LFCS

Laboratory for Foundations of Computer Science

Department of Computer Science - University of Edinburgh

The Semantics of Standard ML
Version 1

by

Robert Harper Robin Milner
Mads Tofte

The Semantics of Standard ML
Version 1

ECS-LFCS-87-36
(also published as CSR-244-87)

LFCS Report Series

August 1987

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Preface to Version 1

Great care has been taken to make this document clear, accurate and complete. Despite this we have called it "Version 1", since we expect to amend it for various reasons.

First, neither the greatest clarity nor the greatest accuracy is possible in a document of this complexity without feedback from readers. We therefore encourage readers to send us suspected errors, and to indicate points which are not clear to them. Although we do not intend to turn this document a pedagogic exposition, we shall willingly add short illuminating comments.

Second, the design of ML Modules – particularly the grammar – is still somewhat experimental, even though it is considerably refined from its original form. As a result of experimental use it may be changed or extended, and these changes or extensions will be defined in later versions of the present document.

Third, though the ML Core Language is more stable – simply because it has been subjected to more experiment – changes here may also occur. Wherever possible they will be "upwards compatible" – that is, the validity and semantics of existing programs will be preserved. One change is at present under discussion, and (for reasons of human resource) we are not delaying the issue of this document to include it. The proposed change is to the *exception* facility; it will not only add power but will also simplify the language – in particular, it will unite the notions of *handler* and *match*. This simplification is so significant that it deserves consideration even though it slightly violates the principle of upwards compatible change. But if it is adopted it will be possible to automate the necessary small modifications to existing programs.

Version 1 treats the ML Core Language and its Input/Output facilities as defined in **Standard ML** by Robert Harper, David MacQueen and Robin Milner (Report ECS-LFCS-86-2, Edinburgh University, Computer Science Department), but incorporating the changes defined in **Changes to the Standard ML Core Language** by Robin Milner (Report ECS-LFCS-87-33). As explained above, the Modules part of the language described here is considerably refined from that presented by MacQueen in ECS-LFCS-86-2.

Any future Version of this document will indicate precisely how it differs from its predecessor.

The Semantics of Standard ML
Version 1

Robert Harper Robin Milner
Mads Tofte
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh
Edinburgh EH9 3JZ, Scotland

August 13, 1987

Contents

1 Introduction

This document provides a complete formal description of Standard ML.

To understand the method of description, 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 description 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 description, it eases presentation and understanding to keep the static and dynamic parts of the basis completely 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 description is possible, because of the structure of the language. ML consists of a lower level, called the *Core language* (or *Core*, for short) and an upper level concerned with programming-in-the-large, called *Modules*. The Core is a complete language in its own right, and its embedding in the full language is simple; therefore each of the three parts of the formal description is further divided into two – one for the Core, and one for Modules.

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. (There are no derived forms for Modules). Thus each of the three parts for the Core treats only the bare language; the derived forms are treated in Appendix A. A full grammar for the Core including derived forms is presented in Appendix B.

In Appendix 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

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

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: "in the basis A, the phrase *phrase* evaluates – or elaborates – 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. Some of these are stated but not proved; others are presented as "claims" rather than theorems – often they have been proved for a skeletal language, and although we are confident of their truth their proofs in the context of the full language will present an interesting challenge to a computer-assisted proof methodology, to attain complete certainty.

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. In this document the alphabetic reserved words are always shown in typewriter font.

```
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` .

A string constant is a sequence, between quotes (`"`), of zero or more printable characters, spaces or escape sequences. Each escape sequence is introduced by the escape character `\` , and stands for a character sequence. The allowed escape sequences are as follows (all other uses of `\` being incorrect):

`\n`	A single character interpreted by the system as end-of-line.
`\t`	Tab.
`\^`*c*	The control character *c*, for any appropriate *c*.
`\`*ddd*	The single character with ASCII code *ddd* (3 decimal digits).
`\"`	`"`
`\\`	`\`
`\`*f* ··*f*`\`	This sequence is ignored, where *f* ··*f* stands for a sequence of one or more formatting characters.

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

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

Figure 1: Identifiers

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 also a class

$$\mathrm{LongX} = \mathrm{StrId}^* \times \mathrm{X}$$

If x ranges over X, then *longx*, or $strid_1.\cdots.strid_k.x$, $k \geq 0$, ranges over LongX. These long identifiers constitute the only link between the Core and the language constructs for Modules; by omitting them, and the open declaration, we obtain the Core as a complete programming language in its own right. (The corresponding adjustment to the Core static and dynamic semantics is simply to omit structure environments *SE*.).

An identifier is either *alphanumeric*: any sequence of letters, digits, primes (') and underbars (_) starting with a letter or prime, or *symbolic*: any 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 other six classes (Var, Con, Exn, TyCon, Lab and StrId) are represented by identifiers not starting with a prime; the class Lab is also extended to include the *numeric* labels `1 2 3 ···`.

TyVar is therefore disjoint from the other six classes. Otherwise, the syntax class of an occurrence of identifier *id* in a Core phrase 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 name if it occurs immediately after `raise`, at the start of a handler rule *hrule*, or within an `exception` declaration or specification.

5. Elsewhere, *id* is 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 constructor by introducing the same identifier as a variable; this is because, in the scope of the declaration which introduces *id* as a value 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 parser can determine the class to which each identifier class 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 an 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* or a *con* within the scope (see below) of the directive. 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, non-infixed occurrences of *id* must be prefixed by the keyword `op` . 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 ($n \geq 1$):

`infix` $\langle d \rangle$ $id_1 \cdots id_n$

`infixr` $\langle d \rangle$ $id_1 \cdots id_n$

`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 effect 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.

AtExp	atomic expressions
ExpRow	expression rows
Exp	expressions
Match	matches
Mrule	match rules
Handler	handlers
Hrule	handler rules
Dec	declarations
ValBind	value bindings
TypBind	type bindings
DatBind	datatype bindings
Constrs	datatype constructions
ExnBind	exception bindings
AtPat	atomic patterns
PatRow	pattern rows
Pats	patterns
Ty	type expressions
TyRow	type expression rows

Figure 2: Core Phrase Classes

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

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

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

  (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.
- L (resp. R) means left (resp. right) association.

- The syntax of types binds more tightly that that of expressions.
- Each iterated construct (e.g. *match*, *handler*, $\cdots$) 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.

The grammatical rules for the Core are shown in Figures 3, 4, 5 and 6.

atexp	::=	⟨op⟩*longvar*	value variable
		⟨op⟩*longcon*	value 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` *handler*	handle exception
		`raise` *longexn* `with` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	
handler	::=	*hrule* ⟨ `||` *handler*⟩	
hrule	::=	*longexn* `with` *match*	
		`?` `=>` *exp*	

Figure 3: Grammar: Expressions, Matches and Handlers

dec	::=	`val` *valbind*	value declaration
		`type` *typbind*	type declaration
		`datatype` *datbind*	datatype declaration
		`abstype` *datbind* `with` *dec* `end`	abstype declaration
		`exception` *exnbind*	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* `=` *constrs* ⟨`and` *datbind*⟩	
constrs	::=	⟨`op`⟩*con* ⟨`of` *ty*⟩ ⟨ `|` *constrs*⟩	
exnbind	::=	*exn* ⟨`:` *ty*⟩⟨`=` *longexn*⟩ ⟨`and` *exnbind*⟩	

Figure 4: Grammar: Declarations and Bindings

atpat	::=	`_`	wildcard
		⟨`op`⟩*var*	variable
		longcon	constant
		`{` ⟨*patrow*⟩ `}`	record
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow*⟩	pattern row
pat		*atpat*	atomic
		⟨`op`⟩*longcon atpat*	construction
		pat_1 *con* pat_2	infixed construction
		pat `:` *ty*	typed
		⟨`op`⟩*var*⟨`:` *ty*⟩ `as` *pat*	layered

Figure 5: Grammar: Patterns

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 6: Grammar: 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 *exnbind* 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*.
- Every non-local exception binding – that is, not localised by `let` or `local` – must be explicitly constrained by a type containing no type variables.

3 Syntax of Modules

For Modules there are further keywords and identifier classes, but no further special constants and at present no further derived forms. Comments and lexical analysis are as for the Core.

3.1 Reserved Words

The following are the additional reserved words used in Modules.

`eqtype` `functor` `include` `open` `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` ⋯ `end`

`local` *strdec* `in` ⋯ `end`

`struct` *strdec* `end`

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

One effect of this limitation is that fixity is local to a 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; 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 7. We use the variable *strexp* to range over StrExp, etc. The conventions adopted in presenting the grammatical

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
ExnDesc	exception descriptions
StrDesc	structure descriptions
SharEq	sharing equations
FunDec	functor declarations
FunBind	functor bindings
FunSigExp	functor signature expressions
FunSpec	functor specifications
FunDesc	functor descriptions
Program	programs

Figure 7: Modules Phrase Classes

rules for Modules are the same as for the Core. The grammatical rules are shown in Figures 8, 9 and 10.

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

Figure 8: Grammar: Structures and Signatures

3.5 Syntactic Restrictions

- No binding *strbind*, *sigbind*, or *funbind* may bind the same identifier twice.
- No description *valdesc*, *typdesc*, *datdesc*, *exndesc*, *strdesc* or *fundesc* may describe the same identifier twice.

spec	::=	`val` *valdesc*	value
		`type` *typdesc*	type
		`eqtype` *typdesc*	eqtype
		`datatype` *datdesc*	datatype
		`exception` *exndesc*	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
		$spec_1$ ⟨`;`⟩ $spec_2$	sequential
valdesc	::=	*var* `:` *ty* ⟨and *valdesc*⟩	
typdesc	::=	*tyvarseq tycon* ⟨and *typdesc*⟩	
datdesc	::=	*tyvarseq tycon* `=` *constrs* ⟨and *datdesc*⟩	
exndesc	::=	*exn* `:` *ty* ⟨and *exndesc*⟩	
strdesc	::=	*strid* `:` *sigexp* ⟨and *strdesc*⟩	
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 9: Grammar: Specifications

fundec	::=	`functor` *funbind*	single
			empty
		$fundec_1$ ⟨`;`⟩ $fundec_2$	sequence
funbind	::=	*funid* `(` *spec* `)` ⟨`:` *sigexp*⟩ `=` *strexp* ⟨and *funbind*⟩	functor binding
funsigexp	::=	`(` *spec* `)` `:` *sigexp*	functor signature
funspec	::=	`functor` *fundesc*	functor specification
		$funspec_1$ ⟨`;`⟩ $funspec_2$	
fundesc	::=	*funid* *funsigexp* ⟨and *fundesc*⟩	
program	::=	*strdec*	structure-level declaration
		sigdec	signature declaration
		fundec	functor declaration
		$program_1$ ⟨`;`⟩ $program_2$	sequence

Figure 10: Grammar: Functors and Programs

3.6 Pure Functor Forms

The grammatical forms of functor bindings, functor signature expressions and functor applications which are treated in the formal semantics to follow, and which we shall call *pure* forms, differ slightly from those given in the foregoing grammatical rules, which we shall call *applied* forms. The pure forms are given in Figure 11.

strexp	::=	*funid* `(` *strexp* `)`	functor application
funbind	::=	*funid* `(` *strid* `:` *sigexp* `)` ⟨`:` *sigexp'*⟩ `=` *strexp* ⟨and *funbind*⟩	functor binding
funsigexp	::=	`(` *strid* `:` *sigexp* `)` `:` *sigexp'*	functor signature

Figure 11: Functor forms defined in the Semantics

These pure forms are more tractable in the semantic theory, since they treat functors as functions of a single structure argument. On the other hand the applied forms given in the grammar (Figure 10) are more suitable for programming, since they allow a functor to take merely a (named) type or value as argument. These applied forms are mandatory in programming. Their semantics in terms of the pure forms is given by translation, as follows. The applied form of functor application,

$$funid\ \texttt{(}\ strdec\ \texttt{)}$$

is translated to the pure form

$$funid\ \texttt{(}\ \texttt{struct}\ strdec\ \texttt{end}\ \texttt{)}$$

which "wraps up" the *strdec* as a structure. On the other hand the applied form of functor binding,

$$funid\ \texttt{(}\ spec\ \texttt{)}\ \langle\texttt{:}\ sigexp\rangle\ \texttt{=}\ strexp$$

is translated to the pure form

$$funid\ \texttt{(}\ \mathrm{X}\ \texttt{:sig}\ spec\ \texttt{end}\ \texttt{)}\ \langle\texttt{:}\ sigexp'\rangle\ \texttt{=}\ \texttt{let open}\ \mathrm{X}\ \texttt{in}\ strexp\ \texttt{end}$$

(where X is a structure identifier not previously occurring in the functor binding) which "wraps up" the *spec* as a signature, but compensates in the body to allow direct reference to members of *spec*. The form of *sigexp′* depends on the form of *sigexp*. 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

$$\texttt{sig local open}\ \mathrm{X}\ \texttt{in}\ spec_1\ \texttt{end end}$$

Finally, the applied form of functor signature expression,

$$\texttt{(}\ spec\ \texttt{)}\ \texttt{:}\ sigexp$$

is translated to the pure form

$$\texttt{(}\ \mathrm{X}\ \texttt{:}\ \texttt{sig}\ spec\ \texttt{end}\ \texttt{)}\ \texttt{:}\ sigexp'$$

where *sigexp′* is obtained from *sigexp* exactly as for functor binding above.

3.7 Closure Restrictions

The semantics presented in later sections requires no restriction on reference to non-local identifiers. For example, it allows a signature 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, in the present version of the language, apart from references to identifiers bound in the initial basis B_0 (which may occur anywhere), it is required that signatures only refer non-locally to signature identifiers and that functors only refer non-locally to functor and signature identifiers. These restrictions ease separate compilation; however, they may be relaxed in a future version of the language.

More precisely, the restrictions are as follows (ignoring reference to identifiers bound in B_0):

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

- In any functor description *funid* (*spec*) : *sigexp* , the only non-local references in *spec* and *sigexp* are to signature identifiers, except that *sigexp* may refer to identifiers specified in *spec*.

- In any functor binding *funid* (*spec*) ⟨: *sigexp*⟩ = *strexp* , the only non-local references in *spec*, *sigexp* and *strexp* are to functor and signature identifiers, except that both *sigexp* and *strexp* may refer to identifiers specified in *spec*.

In the last two cases the final qualification allows, for example, sharing constraints to be specified between functor argument and result.

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 12. We have included TyVar in the table to make visible the use of α in the semantics to range over TyVar.

$$\begin{array}{rcll} \alpha \text{ or } \mathit{tyvar} & \in & \mathrm{TyVar} & \text{type variables} \\ t & \in & \mathrm{TyName} & \text{type names} \\ m & \in & \mathrm{StrName} & \text{structure names} \end{array}$$

Figure 12: Simple Semantic Objects

Each $\alpha \in \mathrm{TyVar}$ possesses a boolean *equality* attribute, which determines whether or not it *admits equality* – in which case we also say that it is an *equality* type variable. Each $t \in \mathrm{TyName}$ has an arity $k \geq 0$, and also possesses an equality attribute. We denote the class of type names with arity k by $\mathrm{TyName}^{(k)}$.

4.2 Compound Objects

When A and B are sets $\mathrm{Fin}\, A$ denotes the set of finite subsets of A, and $A \xrightarrow{\mathrm{fin}} B$ 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 13.

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

Figure 13: Compound Semantic Objects

Note that Λ and $\forall$ bind type variables. For any semantic object A, $\mathrm{tynames}\, A$ and $\mathrm{tyvars}\, A$ denote respectively the set of type names and the set of 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 $C(\mathit{tycon})$ means $(TE \text{ of } C)\mathit{tycon}$.

A particular case needs mention: $C(\mathit{con})$ is taken to stand for $(VE \text{ of } C)\mathit{con}$. The type scheme of a value constructor is held in VE as well as in TE (where it will be recorded within a CE); thus the re-binding of a value constructor is given proper effect by accessing it in VE rather than TE.

Finally, environments may be applied to long identifiers. For instance if $\mathit{longcon} = \mathit{strid}_1.\cdots.\mathit{strid}_k.\mathit{con}$ then $E(\mathit{longcon})$ means

$$(VE \text{ of } (SE \text{ of } \cdots (SE \text{ of } (SE \text{ of } E)\mathit{strid}_1)\mathit{strid}_2\cdots)\mathit{strid}_k)\mathit{con}.$$

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

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

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 $C + (\mathrm{tynames}\, TE, TE)$.

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;
- $\{\mathit{lab}_1 \mapsto \tau_1,\ \cdots,\ \mathit{lab}_n \mapsto \tau_n\}$, where each τ_i admits equality;
- $\tau^{(k)}t$, where t and all members of $\tau^{(k)}$ admit equality;

- (τ')`ref`.

A type function $\theta = \Lambda\alpha^{(k)}.\tau$ has arity k; it must be *closed* – i.e. 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). 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

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

4.5 Type Schemes

A type scheme $\sigma = \forall\alpha^{(k)}.\tau$ *generalises* a type τ', written $\sigma \succ \tau'$, if $\tau' = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ for some $\tau^{(k)}$, where each member τ_i of $\tau^{(k)}$ admits equality if α_i does. If $\sigma' = \forall\beta^{(l)}.\tau'$ then σ *generalises* σ', written $\sigma \succ \sigma'$, if $\sigma \succ \tau'$ and $\beta^{(l)}$ contains no free type variable of σ. It can be shown that $\sigma \succ \sigma'$ iff, for all τ'', whenever $\sigma' \succ \tau''$ then also $\sigma \succ \tau''$.

Two type schemes σ and σ' are considered equal if they can be obtained from each other by renaming and reordering of bound type variables, and deleting type variables from the prefix which do not occur in the body. 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 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

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

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

4.7 Type Environments and Well-formedness

A type environment takes the form

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

and is *well-formed* if it satisfies the following conditions:

1. Either $CE_i = \{\}$, or θ_i has the form t_i and each $CE_i(con)$ has the form $\forall\alpha^{(k)}.(\tau \to \alpha^{(k)}t_i)$. The latter case occurs when $tycon_i$ is a datatype constructor; it is conveniently distinguished from an ordinary type constructor by possessing at least one value constructor.

2. If $tycon_i$ is a datatype constructor different from `ref`, so that $TE(tycon_i) = (t_i, CE_i)$ with $CE_i \neq \{\}$, then t_i admits equality only if, for each $CE_i(con) = \forall\alpha^{(k)}.(\tau \to \alpha^{(k)}t_i)$, the type function $\Lambda\alpha^{(k)}.\tau$ also admits equality. Furthermore, as many such t_i as possible admit equality, subject to the foregoing condition.

   This ensures that the equality predicate `=` will be applicable to a constructed value $con(v)$ of type $\tau^{(k)}t_i$ only when it is applicable to the value v itself, whose type is $\tau\{\tau^{(k)}/\alpha^{(k)}\}$.

3. Different datatype constructors are bound to different type names; i.e., if $i \neq j$ and $TE(tycon_i) = (t_i, CE_i)$ and $\mathrm{Dom}\,CE_i \neq \emptyset$ and $TE(tycon_j) = (t_j, CE_j)$ and $\mathrm{Dom}\,CE_j \neq \emptyset$ then $t_i \neq t_j$.

All type environments occurring in the rules are assumed well-formed.

For any TE as above, $\mathrm{Abs}\,TE$ is the type environment $\{tycon_1 \mapsto (\theta_1, \{\}), \cdots\}$ in which all constructor environments CE_i have been replaced by the empty map. The effect is to convert each $tycon_i$ into an ordinary type constructor.

4.8 Inference Rules

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

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

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 may be 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 $C = T, E$ has the property 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, 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 $T, E \vdash \textit{phrase} \Rightarrow A$ such that tynames $E \subseteq T$, and let S′ be a sentence $T', E' \vdash \textit{phrase}' \Rightarrow A'$ occurring in a proof of S; then also tynames $E' \subseteq T'$.

Atomic Expressions

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

$$\frac{C(\textit{longvar}) \succ \tau}{C \vdash \textit{longvar} \Rightarrow \tau} \qquad (1)$$

$$\frac{C(\textit{longcon}) \succ \tau}{C \vdash \textit{longcon} \Rightarrow \tau} \qquad (2)$$

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

$$\frac{C \vdash dec \Rightarrow E \qquad C \oplus E \vdash exp \Rightarrow \tau}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{4}$$

$$\frac{C \vdash exp \Rightarrow \tau}{C \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow \tau} \tag{5}$$

Comments:

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

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

Expression Rows

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

$$\frac{C \vdash exp \Rightarrow \tau \qquad \langle C \vdash exprow \Rightarrow \varrho \rangle}{C \vdash lab\ \texttt{=}\ exp\ \langle\ \texttt{,}\ exprow\rangle \Rightarrow \{lab \mapsto \tau\}\langle + \varrho\rangle} \tag{6}$$

Expressions

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

$$\frac{C \vdash atexp \Rightarrow \tau}{C \vdash atexp \Rightarrow \tau} \tag{7}$$

$$\frac{C \vdash exp \Rightarrow \tau' \to \tau \qquad C \vdash atexp \Rightarrow \tau'}{C \vdash exp\ atexp \Rightarrow \tau} \tag{8}$$

$$\frac{C \vdash exp \Rightarrow \tau \qquad C \vdash ty \Rightarrow \tau}{C \vdash exp\ \texttt{:}\ ty \Rightarrow \tau} \tag{9}$$

$$\frac{C \vdash exp \Rightarrow \tau \qquad C \vdash handler \Rightarrow \tau}{C \vdash exp\ \texttt{handle}\ handler \Rightarrow \tau} \tag{10}$$

$$\frac{C(longexn) = \tau \qquad C \vdash exp \Rightarrow \tau}{C \vdash \texttt{raise}\ longexn\ \texttt{with}\ exp \Rightarrow \tau'} \tag{11}$$

$$\frac{C \vdash match \Rightarrow \tau}{C \vdash \texttt{fn}\ match \Rightarrow \tau} \tag{12}$$

Comments:

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

(9) Here τ is determined by C and *ty*.

(11) Note that τ' does not occur in the premises; thus a **raise** expression has "arbitrary" type.

Matches

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

$$\frac{C \vdash mrule \Rightarrow \tau \qquad \langle C \vdash match \Rightarrow \tau \rangle}{C \vdash mrule \;\langle \mid match \rangle \Rightarrow \tau} \tag{13}$$

Match Rules

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

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C + VE \vdash exp \Rightarrow \tau'}{C \vdash pat \texttt{ => } exp \;\Rightarrow \tau \rightarrow \tau'} \tag{14}$$

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

Handlers

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

$$\frac{C \vdash hrule \Rightarrow \tau \qquad \langle C \vdash handler \Rightarrow \tau \rangle}{C \vdash hrule \;\langle \mid\mid handler \rangle \Rightarrow \tau} \tag{15}$$

Handle Rules

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

$$\frac{C(longexn) = \tau' \qquad C \vdash match \Rightarrow \tau' \rightarrow \tau}{C \vdash longexn \texttt{ with } match \Rightarrow \tau} \tag{16}$$

$$\frac{C \vdash exp \Rightarrow \tau}{C \vdash \texttt{?} \texttt{ => } exp \;\Rightarrow \tau} \tag{17}$$

Declarations

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

$$\frac{C \vdash valbind \Rightarrow VE}{C \vdash \texttt{val } valbind \Rightarrow \mathrm{Clos}_C VE \text{ in Env}} \tag{18}$$

$$\frac{C \vdash \mathit{typbind} \Rightarrow \mathit{TE}}{C \vdash \texttt{type}\ \mathit{typbind} \Rightarrow \mathit{TE} \text{ in Env}} \tag{19}$$

$$\frac{C \oplus \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \qquad \forall(t, \mathit{CE}) \in \mathrm{Ran}\,\mathit{TE},\ t \notin (T \text{ of } C)}{C \vdash \texttt{datatype}\ \mathit{datbind} \Rightarrow (\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{20}$$

$$\frac{\begin{array}{c} C \oplus \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \qquad \forall(t, \mathit{CE}) \in \mathrm{Ran}\,\mathit{TE},\ t \notin (T \text{ of } C) \\ C \oplus (\mathit{VE}, \mathit{TE}) \vdash \mathit{dec} \Rightarrow E \end{array}}{C \vdash \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\ \texttt{end} \Rightarrow E + \mathrm{Abs}\,\mathit{TE}} \tag{21}$$

$$\frac{C \vdash \mathit{exnbind} \Rightarrow \mathit{EE}}{C \vdash \texttt{exception}\ \mathit{exnbind} \Rightarrow \mathit{EE} \text{ in Env}} \tag{22}$$

$$\frac{C \vdash \mathit{dec}_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash \mathit{dec}_2 \Rightarrow E_2}{C \vdash \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \Rightarrow E_2} \tag{23}$$

$$\frac{C(\mathit{longstrid}_1) = (m_1, E_1) \quad \cdots \quad C(\mathit{longstrid}_n) = (m_n, E_n)}{C \vdash \texttt{open}\ \mathit{longstrid}_1 \cdots \mathit{longstrid}_n \Rightarrow E_1 + \cdots + E_n} \tag{24}$$

$$\frac{}{C \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{25}$$

$$\frac{C \vdash \mathit{dec}_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash \mathit{dec}_2 \Rightarrow E_2}{C \vdash \mathit{dec}_1\ \langle;\rangle\ \mathit{dec}_2 \Rightarrow E_1 + E_2} \tag{26}$$

Comments:

(18) 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 1.

(20),(21) The side condition is the formal way of expressing that the elaboration of each datatype binding generates new type names. Adding TE to the context on the left of the $\vdash$ captures the recursive nature of the binding. Recall that TE is assumed well-formed (as defined in Section 4.7). If $\mathrm{tynames}(E \text{ of } C) \subseteq T \text{ of } C$ and the side condition is satisfied then $C \oplus TE$ is well-formed.

(22) No closure operation is used here, since EE maps exception names to types rather than to general type schemes.

Value Bindings

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

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C \vdash exp \Rightarrow \tau \qquad \langle C \vdash valbind \Rightarrow VE' \rangle}{C \vdash pat = exp \ \langle \texttt{and}\ valbind \rangle \Rightarrow VE \ \langle + \ VE' \rangle} \tag{27}$$

$$\frac{C + VE \vdash valbind \Rightarrow VE}{C \vdash \texttt{rec}\ valbind \Rightarrow VE} \tag{28}$$

Comments:

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

(28) Modifying C by VE on the left captures the recursive nature of the binding. From rule 27 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

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

$$\frac{tyvarseq = \alpha^{(k)} \qquad C \vdash ty \Rightarrow \tau \qquad \langle C \vdash typbind \Rightarrow TE \rangle}{\begin{array}{l} C \vdash tyvarseq\ tycon = ty \ \langle \texttt{and}\ typbind \rangle \Rightarrow \\ \qquad \{tycon \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\} \ \langle + \ TE \rangle \end{array}} \tag{29}$$

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

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

$$\frac{tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)}t \vdash constrs \Rightarrow CE \qquad \langle C \vdash datbind \Rightarrow VE, TE \rangle}{\begin{array}{l} C \vdash tyvarseq\ tycon = constrs \ \langle \texttt{and}\ datbind \rangle \Rightarrow \\ \qquad \mathrm{Clos}CE \langle + \ VE \rangle,\ \{tycon \mapsto (t, \mathrm{Clos}CE)\} \ \langle + \ TE \rangle \end{array}} \tag{30}$$

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

Constructor Bindings

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

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash constrs \Rightarrow CE \rangle\rangle}{\begin{array}{l} C, \tau \vdash con \ \langle \texttt{of}\ ty \rangle \ \langle\langle \ \texttt{|}\ constrs \rangle\rangle \Rightarrow \\ \qquad \{con \mapsto \tau\} \ \langle + \ \{con \mapsto \tau' \to \tau\} \ \rangle \ \langle\langle + \ CE \rangle\rangle \end{array}} \tag{31}$$

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

Exception Bindings

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

$$\frac{\langle C \vdash ty \Rightarrow \tau\rangle \qquad \langle\langle C \vdash exnbind \Rightarrow EE\rangle\rangle}{\begin{array}{l} C \vdash exn\ \langle :\ ty\rangle\ \langle\langle \texttt{and}\ exnbind\rangle\rangle \Rightarrow \\ \qquad \{exn \mapsto \mathrm{unit}\}\ \langle + \ \{exn \mapsto \tau\}\ \rangle\ \langle\langle + \ EE\rangle\rangle \end{array}} \tag{32}$$

$$\frac{C(longexn) = \tau \qquad \langle C \vdash ty \Rightarrow \tau\rangle \qquad \langle\langle C \vdash exnbind \Rightarrow EE\rangle\rangle}{\begin{array}{l} C \vdash exn\ \langle :\ ty\rangle\ \texttt{=}\ longexn\ \langle\langle \texttt{and}\ exnbind\rangle\rangle \Rightarrow \\ \qquad \{exn \mapsto \tau\}\ \langle\langle + \ EE\rangle\rangle \end{array}} \tag{33}$$

Comments:

(32),(33) No matter which of the options are present, given C and *exnbind* there is at most one EE such that $C \vdash exnbind \Rightarrow EE$.

Atomic Patterns

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

$$\frac{}{C \vdash \_ \Rightarrow (\{\}, \tau)} \tag{34}$$

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

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

$$\frac{\langle C \vdash patrow \Rightarrow (VE, \varrho)\rangle}{C \vdash \texttt{\{}\ \langle patrow\rangle\ \texttt{\}} \Rightarrow (\ \{\}\langle + \ VE\rangle,\ \{\}\langle + \ \varrho\rangle \text{ in Type }\ )} \tag{37}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau)}{C \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow (VE, \tau)} \tag{38}$$

Comments:

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

Pattern Rows

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

$$\frac{}{C \vdash \ldots \Rightarrow (\{\}, \varrho)} \tag{39}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad \langle C \vdash patrow \Rightarrow (VE', \varrho) \qquad lab \notin \mathrm{Dom}\,\varrho\rangle}{C \vdash lab\ \texttt{=}\ pat\ \langle\ \texttt{,}\ patrow\rangle \Rightarrow (VE\langle + \ VE'\rangle,\ \{lab \mapsto \tau\}\langle + \ \varrho\rangle)} \tag{40}$$

Comment:

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

Patterns

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

$$\frac{C \vdash atpat \Rightarrow (VE, \tau)}{C \vdash atpat \Rightarrow (VE, \tau)} \tag{41}$$

$$\frac{C(longcon) \succ \tau' \to \tau \qquad C \vdash atpat \Rightarrow (VE, \tau')}{C \vdash longcon\ atpat \Rightarrow (VE, \tau)} \tag{42}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C \vdash ty \Rightarrow \tau}{C \vdash pat\ :\ ty \Rightarrow (VE, \tau)} \tag{43}$$

$$\frac{\begin{array}{c} C \vdash var \Rightarrow (VE, \tau) \qquad \langle C \vdash ty \Rightarrow \tau \rangle \\ C \vdash pat \Rightarrow (VE', \tau) \end{array}}{C \vdash var\langle :\ ty\rangle\ \mathtt{as}\ pat \Rightarrow (VE + VE', \tau)} \tag{44}$$

Comments:

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

Type Expressions

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

$$\frac{tyvar = \alpha}{C \vdash tyvar \Rightarrow \alpha} \tag{45}$$

$$\frac{\langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash \{\ \langle tyrow \rangle\ \} \Rightarrow \{\}\langle + \varrho\rangle \text{ in Type}} \tag{46}$$

$$\frac{\begin{array}{c} tyseq = ty_1 \cdots ty_k \qquad C \vdash ty_i \Rightarrow \tau_i\ (1 \leq i \leq k) \\ C(longtycon) = (\theta, CE) \end{array}}{C \vdash tyseq\ longtycon \Rightarrow \tau^{(k)}\theta} \tag{47}$$

$$\frac{C \vdash ty \Rightarrow \tau \qquad C \vdash ty' \Rightarrow \tau'}{C \vdash ty\ \texttt{->}\ ty' \Rightarrow \tau \to \tau'} \tag{48}$$

$$\frac{C \vdash ty \Rightarrow \tau}{C \vdash (\ ty\ ) \Rightarrow \tau} \tag{49}$$

Comments:

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

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

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash tyrow \Rightarrow \varrho \rangle}{C \vdash lab\ :\ ty\ \langle\ ,\ tyrow \rangle \Rightarrow \{lab \mapsto \tau\} \langle + \ \varrho \rangle} \tag{50}$$

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

4.9 Further Restrictions

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

1. The reference value constructor `ref` may occur in patterns with polymorphic type, but in an expression it must always elaborate to a monotype, i.e. a type containing no type variables. This restriction will be relaxed in future Versions, but some restriction will always be necessary to ensure soundness of the type discipline.

2. 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.

3. In a match, the sequence of patterns $pat_1, \cdots, pat_n$ must be *irredundant* and *exhaustive*. That is, each pat_j must match *some* value (of the right type) which is not matched by pat_i for any $i < j$, and *every* value (of the right type) must be matched by some pat_i. The compiler must give a warning on violation of this restriction, but should still compile the match.

4.10 Principal Environments

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 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 14.

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

Figure 14: 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.

We shall often need to change bound names in semantic objects. For example, we sometimes require that $N \cap N' = \emptyset$ in a functor signature. More generally, for arbitrary A it is sometimes convenient to assume that *all* nameset prefixes N occurring in A are disjoint. In that case we say that we are *disjoining bound names* in A.

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 $\mathrm{Ran}\, SE$ 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. 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 $B + (\mathrm{names}\, SE, SE)$.

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

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

A semantic object A or assembly A of objects is said to be *consistent* if, after disjoining bound names, for all S_1 and S_2 in A and for every *longstrid* and every *longtycon*

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

$$m \text{ of } S_1(\textit{longstrid}) \quad = \quad m \text{ of } S_2(\textit{longstrid})$$

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

$$\theta \text{ of } S_1(\textit{longtycon}) \quad = \quad \theta \text{ of } S_2(\textit{longtycon})$$

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 $N \cap N' = \emptyset$, $A = \{S_1, S_2\}$ 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

Conditions for the well-formedness of type environments TE are given with the Core static semantics.

A signature $(N)S$ is *well-formed* if, 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 $(N)S$, $(N')S'$ and $(N \cup N')S'$ are well-formed, and $\text{names}(N')S' \cap N \subseteq \text{names } S$ (the latter condition is satisfied automatically for user-defined functors).

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; in particular, the admissibility of each semantic object mentioned is taken as a condition throughout the semantic rules which follow. (In our semantic description we have not undertaken to indicate how admissibility should be checked in an implementation.)

5.6 Type Realisation

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

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

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 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 $\mathrm{Supp}\,\varphi \subseteq N_1$. 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 $N_2 \cap (\mathrm{names}\,\Sigma_1) = \emptyset$. We claim that $\Sigma_1 \geq \Sigma_2$ iff, for all S, whenever $\Sigma_2 \geq S$ then $\Sigma_1 \geq S$.

5.9 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 $\mathrm{Supp}\,\varphi \subseteq N_1$.

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

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 $CE_2 = \{\}$

5.11 Principal Signatures

Let B be a basis, and suppose that $B \vdash \mathit{sigexp} \Rightarrow S$ according to the rules below. Then $(N)S$ is *principal* (for *sigexp* in the basis B) if $(N \text{ of } B) \cap N = \emptyset$, and for all S' for which $B \vdash \mathit{sigexp} \Rightarrow S'$ we have $(N)S \geq S'$. We claim that if *sigexp* elaborates to any structure S in B then it possesses a principal signature in B.

5.12 Inference Rules

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

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

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

Although not assumed in our definitions, it is intended that every basis $B = N, F, G, E$ in which a *program* is elaborated has the property that $\text{names}\, F \cup \text{names}\, G \cup \text{names}\, E \subseteq N$. 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 $B \vdash \textit{program} \Rightarrow B'$ in which B satisfies the above condition. Then B' also satisfies the condition.
>
> Moreover, if S$'$ is a sentence of the form $B'' \vdash \textit{phrase} \Rightarrow A$ occurring in a proof of S, where *phrase* is either a structure expression or a structure declaration, then B'' also satisfies the condition.
>
> Finally, if $T, E \vdash \textit{phrase} \Rightarrow A$ occurs in a proof of S, where *phrase* is a phrase of the Core, then $\text{tynames}\, E \subseteq T$.

Structure Expressions

$$\boxed{B \vdash \textit{strexp} \Rightarrow S}$$

$$\frac{B \vdash \textit{strdec} \Rightarrow E \qquad m \notin (N \text{ of } B) \cup \text{names}\, E}{B \vdash \texttt{struct}\ \textit{strdec}\ \texttt{end} \Rightarrow (m, E)} \tag{51}$$

$$\frac{B(\textit{longstrid}) = S}{B \vdash \textit{longstrid} \Rightarrow S} \tag{52}$$

$$\frac{\begin{array}{c} B \vdash \textit{strexp} \Rightarrow S \\ B(\textit{funid}) \geq (S'', (N')S')\ ,\ S \succ S'' \\ (N \text{ of } B) \cap N' = \emptyset \end{array}}{B \vdash \textit{funid}\ (\ \textit{strexp}\ )\ \Rightarrow S'} \tag{53}$$

$$\frac{B \vdash \textit{strdec} \Rightarrow E \qquad B \oplus E \vdash \textit{strexp} \Rightarrow S}{B \vdash \texttt{let}\ \textit{strdec}\ \texttt{in}\ \textit{strexp}\ \texttt{end} \Rightarrow S} \tag{54}$$

Comments:

(51) 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 95; this will ensure that for each application of the functor, by rule 53, a new distinct name will be chosen for the structure generated.

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

The realisation φ for which $\varphi(B(\mathit{funid})) = (S'', (N')S')$ is uniquely determined by $B(\mathit{funid})$ and 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 specified in the declaration of the functor *funid* is represented by the occurrence of the same name in both components of $B(\mathit{funid})$, and this repeated occurrence is preserved by φ, yielding sharing between the argument structure S and the result structure S' of this functor application.

(54) 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 $\boxed{B \vdash \mathit{strdec} \Rightarrow E}$

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E \qquad E \text{ principal in } (C \text{ of } B)}{B \vdash \mathit{dec} \Rightarrow E} \tag{55}$$

$$\frac{B \vdash \mathit{strbind} \Rightarrow SE}{B \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow SE \text{ in Env}} \tag{56}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2} \tag{57}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{58}$$

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \mathit{strdec}_1\ \langle ; \rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2} \tag{59}$$

Comments:

(55) The side condition ensures that all type schemes in E are as general as possible and that all new type names in E admit equality, if possible.

Structure-level Bindings

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

$$\frac{\begin{array}{c} B \vdash strexp \Rightarrow S \qquad \langle B \vdash sigexp \Rightarrow S' \;,\; S \succ S' \rangle \\ \langle\langle B + \text{names}\, S \vdash strbind \Rightarrow SE \rangle\rangle \end{array}}{B \vdash strid \ \langle :\ sigexp \rangle\ \texttt{=}\ strexp\ \langle\langle \texttt{and}\ strbind \rangle\rangle \Rightarrow \{strid \mapsto S\langle ' \rangle\}\ \langle\langle + \ SE \rangle\rangle} \quad (60)$$

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

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

$$\frac{B \vdash spec \Rightarrow E}{B \vdash \texttt{sig}\ spec\ \texttt{end} \Rightarrow (m, E)} \quad (61)$$

$$\frac{B(sigid) \geq S}{B \vdash sigid \Rightarrow S} \quad (62)$$

Comments:

(61) In contrast to rule 51, m is not here required to be new. The name m may be chosen to achieve the sharing required in rule 84, or to achieve the enrichment side conditions of rule 60 or 95. The choice of m must result in an admissible object.

(62) The instance S of $B(sigid)$ is not determined by this rule, but – as in rule 61 – the instance may be chosen to achieve sharing properties or enrichment conditions.

Signature Declarations

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

$$\frac{B \vdash sigbind \Rightarrow G}{B \vdash \texttt{signature}\ sigbind \Rightarrow G} \quad (63)$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\}} \quad (64)$$

$$\frac{B \vdash sigdec_1 \Rightarrow G_1 \qquad B + G_1 \vdash sigdec_2 \Rightarrow G_2}{B \vdash sigdec_1\ \langle ; \rangle\ sigdec_2 \Rightarrow G_1 + G_2} \quad (65)$$

Comments:

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

Signature Bindings

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

$$\frac{B \vdash sigexp \Rightarrow S \qquad (N)S \text{ principal in B} \qquad \langle B \vdash sigbind \Rightarrow G\rangle}{B \vdash sigid = sigexp\ \langle\text{and } sigbind\rangle \Rightarrow \{sigid \mapsto (N)S\}\ \langle + \ G\rangle} \tag{66}$$

Comment: The principality condition ensures that the signature found is as general as possible given the sharing constraints present in *sigexp*. The set N is determined by the definition of principality in Section 5.11.

Specifications

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

$$\frac{C \text{ of } B \vdash valdesc \Rightarrow VE}{B \vdash \texttt{val}\ valdesc \Rightarrow \text{Clos}VE \text{ in Env}} \tag{67}$$

$$\frac{C \text{ of } B \vdash typdesc \Rightarrow TE}{B \vdash \texttt{type}\ typdesc \Rightarrow TE \text{ in Env}} \tag{68}$$

$$\frac{C \text{ of } B \vdash typdesc \Rightarrow TE \qquad \forall(\theta, CE) \in \text{Ran}\, TE,\ \theta \text{ admits equality}}{B \vdash \texttt{eqtype}\ typdesc \Rightarrow TE \text{ in Env}} \tag{69}$$

$$\frac{C \text{ of } B + TE \vdash datdesc \Rightarrow VE, TE}{B \vdash \texttt{datatype}\ datdesc \Rightarrow (VE, TE) \text{ in Env}} \tag{70}$$

$$\frac{C \text{ of } B \vdash exndesc \Rightarrow EE}{B \vdash \texttt{exception}\ exndesc \Rightarrow EE \text{ in Env}} \tag{71}$$

$$\frac{B \vdash strdesc \Rightarrow SE}{B \vdash \texttt{structure}\ strdesc \Rightarrow SE \text{ in Env}} \tag{72}$$

$$\frac{B \vdash shareq \Rightarrow \{\}}{B \vdash \texttt{sharing}\ shareq \Rightarrow \{\} \text{ in Env}} \tag{73}$$

$$\frac{B \vdash spec_1 \Rightarrow E_1 \qquad B + E_1 \vdash spec_1 \Rightarrow E_2}{B \vdash \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow E_2} \tag{74}$$

$$\frac{B(longstrid_1) = (m_1, E_1) \quad \cdots \quad B(longstrid_n) = (m_n, E_n)}{B \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n} \tag{75}$$

$$\frac{B(sigid_1) \geq (m_1, E_1) \quad \cdots \quad B(sigid_n) \geq (m_n, E_n)}{B \vdash \texttt{include}\ sigid_1 \cdots sigid_n \Rightarrow E_1 + \cdots + E_n} \tag{76}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{77}$$

$$\frac{B \vdash spec_1 \Rightarrow E_1 \qquad B + E_1 \vdash spec_1 \Rightarrow E_2}{B \vdash spec_1 \ \langle ; \rangle \ spec_2 \Rightarrow E_1 + E_2} \tag{78}$$

Comments:

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

(68)–(70) The type functions in TE may be chosen to achieve the sharing hypotesis of rule 85 or the enrichment conditions of rules 60 and 95. In particular, the type names in TE in rule 70 need not be new. Also, in rule 68 the type functions in TE may admit equality.

(71) EE is determined by B and *exndesc* and contains monotypes only.

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

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

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash valdesc \Rightarrow VE \rangle}{C \vdash var \ : \ ty \ \langle \text{and } valdesc \rangle \Rightarrow \{var \mapsto \tau\} \ \langle + \ VE \rangle} \tag{79}$$

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

$$\frac{tyvarseq = \alpha^{(k)} \qquad \langle C \vdash typdesc \Rightarrow TE \rangle \qquad \text{arity}\,\theta = k}{C \vdash tyvarseq \ tycon \ \langle \text{and } typdesc \rangle \Rightarrow \{tycon \mapsto (\theta, \{\})\} \ \langle + \ TE \rangle} \tag{80}$$

Comment: Note that any θ of arity k may be chosen.

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

$$\frac{tyvarseq = \alpha^{(k)} \qquad C, \alpha^{(k)}t \vdash constrs \Rightarrow CE \qquad \langle C \vdash datdesc \Rightarrow VE, TE \rangle}{\begin{array}{c} C \vdash tyvarseq \ tycon = constrs \ \langle \text{and } datdesc \rangle \Rightarrow \\ \text{Clos}CE \langle + \ VE \rangle, \ \{tycon \mapsto (t, \text{Clos}CE)\} \ \langle + \ TE \rangle \end{array}} \tag{81}$$

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

$$\frac{C \vdash ty \Rightarrow \tau \qquad \text{tyvars}(\tau) = \emptyset \qquad \langle C \vdash exndesc \Rightarrow EE \rangle}{C \vdash exn \ : \ ty \ \langle \text{and } exndesc \rangle \Rightarrow \{exn \mapsto \tau\} \ \langle + \ EE \rangle} \tag{82}$$

Structure Descriptions

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

$$\frac{B \vdash \mathit{sigexp} \Rightarrow S \qquad \langle B \vdash \mathit{strdesc} \Rightarrow SE \rangle}{B \vdash \mathit{strid} \;:\; \mathit{sigexp} \;\langle \texttt{and}\ \mathit{strdesc} \rangle \Rightarrow \{\mathit{strid} \mapsto S\} \;\langle + \;SE \rangle} \tag{83}$$

Sharing Equations

$$\boxed{B \vdash \mathit{shareq} \Rightarrow \{\}}$$

$$\frac{m \text{ of } (B(\mathit{longstrid}_1)) = \cdots = m \text{ of } (B(\mathit{longstrid}_n))}{B \vdash \mathit{longstrid}_1 \;\texttt{=}\; \cdots \;\texttt{=}\; \mathit{longstrid}_n \;\Rightarrow \{\}} \tag{84}$$

$$\frac{B(\mathit{longtycon}_1) = \cdots = B(\mathit{longtycon}_n)}{B \vdash \texttt{type}\ \mathit{longtycon}_1 \;\texttt{=}\; \cdots \;\texttt{=}\; \mathit{longtycon}_n \;\Rightarrow \{\}} \tag{85}$$

$$\frac{B \vdash \mathit{shareq}_1 \Rightarrow \{\} \qquad B \vdash \mathit{shareq}_2 \Rightarrow \{\}}{B \vdash \mathit{shareq}_1 \;\texttt{and}\; \mathit{shareq}_2 \Rightarrow \{\}} \tag{86}$$

Comments:

(84) By the definition of consistency the premise is weaker than

$$B(\mathit{longstrid}_1) = \cdots = B(\mathit{longstrid}_n)$$

Two different structures with the same name may be thought of as representing different views.

Functor Specifications

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

$$\frac{B \vdash \mathit{fundesc} \Rightarrow F}{B \vdash \texttt{functor}\ \mathit{fundesc} \Rightarrow F} \tag{87}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\}} \tag{88}$$

$$\frac{B \vdash \mathit{funspec}_1 \Rightarrow F_1 \qquad B + F_1 \vdash \mathit{funspec}_2 \Rightarrow F_2}{B \vdash \mathit{funspec}_1 \;\langle ; \rangle\; \mathit{funspec}_2 \Rightarrow F_1 + F_2} \tag{89}$$

Functor Descriptions

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

$$\frac{B \vdash \mathit{funsigexp} \Rightarrow \Phi \qquad \langle B \vdash \mathit{fundesc} \Rightarrow F \rangle}{B \vdash \mathit{funid}\ \mathit{funsigexp} \;\langle \texttt{and}\ \mathit{fundesc} \rangle \Rightarrow \{\mathit{funid} \mapsto \Phi\} \langle + \;F \rangle} \tag{90}$$

Functor Signature Expressions

$\boxed{B \vdash \mathit{funsigexp} \Rightarrow \Phi}$

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow S \qquad (N)S \text{ principal in } B \\ B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow S' \\ N' = \text{names}\, S' \setminus ((N \text{ of } B) \cup N) \end{array}}{B \vdash \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{)}\ \texttt{:}\ \mathit{sigexp}' \Rightarrow (N)(S, (N')S')} \qquad (91)$$

Functor Declarations

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

$$\frac{B \vdash \mathit{funbind} \Rightarrow F}{B \vdash \texttt{functor}\ \mathit{funbind} \Rightarrow F} \qquad (92)$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\}} \qquad (93)$$

$$\frac{B \vdash \mathit{fundec}_1 \Rightarrow F_1 \qquad B + F_1 \vdash \mathit{fundec}_2 \Rightarrow F_2}{B \vdash \mathit{fundec}_1\ \langle\texttt{;}\rangle\ \mathit{fundec}_2 \Rightarrow F_1 + F_2} \qquad (94)$$

Functor Bindings

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

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow S \qquad (N)S \text{ principal in } B \\ B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{strexp} \Rightarrow S' \\ \langle B \oplus \{\mathit{strid} \mapsto S\} \vdash \mathit{sigexp}' \Rightarrow S'',\ S' \succ S'' \rangle \\ N' = \text{names}\, S' \setminus ((N \text{ of } B) \cup N) \\ \langle\langle B \vdash \mathit{funbind} \Rightarrow F \rangle\rangle \end{array}}{\begin{array}{c} B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strid}\ \texttt{:}\ \mathit{sigexp}\ \texttt{)}\ \langle\texttt{:}\ \mathit{sigexp}'\rangle\ \texttt{=}\ \mathit{strexp}\ \langle\langle \text{and}\ \mathit{funbind}\rangle\rangle \Rightarrow \\ \{\mathit{funid} \mapsto (N)(S, (N')S'\langle'\rangle)\}\ \langle\langle + F\rangle\rangle \end{array}} \qquad (95)$$

Comment: Here $(N)S$ is required to be principal so as to be as general as possible given the sharing constraints in *sigexp*. 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.

Programs

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

$$\frac{B \vdash \mathit{strdec} \Rightarrow E}{B \vdash \mathit{strdec} \Rightarrow (\text{names}\, E, E) \text{ in Basis}} \qquad (96)$$

$$\frac{B \vdash \mathit{sigdec} \Rightarrow G}{B \vdash \mathit{sigdec} \Rightarrow (\mathrm{names}\, G, G) \text{ in Basis}} \tag{97}$$

$$\frac{B \vdash \mathit{fundec} \Rightarrow F}{B \vdash \mathit{fundec} \Rightarrow (\mathrm{names}\, F, F) \text{ in Basis}} \tag{98}$$

$$\frac{B \vdash \mathit{program}_1 \Rightarrow B_1 \qquad B + B_1 \vdash \mathit{program}_2 \Rightarrow B_2}{B \vdash \mathit{program}_1 \ \langle ; \rangle\ \mathit{program}_2 \Rightarrow B_1 + B_2} \tag{99}$$

5.13 Functor Matching

As pointed out in Section 3.4 on the grammar for Modules, there is no phrase class whose elaboration requires matching a functor to a functor specification. 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

1. There is a realisation φ, $\mathrm{Supp}\,\varphi \subseteq N_2$, such that $\varphi S_2 \prec S_1$
2. Assuming that $\mathrm{Yield}\,\varphi \cap N_2' = \emptyset$, there is a realisation φ', $\mathrm{Supp}\,\varphi' \subseteq N_1'$, such that $\varphi' S_1' \prec \varphi S_2'$

The first condition ensures that the real functor signature Φ_2 for f requires the argument *strexp* of any application $f(\mathit{strexp})$ 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 $(\varphi S_2, (N_2')\varphi S_2')$, provides in the result of the application $f(\mathit{strexp})$ 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.
- 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*, *constrs*, *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 15.

a	$\in$	Addr	addresses
e	$\in$	Exc	exceptions
b	$\in$	BasVal	basic values
		{FAIL}	failure

Figure 15: Simple Semantic Objects

Addr and Exc are infinite sets. BasVal is described below. FAIL is the result of a failing attempt to match a value and a pattern or of a failing attempt to handle an exception with a handle rule. Thus FAIL is neither a value nor an exception, but simply a semantic object used in the rules to express operationally how matching proceeds.

6.3 Compound Objects

The compound objects for the dynamic semantics are shown in Figure 16. Many conventions and notations are adopted as in the static semantics; in particular

$$
\begin{array}{rcl}
v & \in & \mathrm{Val} = \{\mathtt{:=}\} \cup \mathrm{BasVal} \cup \mathrm{Con} \cup (\mathrm{Con} \times \mathrm{Val}) \cup \\
 & & \qquad\qquad \mathrm{Record} \cup \mathrm{Addr} \cup \mathrm{Closure} \\
r & \in & \mathrm{Record} = \mathrm{Lab} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
{[e,v]} \text{ or } p & \in & \mathrm{Pack} = \mathrm{Exc} \times \mathrm{Val} \\
(match, E, VE) & \in & \mathrm{Closure} = \mathrm{Match} \times \mathrm{Env} \times \mathrm{VarEnv} \\
mem & \in & \mathrm{Mem} = \mathrm{Addr} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
excs & \in & \mathrm{ExcSet} = \mathrm{Fin}(\mathrm{Exc}) \\
(mem, excs) \text{ or } s & \in & \mathrm{State} = \mathrm{Mem} \times \mathrm{ExcSet} \\
(SE, VE, EE) \text{ or } E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{VarEnv} \times \mathrm{ExnEnv} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \xrightarrow{\mathrm{fin}} \mathrm{Env} \\
VE & \in & \mathrm{VarEnv} = \mathrm{Var} \xrightarrow{\mathrm{fin}} \mathrm{Val} \\
EE & \in & \mathrm{ExnEnv} = \mathrm{Exn} \xrightarrow{\mathrm{fin}} \mathrm{Exc}
\end{array}
$$

Figure 16: Compound Semantic Objects

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.

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 completely 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:

```
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

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

which is detailed in Appendix D.

6.5 Basic Exceptions

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

```
ord  chr  div  mod  /  *  +  -  floor  sqrt  exp  ln
          io_failure  match  bind  interrupt
```

The exceptions on the first line are raised by basic functions of the same name, and `io_failure` 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 *match* or a *valbind*, as detailed in the rules to follow. Finally, `interrupt` is raised by external intervention.

In a match of the form pat_1 => exp_1 | ... | pat_n => exp_n the pattern sequence $pat_1, \ldots, pat_n$ should be *irredundant* and *exhaustive*. That is, each pat_j must match some value (of the right type) which is not matched by pat_i for any $i < j$, and every value (of the right type) must be matched by some pat_i. The compiler must give warning on violation of this restriction, but should still compile the match. Thus the `match` exception will only be raised for a match which has been flagged by the compiler. The restriction is inherited by derived forms; in particular, this means that in the function binding *var* $atpat_1 \cdots atpat_n\langle$: *ty*$\rangle$ (consisting of one clause only), each separate $atpat_i$ should be exhaustive by itself.

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 :: 1` =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 $VE(var) \notin \mathrm{Closure}$, 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

$$\frac{\begin{array}{c} A_1 \vdash phrase_1 \Rightarrow A'_1 \qquad A_2 \vdash phrase_2 \Rightarrow A'_2 \quad \cdots \\ \qquad\qquad \cdots \quad A_n \vdash phrase_n \Rightarrow A'_n \end{array}}{A \vdash phrase \Rightarrow A'}$$

then the full form is intended to be

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

(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.[1] This indicates that evaluation of phrases in the hypothesis terminates with the first whose result is a packet (other than one already treated in the rule), and this packet is the result of the phrase in the conclusion.

A third convention is that we allow compound variables (variables built from the variables in Figure 16 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/FAIL where x ranges over X; furthermore, we extend environment modification to allow for failure as follows:

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

Atomic Expressions

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

$$\frac{E(\mathit{longvar}) = v}{E \vdash \mathit{longvar} \Rightarrow v} \tag{100}$$

[1] There is one exception to the exception convention; no extra rule is added for rule 112 which deals with handlers, since a handler is the only means by which propagation of an exception can be arrested.

$$\frac{\textit{longcon} = \textit{strid}_1.\cdots.\textit{strid}_k.\textit{con}}{E \vdash \textit{longcon} \Rightarrow \textit{con}} \tag{101}$$

$$\frac{\langle E \vdash \textit{exprow} \Rightarrow r\rangle}{E \vdash \texttt{\{}\ \langle \textit{exprow}\rangle\ \texttt{\}} \Rightarrow \{\}\langle + r\rangle \text{ in Val}} \tag{102}$$

$$\frac{E \vdash \textit{dec} \Rightarrow E' \qquad E + E' \vdash \textit{exp} \Rightarrow v}{E \vdash \texttt{let}\ \textit{dec}\ \texttt{in}\ \textit{exp}\ \texttt{end} \Rightarrow v} \tag{103}$$

$$\frac{E \vdash \textit{exp} \Rightarrow v}{E \vdash \texttt{(}\ \textit{exp}\ \texttt{)} \Rightarrow v} \tag{104}$$

Comments:

(101) Constructors denote themselves

Expression Rows

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

$$\frac{E \vdash \textit{exp} \Rightarrow v \qquad \langle E \vdash \textit{exprow} \Rightarrow r\rangle}{E \vdash \textit{lab}\ \texttt{=}\ \textit{exp}\ \langle\ \texttt{,}\ \textit{exprow}\rangle \Rightarrow \{\textit{lab} \mapsto v\}\langle + r\rangle} \tag{105}$$

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

Expressions

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

$$\frac{E \vdash \textit{atexp} \Rightarrow v}{E \vdash \textit{atexp} \Rightarrow v} \tag{106}$$

$$\frac{E \vdash \textit{exp} \Rightarrow \textit{con} \qquad \textit{con} \neq \texttt{ref} \qquad E \vdash \textit{atexp} \Rightarrow v}{E \vdash \textit{exp}\ \textit{atexp} \Rightarrow (\textit{con}, v)} \tag{107}$$

$$\frac{s, E \vdash \textit{exp} \Rightarrow\ \texttt{ref}\ , s' \qquad s', E \vdash \textit{atexp} \Rightarrow v, s'' \qquad a \notin \text{Dom}(\textit{mem}\text{ of } s'')}{s, E \vdash \textit{exp}\ \textit{atexp} \Rightarrow a,\ s'' + \{a \mapsto v\}} \tag{108}$$

$$\frac{s, E \vdash \textit{exp} \Rightarrow\ \texttt{:=}\ , s' \qquad s', E \vdash \textit{atexp} \Rightarrow \{1 \mapsto a,\ 2 \mapsto v\}, s''}{s, E \vdash \textit{exp}\ \textit{atexp} \Rightarrow \{\} \text{ in Val},\ s'' + \{a \mapsto v\}} \tag{109}$$

$$\frac{E \vdash \textit{exp} \Rightarrow b \qquad E \vdash \textit{atexp} \Rightarrow v \qquad \text{APPLY}(b, v) = v'}{E \vdash \textit{exp}\ \textit{atexp} \Rightarrow v'} \tag{110}$$

$$\frac{E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \qquad E' + \mathrm{Rec}\,VE,\ v \vdash match \Rightarrow v'}{E \vdash exp\ atexp \Rightarrow v'} \tag{111}$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash exp\ \mathtt{handle}\ handler \Rightarrow v} \tag{112}$$

$$\frac{E \vdash exp \Rightarrow p \qquad E, p \vdash handler \Rightarrow v}{E \vdash exp\ \mathtt{handle}\ handler \Rightarrow v} \tag{113}$$

$$\frac{E(longexn) = e \qquad E \vdash exp \Rightarrow v}{E \vdash \mathtt{raise}\ longexn\ \mathtt{with}\ exp \Rightarrow [e, v]} \tag{114}$$

$$\frac{}{E \vdash \mathtt{fn}\ match \Rightarrow (match, E, \{\})} \tag{115}$$

Comments:

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

(107)–(111) 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, so v will be either a closure, a constructor, a basic value or `:=`.

(112) This is the only rule to which the exception convention does not apply. If the operator evaluates to a packet then rule 113 must be used.

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

Matches

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

$$\frac{E, v \vdash mrule \Rightarrow v'}{E, v \vdash mrule\ \langle\ \mid\ match\rangle \Rightarrow v'} \tag{116}$$

$$\frac{E, v \vdash mrule \Rightarrow \mathrm{FAIL}}{E, v \vdash mrule \Rightarrow [\mathtt{match}, \{\}]} \tag{117}$$

$$\frac{E, v \vdash mrule \Rightarrow \mathrm{FAIL} \qquad E, v \vdash match \Rightarrow v'}{E \vdash mrule\ \mid\ match \Rightarrow v'} \tag{118}$$

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

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

$$\frac{v \vdash pat \Rightarrow VE \qquad E + VE \vdash exp \Rightarrow v'}{E, v \vdash pat \;\texttt{=>}\; exp \;\Rightarrow v'} \tag{119}$$

$$\frac{v \vdash pat \Rightarrow \mathrm{FAIL}}{E, v \vdash pat \;\texttt{=>}\; exp \;\Rightarrow \mathrm{FAIL}} \tag{120}$$

Handlers

$$\boxed{E, p \vdash handler \Rightarrow v/p}$$

$$\frac{E, p \vdash hrule \Rightarrow v}{E, p \vdash hrule \;\langle\; \texttt{||}\; handler\rangle \Rightarrow v} \tag{121}$$

$$\frac{E, p \vdash hrule \Rightarrow \mathrm{FAIL}}{E, p \vdash hrule \Rightarrow p} \tag{122}$$

$$\frac{E, p \vdash hrule \Rightarrow \mathrm{FAIL} \qquad E, p \vdash handler \Rightarrow v}{E, p \vdash hrule \;\texttt{||}\; handler \Rightarrow v} \tag{123}$$

Handle Rules

$$\boxed{E, p \vdash hrule \Rightarrow v/p/\mathrm{FAIL}}$$

$$\frac{E(longexn) \neq e}{E, [e, v] \vdash longexn \;\texttt{with}\; match \Rightarrow \mathrm{FAIL}} \tag{124}$$

$$\frac{E(longexn) = e \qquad E, v \vdash match \Rightarrow v'}{E, [e, v] \vdash longexn \;\texttt{with}\; match \Rightarrow v'} \tag{125}$$

$$\frac{E \vdash exp \Rightarrow v}{E, p \vdash \texttt{?} \;\texttt{=>}\; exp \;\Rightarrow v} \tag{126}$$

Comments:

(126) This form of handle rule handles all exceptions.

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

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \mathtt{val}\ valbind \Rightarrow VE \text{ in Env}} \tag{127}$$

$$\frac{E \vdash exnbind \Rightarrow EE}{E \vdash \mathtt{exception}\ exnbind \Rightarrow EE \text{ in Env}} \tag{128}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash \mathtt{local}\ dec_1\ \mathtt{in}\ dec_2\ \mathtt{end} \Rightarrow E_2} \tag{129}$$

$$\frac{E(longstrid_1) = E_1 \quad \cdots \quad E(longstrid_k) = E_k}{E \vdash \mathtt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_k} \tag{130}$$

$$\frac{}{E \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{131}$$

$$\frac{E \vdash dec_1 \Rightarrow E_1 \qquad E + E_1 \vdash dec_2 \Rightarrow E_2}{E \vdash dec_1\ \langle;\rangle\ dec_2 \Rightarrow E_1 + E_2} \tag{132}$$

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

$$\frac{E \vdash exp \Rightarrow v \qquad v \vdash pat \Rightarrow VE \qquad \langle E \vdash valbind \Rightarrow VE' \rangle}{E \vdash pat\ \mathtt{=}\ exp\ \langle \mathtt{and}\ valbind \rangle \Rightarrow VE\ \langle + VE' \rangle} \tag{133}$$

$$\frac{E \vdash exp \Rightarrow v \qquad v \vdash pat \Rightarrow \text{FAIL}}{E \vdash pat\ \mathtt{=}\ exp\ \langle \mathtt{and}\ valbind \rangle \Rightarrow [\mathtt{bind}, \{\}]} \tag{134}$$

$$\frac{E \vdash valbind \Rightarrow VE}{E \vdash \mathtt{rec}\ valbind \Rightarrow \text{Rec}\, VE} \tag{135}$$

Exception Bindings $\boxed{E \vdash exnbind \Rightarrow EE/p}$

$$\frac{e \notin excs \text{ of } s \qquad s' = s + \{e\} \qquad \langle s', E \vdash exnbind \Rightarrow EE, s'' \rangle}{s, E \vdash exn\ \langle \mathtt{and}\ exnbind \rangle \Rightarrow \{exn \mapsto e\} \langle + EE \rangle,\ s' \langle ' \rangle} \tag{136}$$

$$\frac{E(longexn) = e \qquad \langle E \vdash exnbind \Rightarrow EE \rangle}{E \vdash exn\ \mathtt{=}\ longexn\ \langle \mathtt{and}\ exnbind \rangle \Rightarrow \{exn \mapsto e\} \langle + EE \rangle} \tag{137}$$

Comments:

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

Atomic Patterns

$$\boxed{v \vdash atpat \Rightarrow VE/\mathrm{FAIL}}$$

$$\frac{}{v \vdash \_ \Rightarrow \{\}} \tag{138}$$

$$\frac{}{v \vdash var \Rightarrow \{var \mapsto v\}} \tag{139}$$

$$\frac{longcon = strid_1.\cdots.strid_k.con \qquad v = con}{v \vdash longcon \Rightarrow \{\}} \tag{140}$$

$$\frac{longcon = strid_1.\cdots.strid_k.con \qquad v \neq con}{v \vdash longcon \Rightarrow \mathrm{FAIL}} \tag{141}$$

$$\frac{v = \{\}\langle + r\rangle \text{ in Val} \qquad \langle r \vdash patrow \Rightarrow VE/\mathrm{FAIL}\rangle}{v \vdash \{\langle patrow\rangle\} \Rightarrow \{\}\langle + VE/\mathrm{FAIL}\rangle} \tag{142}$$

$$\frac{v \vdash pat \Rightarrow VE}{v \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow VE} \tag{143}$$

Comments:

(141) Any evaluation resulting in FAIL must do so because rule 141 or rule 149 has been applied.

Labelled Patterns

$$\boxed{r \vdash patrow \Rightarrow VE/\mathrm{FAIL}}$$

$$\frac{}{r \vdash \texttt{...} \Rightarrow \{\}} \tag{144}$$

$$\frac{r(lab) \vdash pat \Rightarrow \mathrm{FAIL}}{r \vdash lab \texttt{ = } pat \ \langle \texttt{ , } patrow\rangle \Rightarrow \mathrm{FAIL}} \tag{145}$$

$$\frac{r(lab) \vdash pat \Rightarrow VE \qquad \langle r \vdash patrow \Rightarrow VE'/\mathrm{FAIL}\rangle}{r \vdash lab \texttt{ = } pat \ \langle \texttt{ , } patrow\rangle \Rightarrow VE\langle + VE'/\mathrm{FAIL}\rangle} \tag{146}$$

Comments:

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

Patterns

$$\boxed{v \vdash pat \Rightarrow VE/\mathrm{FAIL}}$$

$$\frac{v \vdash atpat \Rightarrow VE/\mathrm{FAIL}}{v \vdash atpat \Rightarrow VE/\mathrm{FAIL}} \tag{147}$$

$$\frac{\begin{array}{c} longcon = strid_1.\cdots.strid_k.con \neq \mathtt{ref} \qquad v = (con, v') \\ v' \vdash atpat \Rightarrow VE/\mathrm{FAIL} \end{array}}{v \vdash longcon\ atpat \Rightarrow VE/\mathrm{FAIL}} \tag{148}$$

$$\frac{longcon = strid_1.\cdots.strid_k.con \neq \mathtt{ref} \qquad v \neq (con, v')}{v \vdash longcon\ atpat \Rightarrow \mathrm{FAIL}} \tag{149}$$

$$\frac{s(a) = v \qquad s, v \vdash atpat \Rightarrow VE/\mathrm{FAIL}, s}{s, a \vdash \mathtt{ref}\ atpat \Rightarrow VE/\mathrm{FAIL}, s} \tag{150}$$

$$\frac{v \vdash pat \Rightarrow VE/\mathrm{FAIL}}{v \vdash var\langle :\ ty\rangle\ \mathtt{as}\ pat \Rightarrow \{var \mapsto v\} + VE/\mathrm{FAIL}} \tag{151}$$

Comments:

(149) Any evaluation resulting in FAIL must do so because rule 141 or rule 149 has been applied.

7 Dynamic Semantics for Modules

7.1 Reduced Syntax

Since signatures 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:

- 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* 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 17. An *interface* $I \in \mathrm{Int}$ represents

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

Figure 17: Compound Semantic Objects

a "view" of a structure. Specifications and signatures will evaluate to interfaces; moreover, during their evaluation, structures (to which a specification or signature may refer via "open") are represented only by their interfaces. To extract an interface from a dynamic environment we define the operation

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

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 signatures 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:

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

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)

$$VE' = VE \downarrow vars, \;\; EE' = EE \downarrow exns.$$

It is important to note that an interface is also a projection of the static value Σ of a signature; it is obtained by omitting the structure names m, type functions θ and type environments TE. Thus in an implementation interfaces would naturally be obtained from the static elaboration; we choose to give separate rules here for obtaining them in the dynamic semantics since we wish to maintain our separation of the static and dynamic semantics, for reasons of presentation.

7.3 Inference Rules

The semantic rules allow sentences of the form

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

to be inferred, where A is either a basis 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 phrases whose evaluation may cause a side-effect or generate an exception packet are of the form *strexp*, *strdec*, *strbind* or *program*.

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

$$\frac{B \vdash strdec \Rightarrow E}{B \vdash \mathtt{struct}\ strdec\ \mathtt{end} \Rightarrow E} \tag{152}$$

$$\frac{B(longstrid) = E}{B \vdash longstrid \Rightarrow E} \tag{153}$$

$$\frac{\begin{array}{c} B(funid) = (strid : I, strexp'\langle : I'\rangle, B') \\ B \vdash strexp \Rightarrow E \qquad B' + \{strid \mapsto E \downarrow I\} \vdash strexp' \Rightarrow E' \end{array}}{B \vdash funid\ (\ strexp\ )\ \Rightarrow E'\langle \downarrow I'\rangle} \tag{154}$$

$$\frac{B \vdash strdec \Rightarrow E \qquad B + E \vdash strexp \Rightarrow E'}{B \vdash \mathtt{let}\ strdec\ \mathtt{in}\ strexp\ \mathtt{end} \Rightarrow E'} \tag{155}$$

Comments:

(154) 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 $\boxed{B \vdash strdec \Rightarrow E}$

$$\frac{E \text{ of } B \vdash dec \Rightarrow E}{B \vdash dec \Rightarrow E} \tag{156}$$

$$\frac{B \vdash strbind \Rightarrow SE}{B \vdash \mathtt{structure}\ strbind \Rightarrow SE \text{ in Env}} \tag{157}$$

$$\frac{B \vdash strdec_1 \Rightarrow E_1 \qquad B + E_1 \vdash strdec_2 \Rightarrow E_2}{B \vdash \mathtt{local}\ strdec_1\ \mathtt{in}\ strdec_2\ \mathtt{end} \Rightarrow E_2} \tag{158}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{159}$$

$$\frac{B \vdash strdec_1 \Rightarrow E_1 \qquad B + E_1 \vdash strdec_2 \Rightarrow E_2}{B \vdash strdec_1 \ \langle ; \rangle \ strdec_2 \Rightarrow E_1 + E_2} \tag{160}$$

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

$$\frac{\begin{array}{c} B \vdash strexp \Rightarrow E \qquad \langle \mathrm{Inter}\, B \vdash sigexp \Rightarrow I \rangle \\ \langle\langle B \vdash strbind \Rightarrow SE \rangle\rangle \end{array}}{\begin{array}{c} B \vdash strid \ \langle \texttt{:}\ sigexp \rangle \ \texttt{=}\ strexp \ \langle\langle \texttt{and}\ strbind \rangle\rangle \Rightarrow \\ \{strid \mapsto E\langle \downarrow I \rangle\} \ \langle\langle +\ SE \rangle\rangle \end{array}} \tag{161}$$

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

$$\frac{IB \vdash spec \Rightarrow I}{IB \vdash \texttt{sig}\ spec\ \texttt{end} \Rightarrow I} \tag{162}$$

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

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

$$\frac{IB \vdash sigbind \Rightarrow G}{IB \vdash \texttt{signature}\ sigbind \Rightarrow G} \tag{164}$$

$$\frac{}{IB \vdash \qquad \Rightarrow \{\}} \tag{165}$$

$$\frac{IB \vdash sigdec_1 \Rightarrow G_1 \qquad IB + G_1 \vdash sigdec_2 \Rightarrow G_2}{IB \vdash sigdec_1 \ \langle ; \rangle \ sigdec_2 \Rightarrow G_1 + G_2} \tag{166}$$

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

$$\frac{IB \vdash sigexp \Rightarrow I \qquad \langle IB \vdash sigbind \Rightarrow G \rangle}{IB \vdash sigid \ \texttt{=}\ sigexp \ \langle \texttt{and}\ sigbind \rangle \Rightarrow \{sigid \mapsto I\} \ \langle +\ G \rangle} \tag{167}$$

Specifications

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

$$\frac{\vdash valdesc \Rightarrow vars}{IB \vdash \texttt{val}\ valdesc \Rightarrow vars \text{ in Int}} \tag{168}$$

$$\frac{\vdash exndesc \Rightarrow exns}{IB \vdash \texttt{exception}\ exndesc \Rightarrow exns \text{ in Int}} \tag{169}$$

$$\frac{IB \vdash strdesc \Rightarrow IE}{IB \vdash \texttt{structure}\ strdesc \Rightarrow IE \text{ in Int}} \tag{170}$$

$$\frac{IB \vdash spec_1 \Rightarrow I_1 \qquad IB + IE \text{ of } I_1 \vdash spec_1 \Rightarrow I_2}{IB \vdash \texttt{local}\ spec_1\ \texttt{in}\ spec_2\ \texttt{end} \Rightarrow I_2} \tag{171}$$

$$\frac{IB(longstrid_1) = I_1 \quad \cdots \quad IB(longstrid_n) = I_n}{IB \vdash \texttt{open}\ longstrid_1 \cdots longstrid_n \Rightarrow I_1 + \cdots + I_n} \tag{172}$$

$$\frac{IB(sigid_1) = I_1 \quad \cdots \quad IB(sigid_n) = I_n}{IB \vdash \texttt{include}\ sigid_1 \cdots sigid_n \Rightarrow I_1 + \cdots + I_n} \tag{173}$$

$$\frac{}{IB \vdash \qquad \Rightarrow \{\} \text{ in Int}} \tag{174}$$

$$\frac{IB \vdash spec_1 \Rightarrow I_1 \qquad IB + IE \text{ of } I_1 \vdash spec_2 \Rightarrow I_2}{IB \vdash spec_1\ \langle ; \rangle\ spec_2 \Rightarrow I_1 + I_2} \tag{175}$$

Comments:

(171),(175) Note that *vars* of I_1 and *exns* of I_1 are not needed for the evaluation of $spec_2$.

Value Descriptions

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

$$\frac{\langle \vdash valdesc \Rightarrow vars \rangle}{\vdash var\ \langle \texttt{and}\ valdesc \rangle \Rightarrow \{var\}\ \langle \cup\ vars \rangle} \tag{176}$$

Exception Descriptions

$$\boxed{\vdash exndesc \Rightarrow exns}$$

$$\frac{\langle \vdash exndesc \Rightarrow exns \rangle}{\vdash exn\ \langle exndesc \rangle \Rightarrow \{exn\}\ \langle \cup\ exns \rangle} \tag{177}$$

Structure Descriptions

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

$$\frac{IB \vdash sigexp \Rightarrow I \qquad \langle IB \vdash strdesc \Rightarrow IE \rangle}{IB \vdash strid : sigexp \ \langle \text{and } strdesc \rangle \Rightarrow \{strid \mapsto I\} \ \langle + \ IE \rangle} \tag{178}$$

Functor Bindings

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

$$\frac{\begin{array}{c}\mathrm{Inter}\,B \vdash sigexp \Rightarrow I \qquad \langle \mathrm{Inter}\,B + \{strid \mapsto I\} \vdash sigexp' \Rightarrow I' \rangle \\ \langle\langle B \vdash funbind \Rightarrow F \rangle\rangle\end{array}}{\begin{array}{c}B \vdash funid \texttt{ (} strid \texttt{ : } sigexp \texttt{) } \langle \texttt{:}\ sigexp' \rangle \texttt{ = } strexp \ \langle\langle \text{and } funbind \rangle\rangle \Rightarrow \\ \{funid \mapsto (strid : I, strexp \langle \texttt{:}\ I' \rangle, B)\} \ \langle\langle + \ F \rangle\rangle\end{array}} \tag{179}$$

Functor Declarations

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

$$\frac{B \vdash funbind \Rightarrow F}{B \vdash \texttt{functor } funbind \Rightarrow F} \tag{180}$$

$$\frac{}{B \vdash \qquad \Rightarrow \{\}} \tag{181}$$

$$\frac{B \vdash fundec_1 \Rightarrow F_1 \qquad B + F_1 \vdash fundec_2 \Rightarrow F_2}{B \vdash fundec_1 \ \langle ; \rangle \ fundec_2 \Rightarrow F_1 + F_2} \tag{182}$$

Programs

$$\boxed{B \vdash program \Rightarrow B'}$$

$$\frac{B \vdash strdec \Rightarrow E}{B \vdash strdec \Rightarrow E \text{ in Basis}} \tag{183}$$

$$\frac{\mathrm{Inter}\,B \vdash sigdec \Rightarrow G}{B \vdash sigdec \Rightarrow G \text{ in Basis}} \tag{184}$$

$$\frac{B \vdash fundec \Rightarrow F}{B \vdash fundec \Rightarrow F \text{ in Basis}} \tag{185}$$

$$\frac{B \vdash program_1 \Rightarrow B_1 \qquad B + B_1 \vdash program_2 \Rightarrow B_2}{B \vdash program_1 \ \langle ; \rangle \ program_2 \Rightarrow B_1 + B_2} \tag{186}$$

A Appendix: Derived Forms

Several derived grammatical forms are provided in the Core; they are presented in Figures 18, 19 and 20. Each derived form is given with its equivalent form. Thus, each row of the tables should be considered as a rewriting rule

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

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 20:

- In the equivalent form for a function-value binding, the variables var_1, $\cdots$, var_n must be chosen not to occur in the derived form. The condition $m, n \geq 1$ applies.

- In the two forms involving `withtype` , the identifiers bound by *datbind* and by *typbind* must be distinct. Then the transformed binding *datbind′* in the equivalent form is obtained from *datbind* by expanding out all the definitions made by *typbind*. More precisely, if *typbind* is

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

  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

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

- The abbreviation of `val it =` *exp* to *exp* is only permitted at top-level, i.e. as a *program* .

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)
`raise` *longexn*	`raise` *longexn* `with ()`	
`case` *exp* `of` *match*	`(fn` *match*`)(`*exp*`)`	
`if` *exp* `then` exp_1 `else` exp_2	`case` *exp* `of true =>` exp_1 `| false =>` exp_2	
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 (_) =>`
	`...`
	`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 18: Derived forms of Expressions

Derived Form	Equivalent Form	

HANDLING RULES *hrule*

$longexn$ `=>` exp	$longexn$ `with (_) =>` exp	

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`	$(n \geq 0)$

LABELLED PATTERNS *patrow*

$id\langle$`:`$ty\rangle$ $\langle$`as` $pat\rangle$ $\langle$`,` $patrow\rangle$	id `=` $id\langle$`:`$ty\rangle$ $\langle$`as` $pat\rangle$ $\langle$`,` $patrow\rangle$	

TYPES *ty*

ty_1 `*` $\cdots$ `*` ty_n	`{1:`ty_1`,` $\cdots$ `,` $\overline{n}$`:`ty_n`}`	$(n \geq 2)$

Figure 19: Derived forms of Handling rules, Patterns and Types

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` $fvalbind\rangle$

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`
exp	`val it =` exp

Figure 20: Derived forms of Function-value Bindings and Declarations

B Appendix: Full Grammar of the Core

In this Appendix, the full Core grammar is given for reference purposes. 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 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.
- L (resp. R) means left (resp. right) association.
- The syntax of types binds more tightly that that of expressions.
- Each iterated construct (e.g. *match*, *handler*, $\cdots$) 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.

The grammatical rules are displayed in Figures 21, 22, 23 and 24.

atexp	::=	⟨`op`⟩*longvar*	value variable
		⟨`op`⟩*longcon*	value 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, $n \geq 1$
		`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 id infexp	infix expression
exp	::=	*infexp*	
		exp `:` *ty*	typed (L)
		exp `andalso` *exp*	conjunction
		exp `orelse` *exp*	disjunction
		exp `handle` *handler*	handle exception
		`raise` *longexn* ⟨`with` *exp*⟩	raise exception
		`if` *exp* `then` exp_1 `else` exp_2	conditional
		`while` exp_1 `do` exp_2	iteration
		`case` *exp* `of` *match*	case analysis
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	
handler	::=	*hrule* ⟨ `||` *handler*⟩	
hrule	::=	*longexn* `with` *match*	
		longexn `=>` *exp*	
		`?` `=>` *exp*	

Figure 21: Grammar: Expressions, Matches and Handlers

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` *exnbind*	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, $n \geq 1$
		exp	expression (top-level only)
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* = *constrs* ⟨and *datbind*⟩	
constrs	::=	⟨op⟩*con* ⟨of *ty*⟩ ⟨ | *constrs*⟩	
exnbind	::=	*exn* ⟨: *ty*⟩⟨= *longexn*⟩ ⟨and *exnbind*⟩	

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 22: Grammar: Declarations and Bindings

atpat	::=	_	wildcard
		⟨op⟩*var*	variable
		longcon	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*	construction
		pat_1 *con* pat_2	infixed construction
		pat : *ty*	typed
		⟨op⟩*var*⟨: *ty*⟩ as *pat*	layered

Figure 23: 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 24: 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

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

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 `instream` and `outstream` admit equality.

The components of E_0 are as follows:

- $SE_0 = \{\}$
- VE_0 is shown in Figures 25 and 26. Note that $\mathrm{Dom}\, VE_0$ contains those identifiers (`true`,`false`,`nil`, `::`) which are basic value constructors, for reasons discussed in Section 4.3.
- TE_0 is shown in Figure 27. Note that the type structures in TE_0 contain the type schemes of all basic value constructors.
- $\mathrm{Dom}\, EE_0 = \mathrm{BasExc}$, the set of basic exceptions listed in Section 6.5. In each case the associated type is `unit` , except that $EE_0(\texttt{io\_failure}) = \texttt{string}$.

NONFIX			INFIX		
`map`	$\mapsto$	$\forall\alpha\beta.\ (\alpha \to \beta) \to$	Precedence 7 :		
		$\alpha\mathtt{list} \to \beta\mathtt{list}$	`/`	$\mapsto$	$\mathtt{real}\ *\ \mathtt{real} \to \mathtt{real}$
`rev`	$\mapsto$	$\forall\alpha.\ \alpha\mathtt{list} \to \alpha\mathtt{list}$	`div`	$\mapsto$	$\mathtt{int}\ *\ \mathtt{int} \to \mathtt{int}$
`not`	$\mapsto$	$\mathtt{bool} \to \mathtt{bool}$	`mod`	$\mapsto$	$\mathtt{int}\ *\ \mathtt{int} \to \mathtt{int}$
`~`	$\mapsto$	$\mathtt{num} \to \mathtt{num}$	`*`	$\mapsto$	$\mathtt{num}\ *\ \mathtt{num} \to \mathtt{num}$
`abs`	$\mapsto$	$\mathtt{num} \to \mathtt{num}$	Precedence 6 :		
`floor`	$\mapsto$	$\mathtt{real} \to \mathtt{int}$	`+`	$\mapsto$	$\mathtt{num}\ *\ \mathtt{num} \to \mathtt{num}$
`real`	$\mapsto$	$\mathtt{int} \to \mathtt{real}$	`-`	$\mapsto$	$\mathtt{num}\ *\ \mathtt{num} \to \mathtt{num}$
`sqrt`	$\mapsto$	$\mathtt{real} \to \mathtt{real}$	`^`	$\mapsto$	$\mathtt{string}\ *\ \mathtt{string} \to \mathtt{string}$
`sin`	$\mapsto$	$\mathtt{real} \to \mathtt{real}$	Precedence 5 :		
`cos`	$\mapsto$	$\mathtt{real} \to \mathtt{real}$	`::`	$\mapsto$	$\forall\alpha.\alpha\ *\ \alpha\mathtt{list} \to \alpha\mathtt{list}$
`arctan`	$\mapsto$	$\mathtt{real} \to \mathtt{real}$	`@`	$\mapsto$	$\forall\alpha.\ \alpha\mathtt{list}\ *\ \alpha\mathtt{list} \to \alpha\mathtt{list}$
`exp`	$\mapsto$	$\mathtt{real} \to \mathtt{real}$	Precedence 4 :		
`ln`	$\mapsto$	$\mathtt{real} \to \mathtt{real}$	`=`	$\mapsto$	$\forall\eta.\ \eta\ *\ \eta \to \mathtt{bool}$
`size`	$\mapsto$	$\mathtt{string} \to \mathtt{int}$	`<>`	$\mapsto$	$\forall\eta.\ \eta\ *\ \eta \to \mathtt{bool}$
`chr`	$\mapsto$	$\mathtt{int} \to \mathtt{string}$	`<`	$\mapsto$	$\mathtt{num}\ *\ \mathtt{num} \to \mathtt{bool}$
`ord`	$\mapsto$	$\mathtt{string} \to \mathtt{int}$	`>`	$\mapsto$	$\mathtt{num}\ *\ \mathtt{num} \to\to \mathtt{bool}$
`explode`	$\mapsto$	$\mathtt{string} \to \mathtt{string\ list}$	`<=`	$\mapsto$	$\mathtt{num}\ *\ \mathtt{num} \to\to \mathtt{bool}$
`implode`	$\mapsto$	$\mathtt{string\ list} \to \mathtt{string}$	`>=`	$\mapsto$	$\mathtt{num}\ *\ \mathtt{num} \to\to \mathtt{bool}$
`!`	$\mapsto$	$\forall\alpha.\ \alpha\mathtt{ref} \to \alpha$	Precedence 3 :		
`ref`	$\mapsto$	$\forall\alpha.\ \alpha \to \alpha\mathtt{ref}$	`:=`	$\mapsto$	$\forall\alpha.\ \alpha\mathtt{ref}\ *\ \alpha \to \mathtt{unit}$
`true`	$\mapsto$	$\mathtt{bool}$	`o`	$\mapsto$	$\forall\alpha\beta\gamma.\ (\beta \to \gamma)\ *\ (\alpha \to \beta) \to$
`false`	$\mapsto$	$\mathtt{bool}$			$(\alpha \to \gamma)$

Notes:

- In this table we have adopted the convention that the type variable η possesses the equality attribute, but that other type variables do not.

- 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. In the case that both types can be inferred for an occurrence of the identifier, an explicit type constraint is needed to determine which type is intended.

- The type schemes associated with pre-defined value constructors or constants are given in Figure 27 which shows the initial static type environment.

Figure 25: Static VE_0, except for Input/Output

$$
\begin{array}{rcl}
\texttt{std\_in} & \mapsto & \texttt{instream} \\
\texttt{open\_in} & \mapsto & \texttt{string} \to \texttt{instream} \\
\texttt{input} & \mapsto & \texttt{instream} * \texttt{int} \to \texttt{string} \\
\texttt{lookahead} & \mapsto & \texttt{instream} \to \texttt{string} \\
\texttt{close\_in} & \mapsto & \texttt{instream} \to \texttt{unit} \\
\texttt{end\_of\_stream} & \mapsto & \texttt{instream} \to \texttt{bool} \\
 & & \\
\texttt{std\_out} & \mapsto & \texttt{outstream} \\
\texttt{open\_out} & \mapsto & \texttt{string} \to \texttt{outstream} \\
\texttt{output} & \mapsto & \texttt{outstream} * \texttt{string} \to \texttt{unit} \\
\texttt{close\_out} & \mapsto & \texttt{outstream} \to \texttt{unit}
\end{array}
$$

Figure 26: Static VE_0 (Input/Output)

$$
\begin{array}{rcll}
\texttt{unit} & \mapsto & \{\ \Lambda().\{\}, & \{\}\ \} \\
\texttt{bool} & \mapsto & \{\ \texttt{bool}, & \{\texttt{true} \mapsto \texttt{bool},\ \texttt{false} \mapsto \texttt{bool}\}\ \} \\
\texttt{int} & \mapsto & \{\ \texttt{int}, & \{i \mapsto \texttt{int}\ ;\ i \text{ an integer constant}\}\ \} \\
\texttt{real} & \mapsto & \{\ \texttt{real}, & \{r \mapsto \texttt{real}\ ;\ r \text{ a real constant}\}\ \} \\
\texttt{string} & \mapsto & \{\ \texttt{string}, & \{s \mapsto \texttt{string}\ ;\ s \text{ a string constant}\}\ \} \\
\texttt{list} & \mapsto & \multicolumn{2}{l}{\{\ \texttt{list}, \{\texttt{nil} \mapsto \forall\alpha.\alpha\texttt{list},\ :: \mapsto \forall\alpha.\alpha * \alpha\texttt{list} \to \alpha\texttt{list}\}\ \}} \\
\texttt{ref} & \mapsto & \{\ \texttt{ref}, & \{\texttt{ref} \mapsto \forall\alpha.\alpha \to \alpha\texttt{ref}\}\ \} \\
\texttt{instream} & \mapsto & \{\ \texttt{instream}, & \{\}\ \} \\
\texttt{outstream} & \mapsto & \{\ \texttt{outstream}, & \{\}\ \}
\end{array}
$$

Figure 27: 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 exceptions BasExc; in fact $E'_0 \;=\; SE'_0, VE'_0, EE'_0$, where:

- $SE'_0 \;=\; \{\}$
- $VE'_0 \;\doteq\; \{id \mapsto id \;;\; id \in \mathrm{BasVal}\} \cup \{\texttt{:=} \mapsto \texttt{:=}\}$
- $EE'_0 \;=\; \{id \mapsto id \;;\; id \in \mathrm{BasExc}\}$

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 initially bound to them. 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 109. Similarly, EE'_0 is the identity function on BasExc, the set of basic exception names, because we have also chosen to denote these exceptions by the exception names initially bound to them. 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 in BasVal. 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.
- abs(n) returns the absolute value of n.
- 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 i'th ASCII character – or the packet [chr,{}] if none exists.

- `ord`(s) returns the ASCII ordinal number of the first character in s, or the packet [ord,{}] if s is empty.

- `explode`(s) returns the list of characters (as single-character lists) 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 such as [+,{}] if the result is 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 order relations `<,>,<=,>=` return boolean values in accord with their usual meanings.

- v_1 `=` v_2 returns the boolean value of $v_1 = v_2$, where the equality of values ($=$) is defined recursively as follows:

  - If v_1, v_2 are constants (including nullary constructors) or addresses, then $v_1 = v_2$ iff v_1 and v_2 are identical.
  - $(con_1, v_1) = (con_2, v_2)$ iff con_1, con_2 are identical and $v_1 = v_2$.
  - $r_1 = r_2$ (for records r_1, r_2) iff $\mathrm{Dom}\, r_1 = \mathrm{Dom}\, r_2$ and, for each $lab \in \mathrm{Dom}\, r_1$, $r_1(lab) = r_2(lab)$.

  The type discipline (in particular, the fact that function types do not admit equality) makes it unnecessary to specify equality in any other cases.

- 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_failure, "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*) is equivalent to `(lookahead(`*is*`)="")` ; 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_failure, "Output stream is closed"]`

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