i.
- `sqrt`(r) returns the square root of r, or the packet [`Sqrt`] if r is negative.

- `sin`(r) , `cos`(r) return the result of the appropriate trigonometric functions.

- `arctan`(r) returns the result of the appropriate trigonometric function in the range $\pm\pi/2$.

- `exp`(r) , `ln`(r) return respectively the exponential and the natural logarithm of r, or an exception packet [`Exp`] or [`Ln`] if the result is out of range.

- `size`(s) returns the number of characters in s.

- `chr`(i) returns the character numbered i (see Section 2.2) if i is in the interval $[0, 255]$, and the packet [`Chr`] otherwise.

- `ord`(s) returns the number of the first character in s (an integer in the interval $[0, 255]$, see Section 2.2), or the packet [`Ord`] if s is empty.

- `explode`(s) returns the list of characters (as single-character strings) of which s consists.

- `implode`(L) returns the string formed by concatenating all members of the list L of strings.

- The arithmetic functions `/`,`*`,`+`,`-` all return the results of the usual arithmetic operations, or exception packets respectively [`Quot`], [`Prod`], [`Sum`], [`Diff`] if the result is undefined or out of range.

- i `mod` d , i `div` d return integers r, q (remainder, quotient) determined by the equation $d \times q + r = i$, where either $0 \leq r < d$ or $d < r \leq 0$. Thus the remainder has the same sign as the divisor d. The packet [`Mod`] or [`Div`] is returned if $d = 0$.

- The order relations `<`,`>`,`<=`,`>=` return boolean values in accord with their usual meanings.

- v_1 `=` v_2 returns `true` or `false` according as the values v_1 and v_2 are, or are not, identical. The type discipline (in particular, the fact that function types do not admit equality) ensures that equality is only ever applied to special values, nullary constructors, addresses, and values built out of such by record formation and constructor application.

- v_1 `<>` v_2 returns the opposite boolean value to v_1 `=` v_2.

It remains to define the effect of APPLY upon basic values concerned with input/output; we therefore proceed to describe the ML input/output system.

Input/Output in ML uses the concept of a *stream*. A stream is a finite or infinite sequence of characters; if finite, it may or may not be terminated. (It may be convenient to think of a special end-of-stream character signifying termination, provided one realises that this "character" is never treated as data). Input streams – or *instreams* – are of type `instream` and will be denoted by is ; output streams – or *outstreams* – are of type `outstream` and will be denoted by os . Both these types of stream are *abstract*, in the sense that streams may only be manipulated by the functions provided in BasVal.

Associated with an instream is a *producer*, normally an I/O device or file; similarly an outstream is associated with a *consumer*. After this association has been established – either initially or by the `open_in` or `open_out` function – the stream acts as a vehicle for character transmission from producer to program, or from program to consumer. The association can be broken by the `close_in` or `close_out` function. A closed stream permits no further character transmission; a closed instream is equivalent to one which is empty and terminated.

There are two streams in BasVal:

- `std_in`: an instream whose producer is the terminal.
- `std_out`: an outstream whose consumer is the terminal.

The other basic values concerned with Input/Output are all functional, and the effect of APPLY upon each of them given below. We take the liberty of abbreviating "APPLY(`open_in`, s)" to "`open_in`(s)" etc., and we shall use s and n to range over strings and integers respectively.

- `open_in`(s) returns a new instream is , whose producer is the external file named s . It returns exception packet

  [(`Io`,"`Cannot open` s")]

  if file s does not exist or does not provide read access.

- `open_out`(s) returns a new outstream os , whose consumer is the external file named s . If file s is non-existent, it is taken to be initially empty.

- `input`(is, n) returns a string s containing the first n characters of is, also removing them from is. If only $k < n$ characters are available on is, then
  - If is is terminated after these k characters, the returned string s contains them alone, and they are removed from is.
  - Otherwise no result is returned until the producer of is either supplies n characters or terminates the stream.
- `lookahead`(is) returns a single-character string s containing the next character of is, without removing it. If no character is available on is then
  - If is is closed, the empty string is returned.
  - Otherwise no result is returned until the producer of is either supplies a character or closes the stream.
- `close_in`(is) empties and terminates the instream is.
- `end_of_stream`(is) returns `true` if `lookahead`(is) returns the empty string, `false` otherwise; it detects the end of the instream is.
- `output`(os, s) writes the characters of s to the outstream os, unless os is closed, in which case it returns the exception packet

  ```
  [(Io,"Output stream is closed")]
  ```

- `close_out`(os) terminates the outstream os.

E 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 implementation environments, or on applications of the language.

Origins

ML and its semantic description have evolved over a period of about fourteen 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 [11]. 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 [12] 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 [24]. Other early influences were the applicative languages already in use in Artificial Intelligence, principally LISP [21], ISWIM [19] and POP2 [5].

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 [7] algorithm was by equation-solving; Hindley [14] used the unification algorithm of Alan Robinson [29] and also presented the precursor of our type inference system. James Morris [26] 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. I am grateful to Roger Hindley for pointing 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 [22].

Milner [23], 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 [9] presented the ML type inference systems following Hindley's style. Damas [8], using ideas from Michael Gordon, also devised the first mathematical treatment of polymorphism in the presence of references and assignment. Tofte [32] produced a different scheme, which has been adopted in the language.

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 [4] and in the functional programming language HOPE [3]; 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 [15] 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 [25]. The latest and final development has been in the exception mechanism, by MacQueen using an idea from Burstall [1]; it unites the ideas of exception and data type construction.

Modules

Besides contributory ideas to the Core Language, HOPE [3] 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 [4]. MacQueen, who was a member of Burstall's group at the time, designed [20] 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 [15].

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 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 [30]. Tofte, in his thesis [31], 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.

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 reported in [2]. 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 [27]. 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, arriving at a language and implementation known as CAML [6]; this is close to the core language of Standard ML, but does not include the Modules.

The first implementation of the Standard ML core language was by Mitchell, Mycroft and John Scott of Edinburgh, around 1984, and this was shortly followed by an implementation by David Matthews at Cambridge, carried out in his language Poly.

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 encountered. The Edinburgh implementation continues to play the role of a test-bed for language development.

Meanwhile Matthews' Cambridge implementation also advanced to embrace Modules. This implementation has supported applications of considerable size, both for machine-assisted proof and for hardware design.

In 1986, as the Modules definition was settling down, David MacQueen began an implementation at Bell AT&T Laboratories, joined later by Andrew Appel and Trevor Jim who are particularly interested in compilation into high quality machine code.

The Bell and Cambridge implementations, the former led by MacQueen and Appel, the latter by Matthews, are currently the most complete and highly engineered. 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 [12] 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 [28], 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 [10]. The static and dynamic semantics of the Core reached a final form mostly through work by Mads Tofte and Robin 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 [15] 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.

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 semantic method.

Literature

The present document is the definition of Standard ML; further versions of it will be produced as the language develops (but the intention is to minimise the number of versions). An informal definition, consistent with Version 2 of this document as far as the Core Language is concerned, is provided by [15], as modified by [25] and [1]. An elementary textbook covering the Core language has been recently published, written by rAke Wikström [33]. Robert Harper [13] has written a shorter introduction which also includes material

on Modules.

Further acknowledgments

Apart from the people mentioned above we also acknowledge the following, all of whom have contributed in some way to the evolution of ML: Guy Cousineau, Simon Finn, Jim Hook, Gerard Huet, Gilles Kahn, Brian Monahan, Peter Mosses, Alan Mycroft, David Park, David Rydeheard, David Schmidt, Stefan Sokolowski, Bernard Sufrin, Philip Wadler.

References

[1] Appel, A., MacQueen, D.B., Milner, R. and Tofte, M., *Unifying Exceptions with Constructors in Standard ML*, Report ECS-LFCS-88-55, Laboratory for Foundations of Computer Science, Computer Science Dept, Edinburgh University, 1988.

[2] Augustsson L. and Johnsson, T., *Lazy ML User's Manual*, Dept. of Computer Sciences, Chalmers University of Technology, Gothenburg, 1987.

[3] Burstall R.M., MacQueen, D.B. and Sannella, D.T., *HOPE: An Experimental Applicative Language*, Report CSR-62-80, Computer Science Dept, Edinburgh University, 1980.

[4] Burstall, R.M. and Goguen, J.A., *Putting Theories together to make Specifications*, Proc Fifth Annual Joint Conference on Artificial Intelligence, Cambridge, Mass., 1977, pp 1045–1058.

[5] Burstall, R.M. and Popplestone, R., *POP-2 Reference Manual*, Machine Intelligence 2, ed Dale and Michie, Oliver and Boyd, 1968.

[6] Cousineau, G., Curien, P.L. and Mauny, M., *The Categorical Abstract Machine*, in Functional Programming Languages and Computer Architecture, ed Jouannaud, Lecture Notes in Computer Science Vol 201, Springer Verlag, 1985, pp 50–64.

[7] Curry, H.B., *Modified Basic Functionality in Combinatory Logic*, Dialectica 23, 1969, pp 83–92.

[8] Damas, L., *Type Assignment in Programming Languages*, PhD thesis, CST-33-85, Computer Science Department, Edinburgh University, 1985.

[9] Damas, L. and Milner, R., *Principal Type-schemes for Functional Programs*, Proc 9th annual symposium on Principles of Programming Languages, ACM, 1982.

[10] Despeyroux, T., *Executable Specification of Static Semantics*, Proc Symposium on Semantics of Data Types, Sophia Antipolis, Springer-Verlag Lecture Notes in Computer Science, Vol.173, 1984.

[11] Gordon, M.J.C., Milner, R., Morris,L., Newey, M.C. and Wadsworth, C.P., *A Metalanguage for Interactive Proof in LCF*, Proc 5th ACM Symposium on Principles of Programming Languages, Tucson, 1978.

[12] Gordon, M.J.C., Milner, R. and Wadsworth, C.P., *Edinburgh LCF: a Machanised Logic of Computation*, Springer-Verlag Lecture Notes in Computer Science, Vol.78, 1979.

[13] Harper, R.W., *Introduction to Standard ML*, Report ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1986.

[14] Hindley, R., *The Principal Type-scheme of an Object in Combinatory Logic*, Transactions of AMS 146, pp29–60, 1969.

[15] Harper, R.M., MacQueen, D.B. and Milner, R., *Standard ML*, Report ECS-LFCS-86-2, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1986.

[16] Harper, R.M., Milner, R., Tofte, M., *The Semantics of Standard ML, Version 1* Report ECS-LFCS-87-36, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1987.

[17] Harper, R.M., Milner, R., Tofte, M., *The Definition of Standard ML, Version 2* Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1988.

[18] Harper, R.M., Milner, R., Tofte, M., *The Definition of Standard ML, Version 3* Report ECS-LFCS-89-81, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1989.

[19] Landin, P.J., *The next 700 Programming Languages*, CACM, Vol.9, No.3, 1966, pp57–164.

[20] MacQueen, D.D., *Structures and parameterisation in a typed functional language*, Proc. Symposium on Functional Programming and Computer Architecture, Aspinas, Sweden, 1981.

[21] McCarthy, J. et al., *LISP 1.5 Programming Manual*, The MIT Press, Cambridge, Mass, 1956.

[22] Meredith, D., *In memoriam Carew Arthur Meredith*, Notre Dame J. Formal Logic, Vol 18, 1977, pp 513–516.

[23] Milner, R., *A theory of type polymorphism in programming*, J. Comp. Sys.Sci, Vol 17, 1978, pp 348–375.

[24] Milner, R., *How ML Evolved*, Polymorphism (The ML/LCF/Hope Newsletter), Vol.1, No.1, 1983.

[25] Milner, R., *Changes to the Standard ML Core Language*, Report ECS-LFCS-87-33, Laboratory for Foundations of Computer Science, Computer Science Department, Edinburgh University, 1987.

[26] Morris, J.H., *Lambda Calculus Models of Programming Languages*, MAC-TR-57 (Thesis), Project MAC, M.I.T., 1968.

[27] Paulson, L.C., *Logic and Computation: Interactive Proof with LCF*, Cambridge Tracts in Theoretical Computer Science 2, Cambridge University Press, 1987.

[28] Plotkin, G.D., *A Structural Approach to Operational Semantics*, Technical Report DAIMI FN-19, Computer Science Department, rArhus University, 1981.

[29] Robinson, J.A., *A Machine-oriented Logic based upon the Resolution Principle*, Journal of ACM, Vol 12, No 1, pp23-41, 1965.

[30] Sannella, D.T. and Tarlecki, A., *Program Specification and Development in Standard ML*, Proc 12th ACM Symposium on Principles of Programming Languages, New Orleans, 1985.

[31] Tofte, M., *Operational Semantics and Polymorphic Type Inference*, PhD Thesis CST-52-88, Computer Science Department, Edinburgh University, 1988. (Also appears as Report ECS-LFCS-88-54 of the Laboratory for Foundations of Computer Science.)

[32] Tofte, M., *Type Inference for Polymorphic References* (To appear in Information and Computation)

[33] Wikström, rA., *Functional Programming using Standard ML*, Prentice Hall, 1987.