The Definition of Non-Standard ML (Syntax and Static Semantics)

Claudio Russo
Laboratory for Foundations of Computer Science
Department of Computer Science
University of Edinburgh

based on
The Definition of Standard ML
Revised 1996
by
Robin Milner, Mads Tofte, Robert Harper and Dave MacQueen

Draft of December 15, 2014

Contents

1 Introduction and Disclaimer

This document, an extension of the definition of Standard ML, was never intended for publication. I drafted it during the design and implementation of Moscow ML and its extended Module system. As far as I can recall, the latex sources were derived from a copy of the SML'90 Definition available at the LFCS, manually updated to reflect the SML'97 revision and then extended appropriately. The design only documents the changes to the syntax and static semantics of Standard ML and does not specify the required changes to the dynamic semantics. Features added are higher-order functors (both applicative and generative), first-class modules and recursive modules as well as minor relaxations of the restrictions in Standard ML to make first-class modules more useful. The design aimed for backward compatibility with SML'97. I am content with most of the design but, as it stands, the formalization of recursive modules is more complicated than it needs to be: the additional semantic objects called recursive structures are unnecessary and the system presented in my ICFP' 2001 paper is simpler.

Claudio Russo, Microsoft Research, December 2014.

2 Syntax of the Core

2.1 Reserved Words

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

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

2.2 Special constants

An *integer constant (in decimal notation)* is an optional negation symbol (`~`) followed by a non-empty sequence of decimal digits (`0`-`9`). An *integer constant (in hexadecimal notation)* is an optional negation symbol followed by `0x` followed by a non-empty sequence of hexadecimal digits (`0`-`9a`-`fA`-`F`, where `A`-`F` are alternatives for `a`-`f`, respectively).

A *word constant (in decimal notation)* is `0w` followed by a non-empty sequence of decimal digits. A *word constant (in hexadecimal notation)* is `0wx` followed by a non-empty sequence of hexadecimal digits.

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

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

`\a`	A single character interpreted by the system as alert (ASCII 7)
`\b`	Backspace (ASCII 8)

`\t`	Horizontal Tab (ASCII 9)
`\n`	Linefeed, also known as newline (ASCII 10)
`\v`	Vertical Tab (ASCII 11)
`\f`	Form Feed (ASCII 12)
`\r`	Carriage return (ASCII 13)
`\^`c	The control character c, where c may be any character with number 64–95. The number of `\^`c is 64 less than the number of c.
`\`ddd	The single character with number ddd (3 decimal digits denoting an integer in the ordinal range of the alphabet).
`\u`$xxxx$	The single character with number $xxxx$ (4 hexadecimal digits denoting an integer in the ordinal range of the alphabet).
`\"`	`"`
`\\`	`\`
`\`$f\cdot\cdot f$`\`	This sequence is ignored, where $f\cdot\cdot f$ stands for a sequence of one or more formatting characters.

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

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

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

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

2.3 Comments

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

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

Figure 1: Identifiers

unmatched `(*` should be detected by the compiler.

2.4 Identifiers

The classes of *identifiers* for the Core are shown in Figure 1.

We use *vid*, *tyid* to range over VId, TyId etc. For each class X marked "long" there is a class longX of *long identifiers*; if x ranges over X then *longx* ranges over longX. The syntax of these long identifiers is given by the following:

longx	::=	x	identifier
		longstrid.x	qualified identifier ($n \geq 1$)

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

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

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

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

A type identifier *tyid* may be any alphanumeric identifier starting with a prime; the subclass ETyId of TyId, the *equality* type identifiers, consists of those which start with two or more primes.

The other four classes (VId, TyCon, Lab and StrId) are represented by identifiers not starting with a prime. However, `*` is excluded from TyCon, to avoid confusion with the derived form of tuple type (see Figure ??). The class Lab is extended to include the *numeric* labels `1 2 3` $\cdots$, i.e. any numeral not starting with `0`.

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

1. Immediately before "." – i.e. in a long identifier – or in an `open` declaration, *id* is a structure identifier. The following rules assume that all occurrences of structure identifiers have been removed.

2. At the start of a component in a record type, record pattern or record expression, *id* is a record label.

3. Elsewhere in types *id* is a type constructor.

4. Elsewhere, *id* is a value identifier.

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

2.5 Lexical analysis

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

2.6 Infixed operators

An identifier may be given *infix status* by the `infix` or `infixr` directive, which may occur as a declaration or specification; this status only pertains to its use as a *vid* within the scope (see below) of the directive. (Note that qualified identifiers never have infix status.) If *vid* has infix status, then "exp_1 *vid* exp_2" (resp. "pat_1 *vid* pat_2") may occur – in parentheses if necessary – wherever the application "*vid*`{1=`exp_1`,2=`exp_2`}`" or its derived form "*vid*`(`exp_1`,`exp_2`)`" (resp "*vid*`(`pat_1`,`pat_2`)`") would otherwise occur. On the other hand, an occurrence of any long identifier (qualified or not) prefixed by `op` is treated as non-infixed. The only required use of `op` in the Core is in prefixing a non-infixed occurrence of an identifier *vid* which has infix status; elsewhere in the Core `op`, where permitted, has no effect.[1]. Infix status is cancelled by the `nonfix` directive. We refer to the three directives collectively as *fixity directives*.

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

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

$$\texttt{infixr}\ \langle d \rangle\ \mathit{vid}_1 \cdots \mathit{vid}_n$$

$$\texttt{nonfix}\ \mathit{vid}_1 \cdots \mathit{vid}_n$$

where $\langle d \rangle$ is an optional decimal digit d indicating binding precedence. A higher value of d indicates tighter binding; the default is 0. `infix` and `infixr` dictate left and right associativity respectively. In an expression of the form $\mathit{exp}_1\ \mathit{vid}_1\ \mathit{exp}_2\ \mathit{vid}_2\ \mathit{exp}_3$, where vid_1 and vid_2 are infixed operators with the same precedence, either both must associate to the left or both must associate to the right. For example, suppose that `<<` and `>>` have equal precedence, but associate to the left and right respectively; then

`x << y << z`	parses as	`(x << y) << z`
`x >> y >> z`	parses as	`x >> (y >> z)`
`x << y >> z`	is illegal	
`x >> y << z`	is illegal	

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

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

$$\texttt{let}\ \mathit{dec}\ \texttt{in} \cdots \texttt{end}$$

$$\texttt{local}\ \mathit{dec}\ \texttt{in} \cdots \texttt{end}$$

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 Core semantic rules, since they affect only parsing.

[1]In Modules, `op` is used to resolve the occurrence of long module identifier to either a long structure identifier or a long functor identifier, whenever the interpretation of the original identifier is not determined by the context

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

Figure 2: Core Phrase Classes

2.7 Derived Forms

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

2.8 Grammar

The phrase classes for the Core are shown in Figure 2. We use the variable *atexp* to range over AtExp, etc.

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

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:

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

  (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; this resolves ambiguity in parsing, as explained in Appendix **??**.

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

- The syntax of types binds more tightly than that of expressions.

- Each iterated construct (e.g. *match*, $\cdots$) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. "`fn` *match*", if this occurs within a larger match.

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvid*	value identifier
		`{` ⟨*exprow*⟩ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		[`structure` *modexp* `as` *sigexp*]	structure package
		[`functor` *modexp* `as` *sigexp*]	functor package
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		exp_1 *vid* exp_2	infixed application
		exp `:` *ty*	typed (L)
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	

Figure 3: Grammar: Expressions and Matches

dec	::=	`val` *tyidseq valbind*	value declaration
		`type` *typbind*	type declaration
		`datatype` *datbind*	datatype declaration
		`datatype` *tycon* `=` `datatype` *tyconpath*	datatype replication
		`abstype` *datbind* `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declaration
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \cdots longstrid_n$	open declaration $(n \geq 1)$
		`structure` *strbind*	structure declaration
		`functor` *funbind*	functor declaration
		`signature` *sigbind*	signature declaration
			empty declaration
		dec_1 ⟨;⟩ dec_2	sequential declaration
		`infix` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (L) directive
		`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (R) directive
		`nonfix` $vid_1 \cdots vid_n$	nonfix directive
valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	
		`rec` *valbind*	
typbind	::=	*tyidseq tycon* `=` *ty* ⟨`and` *typbind*⟩	
datbind	::=	*tyidseq tycon* `=` *conbind* ⟨`and` *datbind*⟩	
conbind	::=	⟨`op`⟩*vid* ⟨`of` *ty*⟩ ⟨ $\mid$ *conbind*⟩	
exbind	::=	⟨`op`⟩*vid* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	
		⟨`op`⟩*vid* `=` ⟨`op`⟩*longvid* ⟨`and` *exbind*⟩	

Figure 4: Grammar: Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		⟨`op`⟩*longvid*	variable
		`{` ⟨*patrow*⟩ `}`	record
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow*⟩	pattern row
pat	::=	*atpat*	atomic
		⟨`op`⟩*longvid atpat*	value construction
		pat_1 *vid* pat_2	infixed value construction
		pat `:` *ty*	typed
		⟨`op`⟩*vid*⟨`:` *ty*⟩ `as` *pat*	layered
tyconpath	::=	*longtycon*	long type constructor
		longtycon `where` *strid* = *modexp*	type projection
ty	::=	*tyid*	type identifier
		`{` ⟨*tyrow*⟩ `}`	record type expression
		tyseq tyconpath	type construction
		ty `->` *ty′*	function type expression (R)
		`[` *sigexp* `]`	package type expression
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow*⟩	type-expression row

Figure 5: Grammar: Patterns and Type expressions

2.9 Syntactic Restrictions

- No expression row, pattern row or type row may bind the same *lab* twice.
- No binding *valbind*, *typbind*, *datbind* or *exbind* may bind the same identifier twice; this applies also to value constructors within a *datbind*.
- No *tyidseq* may contain the *tyid* twice.
- For each value binding *pat* `=` *exp* within `rec`, *exp* must be of the form `fn` *match*. or more type expressions. The derived form of function-value binding given in Appendix A, page 60, necessarily obeys this

restriction.

- No *datbind*, *valbind* or *exbind* may bind `true`, `false`, `nil`, `::` or `ref`. No *datbind* or *exbind* may bind `it`.
- No real constant may occur in a pattern.
- In a value declaration `val` *tyidseq* *valbind*, if *valbind* contains another value declaration `val` *tyidseq*$'$ *valbind*$'$ then *tyidseq* and *tyidseq*$'$ must be disjoint. In other words, no type variable may be scoped by two value declarations of which one occurs inside the other. This restriction applies after *tyidseq* and *tyidseq*$'$ have been extended to include implicitly scoped type variables, as explained in Section 4.

3 Syntax of Modules

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

3.1 Reserved Words

The following are the additional reserved words used in Modules.

`eqtype   include   sharing   sig   struct   :>`

3.2 Identifiers

The additional syntax classes for Modules are SigId (signature identifiers), FunId (functor identifiers), and ModId (unresolved identifiers that may be resolved to either structure or functor identifiers) ; they may be either alphanumeric – not starting with a prime – or symbolic. Functor and module identifiers may be long, in the sense of Section 2.4. (Long) module identifiers range of the union of (long) structure identifiers and (long) functor identifiers: the interpretation of a (long) module identifier cannot be determined grammatically, but is resolved during elaboration. Otherwise, the class of each identifier occurrence is determined by the grammatical rules which follow. Henceforth, therefore, we consider all identifier classes (excluding ModId and longModId) 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 declaration *dec* in any of the phrases

`let` *dec* `in` ⋯ `end`

`struct` *dec* `end`

`sig` *spec* `end`

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

AtModExp	atomic module expressions
ModExp	module expressions
StrBind	structure bindings
FunBind	functor bindings
SigBind	signature bindings
SigExp	signature expressions
Spec	specifications
ValDesc	value descriptions
TypDesc	type descriptions
DatDesc	datatype descriptions
ConDesc	constructor descriptions
ExDesc	exception descriptions
StrDesc	structure descriptions
FunDesc	functor descriptions

Figure 6: Modules Phrase Classes

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

Fixity directives (but not `op`) are omitted from the Modules semantic rules, since they affect only parsing.

3.4 Grammar for Modules

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

3.5 Syntactic Restrictions

- No binding *strbind*, *funbind*, or *sigbind* may bind the same identifier twice.

atmodexp	::=	`struct` *dec* `end`	basic
		⟨`op`⟩*longmodid*	module identifier
		`let` *dec* `in` *modexp* `end`	local declaration
		`(` *modexp* `)`	
modexp	::=	*atmodexp*	atomic
		modexp atmodexp	functor application
		modexp `:` *sigexp*	transparent constraint
		modexp `:>` *sigexp*	opaque constraint
		`functor (`*modid*`:`*sigexp*`) =>` *modexp*	generative functor
		`functor` *modid*`:`*sigexp* `=>` *modexp*	applicative functor
		`rec (`*strid*`:`*sigexp*`)` *modexp*	recursive structure
strbind	::=	*strid* `=` *modexp* ⟨`and` *strbind*⟩	structure binding
		strid `as` *sigexp* `=` *exp* ⟨`and` *strbind*⟩	package binding
funbind	::=	*funid* `=` *modexp* ⟨`and` *funbind*⟩	functor binding
		funid `as` *sigexp* `=` *exp* ⟨`and` *funbind*⟩	package binding
sigbind	::=	*sigid* `=` *sigexp* ⟨`and` *sigbind*⟩	
sigexp	::=	`sig` *spec* `end`	basic
		sigid	signature identifier
		sigexp `where type` *tyidseq longtycon* `=` *ty*	type realisation
		`functor (`*modid* `:` $sigexp_1$`) ->` $sigexp_2$	opaque functor signature
		`functor` *modid* `:` $sigexp_1$ `->` $sigexp_2$	transparent functor signature
		`rec (`*strid*`:`*sigexp*`)` *sigexp*	recursive structure signature

Figure 7: Grammar: Structure and Signature Expressions

- No description *valdesc*, *typdesc*, *datdesc*, *exdesc* or *strdesc* or *fundesc* may describe the same identifier twice; this applies also to value constructors within a *datdesc*.
- No *tyvarseq* may contain the same *tyvar* twice.
- No *datdesc*, *valdesc* or *exdesc* may describe `true`, `false`, `nil`, `::` or `ref`. No *datdesc* or *exdesc* may describe `it`.

spec	::=	`val` *tyidseq valdesc*	value
		`type` *typdesc*	type
		`eqtype` *typdesc*	eqtype
		`datatype` *datdesc*	datatype
		`datatype` *tycon* `=` `datatype` *tyconpath*	replication
		`exception` *exdesc*	exception
		`structure` *strdesc*	structure
		`functor` *fundesc*	functor
		`signature` *sigbind*	signature
		`include` *sigexp*	include
			empty
		$spec_1$ ⟨`;`⟩ $spec_2$	sequential
		spec `sharing type`	sharing
		$longtycon_1$ `=` $\cdots$ `=` $longtycon_n$	$(n \geq 2)$
		`infix` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (L) directive
		`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (R) directive
		`nonfix` $vid_1 \cdots vid_n$	nonfix directive
valdesc	::=	*vid* `:` *ty* ⟨`and` *valdesc*⟩	
typdesc	::=	*tyidseq tycon* ⟨`and` *typdesc*⟩	
datdesc	::=	*tyidseq tycon* `=` *condesc* ⟨`and` *datdesc*⟩	
condesc	::=	*vid* ⟨`of` *ty*⟩ ⟨ `|` *condesc*⟩	
exdesc	::=	*vid* ⟨`of` *ty*⟩ ⟨`and` *exdesc*⟩	
strdesc	::=	*strid* `:` *sigexp* ⟨`and` *strdesc*⟩	
fundesc	::=	*funid* `:` *sigexp* ⟨`and` *fundesc*⟩	

Figure 8: Grammar: Specifications

4 Scope of Explicit Type Identifiers

In the Core language, a type or datatype binding can explicitly introduce type identifiers whose scope is that binding. Similary, in Modules, a type or datatype description can explicitly introduce type identifiers whose scope is that description. Moreover, in a Core value declaration `val` *tyidseq valbind*, the sequence *tyidseq* binds type identifiers: a type identifier occurs free in `val` *tyidseq valbind* iff it occurs free in *valbind* and is not in the sequence *tyidseq*. Similarly, in a Modules value specification `val` *tyidseq valdesc*, the sequence *tyidseq* binds type identifiers: a type identifier occurs free in `val` *tyidseq valdesc* iff it occurs free in *valdesc* and is not in the sequence *tyidseq*. However, explicit binding of type identifiers at `val` is optional, so we still have to account for the scope of any type indentifiers that occur free in type expressions.

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

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

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

Henceforth, we assume that for every value declaration or specification `val` *tyidseq*$\cdots$ occurring in the program, every explicit type identifier implicitly scoped at the `val` has been added to *tyidseq*. Thus for example, in the two declarations

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

the type identifier `'a` is scoped differently; they become respectively

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

Then, according to the inference rules in Section 5.10 the first example can be elaborated, but the second cannot since `'a` is bound at the outer value declaration leaving no possibility of two different instantiations of the type of `id` in the application `id id`.

5 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.

5.1 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 variables, type constructor names and identifier status descriptors. Type variables are the semantic counterparts of type identifiers and range over types. Type constructor names range over the values taken by type constructors; we shall usually refer to them briefly as type names, but they are to be clearly distinguished from type variables and type constructors. The simple object classes, and the variables ranging over them, are shown in Figure 9.

α or *tyvar*	$\in$	TyVar	type variables
t or u	$\in$	TyName	type names
is	$\in$	IdStatus= $\{\texttt{c}, \texttt{e}, \texttt{v}\}$	identifier status descriptors

Figure 9: Simple Semantic Objects

Each $\alpha \in$ TyVar possesses a boolean *equality* attribute, which determines whether or not it *admits equality*. Each $\mathrm{t} \in$ TyName has a kind $K \in$ Kind (defined in Figure 10). We denote the class of type names with kind K by TyName^K, letting t^K range over elements of TyName^K. A type name t *has arity* k, if, and only if, it has kind k or $k^=$. A type name t *admits equality*, or is an *equality type name*, if, and only if, it has kind $k^=$.

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

5.2 Compound Objects

When A and B are sets $\mathrm{Fin}\,A$ denotes the set of finite subsets of A, and $A \stackrel{\mathrm{fin}}{\to} B$ 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 Figures 10 and 11. We take $\cup$ to mean disjoint union over semantic object classes. We also understand all the defined object classes to be disjoint.

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

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

5.3 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*".

When a tuple contains a finite map we shall "apply" the tuple to an argument, relying on the syntactic class of the argument to determine the relevant function. For instance $C(tycon)$ means $(TE \text{ of } C)tycon$.

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

$$
\begin{array}{rcl}
k & \in & \mathrm{Arity} = \{k; k \geq 0\} \\
K \text{ or } k \text{ or } k^{=} \text{ or } K \rightarrow K' & \in & \mathrm{Kind} = \mathrm{Arity} \cup \mathrm{Arity} \cup (\mathrm{Kind} \times \mathrm{Kind}) \\
\tau & \in & \mathrm{Type} = \mathrm{TyVar} \cup \mathrm{RecType} \cup \mathrm{FunType} \cup \\
 & & \qquad \mathrm{ConsType} \cup \mathrm{PackType} \\
(\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 \in \mathrm{Arity}} \mathrm{ConsType}^{k} \\
\tau^{(k)}\ \vartheta^{k} & \in & \mathrm{ConsType}^{k} = \mathrm{Type}^{(k)} \times \mathrm{TypeApp}^{k} \\
[X] & \in & \mathrm{PackType} = \mathrm{ExMod} \\
\theta^{k} \text{ or } \Lambda\alpha^{(k)}.\tau \text{ or } \vartheta^{k} & \in & \mathrm{TypeFcn}^{k} = \\
 & & \qquad (\mathrm{TyVar}^{(k)} \times \mathrm{Type}) \cup \mathrm{TypeApp}^{k} \\
\theta^{k^{=}} & \in & \mathrm{TypeFcn}^{k^{=}} = \\
 & & \qquad \{\theta^{k} \in \mathrm{TypeFcn}^{k}\ ;\ \theta^{k} \textit{admits equality}\} \\
\theta^{K \rightarrow K'} \text{ or } \Lambda \mathrm{t}^{K}.\theta^{K'} \text{ or } \vartheta^{K \rightarrow K'} & \in & \mathrm{TypeFcn}^{K \rightarrow K'} = \\
 & & \qquad (\mathrm{TyName}^{K} \times \mathrm{TypeFcn}^{K'}) \cup \\
 & & \qquad \mathrm{TypeApp}^{K \rightarrow K'} \\
\vartheta^{K} \text{ or } \mathrm{t}^{K} \text{ or } \vartheta^{K' \rightarrow K}\ \theta^{K'} & \in & \mathrm{TypeApp}^{K} = \\
 & & \qquad \mathrm{TyName}^{K} \cup \\
 & & \qquad (\cup_{K' \in \mathrm{Kind}} (\mathrm{TypeApp}^{K' \rightarrow K} \times \mathrm{TypeFcn}^{K'})) \\
\sigma \text{ or } \forall\alpha^{(k)}.\tau & \in & \mathrm{TypeScheme} = \cup_{k \geq 0} \mathrm{TyVar}^{(k)} \times \mathrm{Type}
\end{array}
$$

Figure 10: Compound Semantic Objects

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 $E + (\{\}, \{\}, \{\}, \{\}, VE)$.

5.4 Types, Type Applications 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;

$$
\begin{array}{rcl}
(\theta^k, VE) & \in & \mathrm{TyStr} = (\cup_{k \in \mathrm{Arity}} \mathrm{TypeFcn}^k) \times \mathrm{ValEnv} \\
TE & \in & \mathrm{TyEnv} = \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{TyStr} \\
VE & \in & \mathrm{ValEnv} = \mathrm{VId} \xrightarrow{\mathrm{fin}} \mathrm{TypeScheme} \times \mathrm{IdStatus} \\
E \text{ or } (GE, FE, SE, TE, VE) & \in & \mathrm{Env} = \mathrm{SigEnv} \times \mathrm{FunEnv} \times \\
 & & \quad \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
T & \in & \mathrm{TyNameSet} = \mathrm{Fin}(\mathrm{TyName}) \\
IE & \in & \mathrm{IdEnv} = \mathrm{TyId} \xrightarrow{\mathrm{fin}} \mathrm{Type} \\
C \textit{ or } (IE, E) & \in & \mathrm{Context} = \mathrm{IdEnv} \times \mathrm{Env}
\end{array}
$$

Figure 11: Compound Semantic Objects (continued)

- $\{lab_1 \mapsto \tau_1,\ \cdots,\ lab_n \mapsto \tau_n\}$, where each τ_i admits equality;
- $\tau^{(k)}\ \vartheta$, where $\vartheta\ \in \mathrm{TypeApp}^{k^=}$ and all members of $\tau^{(k)}$ admit equality;
- (τ')`ref`.

(Note that if τ is a package type $[X]$ then it *does not* admit equality.)

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

- $\Lambda\alpha^{(k)}.\tau$, where, when the type variables $\alpha^{(k)}$ are chosen to admit equality, then τ also admits equality;
- ϑ, where $\vartheta\ \in \mathrm{TypeApp}^{k^=}$.

The bound variables of a type function $\theta = \Lambda\alpha^{(k)}.\tau$ must be distinct. The type function has the arity k as its kind. It may also have kind $k^=$, provided it admits equality. A type function $\theta = \Lambda \mathrm{t}^K.\theta'$ has kind $K \to K'$, provided θ' has kind K'.

Two type functions are considered equal if they have the same kind and differ only in their choice of bound variables or type names. In particular, the equality attribute has no significance in a bound type variable of a type function; for example, $\Lambda\alpha.\alpha \to \alpha$ and $\Lambda\beta.\beta \to \beta$ are equal type functions even if α admits equality but β does not.

If the type application ϑ has kind k then we identify the type function $\theta = \vartheta$ with the type function $\Lambda\alpha^{(k)}.\alpha^{(k)}\ \vartheta$ (provided $(\mathrm{tyvars}\,\alpha^{(k)}) \cap (\mathrm{tyvars}\,\vartheta) =$

$\emptyset$) (eta-conversion). If the type application ϑ has kind $K \to K'$ then we identify the type function $\theta = \vartheta$ with the type function $\Lambda \mathrm{t}^K.\vartheta\ \mathrm{t}$ (provided $\mathrm{t} \notin \mathrm{tynames}\,\vartheta$) (eta-conversion).

For convenience, when t has arity k, we shall write the type name t to mean the type function $\Lambda\alpha^{(k)}.\alpha^{(k)}\ \mathrm{t}$.

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

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

(assuming $\mathrm{t} \notin \mathrm{tynames}\,\tau^{(k)}$); and

$$(\mathrm{t}\ \theta)\{\Lambda \mathrm{u}.\theta'/\mathrm{t}\} = \theta'\{\theta/\mathrm{u}\}.$$

(assuming $\mathrm{t} \notin \mathrm{tynames}\,\theta$).

5.5 Type Schemes

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

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

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

5.6 Non-expansive Expressions

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

$$
\begin{array}{rcl}
\textit{nexp} & ::= & \textit{scon} \\
 & & \langle \texttt{op} \rangle \textit{longvid} \\
 & & \texttt{\{}\langle \textit{nexprow} \rangle\texttt{\}} \\
 & & \texttt{(}\textit{nexp}\texttt{)} \\
 & & \textit{conexp}\ \textit{nexp} \\
 & & \textit{nexp}\,\texttt{:}\,\textit{ty} \\
 & & \texttt{fn}\ \textit{match} \\
 & & \texttt{[structure}\ \textit{nmodexp}\ \texttt{as}\ \textit{sigexp}\texttt{]} \\
 & & \texttt{[functor}\ \textit{nmodexp}\ \texttt{as}\ \textit{sigexp}\texttt{]} \\
\textit{nexprow} & ::= & \textit{lab}\ \texttt{=}\ \textit{nexp}\ \langle\ \texttt{,}\textit{nexprow}\rangle \\
\textit{conexp} & ::= & \texttt{(}\textit{conexp}\langle\texttt{:}\,\textit{ty}\rangle\texttt{)} \\
 & & \langle \texttt{op} \rangle \textit{longvid} \\
\textit{nmodexp} & ::= & \langle \texttt{op} \rangle \textit{longmodid} \\
 & & \texttt{(}\ \textit{nmodexp}\ \texttt{)} \\
 & & \textit{nmodexp}\ \texttt{:}\ \textit{sigexp} \\
 & & \textit{nmodexp}\ \texttt{:>}\ \textit{sigexp} \\
 & & \texttt{functor (}\textit{modid}\texttt{:}\,\textit{sigexp}\texttt{) =>}\ \textit{modexp} \\
 & & \texttt{functor}\ \textit{modid}\texttt{:}\,\textit{sigexp}\ \texttt{=>}\ \textit{modexp} \\
 & & \texttt{rec (}\textit{strid}\texttt{:}\,\textit{sigexp}\texttt{)}\,\textit{nmodexp}
\end{array}
$$

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

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

5.7 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 value environment VE contains only types (rather than arbitrary type schemes) we set

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

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

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

5.8 Existential and Parameterised Objects

When A is a set of semantic objects, the set $\mathrm{Ex}(A)$ of *existentially quantified objects in* A and the set $\mathrm{Par}(A)$ of *parameterised objects in* A are defined as follows:

$$\begin{array}{rcl} \exists T.a & \in & \mathrm{Ex}(A) = \mathrm{TyNameSet} \times A \\ \Lambda T.a & \in & \mathrm{Par}(A) = \mathrm{TyNameSet} \times A \end{array}$$

(where a ranges over elements of A).

The prefixes $\exists T.\_$ and $\Lambda T.\_$ are binding constructs. Two objects in $\mathrm{Ex}(A)$ ($\mathrm{Par}(A)$) are considered equal if they are equivalent up to a kind preserving renaming of their bound types names.

5.9 Type Structures and Type Environments

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

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

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

For any TE of the form

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

where no VE_i is the empty map, and for any E we define $\mathrm{Abs}(TE, E)$ to be the environment obtained from E and TE as follows. First, let $\mathrm{Abs}(TE)$ be the type environment $\{tycon_i \mapsto (t_i, \{\})\ ;\ 1 \leq i \leq k\}$ in which all value environments VE_i have been replaced by the empty map. Let $T' = \{t'_1, \cdots, t'_k\}$ be a set of new distinct type names, none of which admit equality, and where t'_i has the same arity as t_i for $1 \leq i \leq k$. Then $\mathrm{Abs}(TE, E) = \exists T'.E'$, where E' is the result of simultaneously substituting t'_i for t_i, $1 \leq i \leq k$, throughout $\mathrm{Abs}(TE) + E$, i.e. $E' = (\mathrm{Abs}(TE) + E)\{t'^{(k)}/t^{(k)}\}$ (The effect of the latter substitution is to ensure that the use of equality on an `abstype` is restricted to the `with` part.)

5.10 Inference Rules

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

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

where A is typically a context, *phrase* is a phrase of the Core, and A' is a semantic object – typically a type or an existentially quantified environment. It may be pronounced "*phrase* elaborates to A' in (context or environment) A". Some rules have extra hypotheses not of this form; they are called *side conditions*. In the presentation of the rules, phrases within single angle brackets $\langle\ \rangle$ are called *first options*, and those within double angle brackets $\langle\langle\ \rangle\rangle$ are called *second options*. To reduce the number of rules, we have adopted the following convention:

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

Long Value Identifiers $\boxed{C \vdash longvid \Rightarrow (\sigma, is)}$

$$\frac{vid \in \mathrm{Dom}\, C}{C \vdash vid \Rightarrow C(vid)} \tag{1}$$

$$\frac{C \vdash longstrid \Rightarrow RS \quad S = S \text{ of } RS \quad vid \in \mathrm{Dom}\, S}{C \vdash longstrid.vid \Rightarrow S(vid)} \tag{2}$$

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

$$\frac{}{C \vdash scon \Rightarrow \mathrm{type}(scon)} \tag{3}$$

$$\frac{C \vdash longvid \Rightarrow (\sigma, is) \quad \sigma \succ \tau}{C \vdash longvid \Rightarrow \tau} \tag{4}$$

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

$$\frac{\begin{array}{c} C \vdash dec \Rightarrow \exists T.E \quad T \cap \mathrm{tynames}\, C = \emptyset \\ C + E \vdash exp \Rightarrow \tau \quad T \cap \mathrm{tynames}\, \tau = \emptyset \end{array}}{C \vdash \texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end} \Rightarrow \tau} \tag{6}$$

$$\frac{\begin{array}{l} C \vdash^{\mathsf{s}} modexp \Rightarrow \exists T.RS \\ C \vdash sigexp \Rightarrow \Lambda T'.RS' \\ T \cap \mathrm{tynames}(\Lambda T'.RS') = \emptyset \\ \Lambda T'.RS' \geq RS'' \prec RS \end{array}}{C \vdash \texttt{[structure}\ modexp\ \texttt{as}\ sigexp\texttt{]} \Rightarrow [\exists T'.RS']} \tag{7}$$

$$\frac{\begin{array}{l} C \vdash^{\mathsf{f}} modexp \Rightarrow \exists T.F \\ C \vdash sigexp \Rightarrow \Lambda T'.F' \\ T \cap \mathrm{tynames}(\Lambda T'.F') = \emptyset \\ \Lambda T'.F' \geq F'' \prec F \end{array}}{C \vdash \texttt{[functor}\ modexp\ \texttt{as}\ sigexp\texttt{]} \Rightarrow [\exists T'.F']} \tag{8}$$

$$\frac{C \vdash exp \Rightarrow \tau}{C \vdash \texttt{(}\ exp\ \texttt{)} \Rightarrow \tau} \tag{9}$$

Comments:

(4) The instantiation of type schemes allows different occurrences of a single *longvid* to assume different types.

(6) The first side condition (that also occurs elsewhere in the rules) ensures that type names generated by the first sub-phrase are distinct from type names already appearing in the context. The second side condition prevents these type names from escaping outside the local declaration.

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

$$\frac{C \vdash exp \Rightarrow \tau \qquad \langle C \vdash exprow \Rightarrow \varrho \rangle}{C \vdash lab\ \texttt{=}\ exp\ \langle\ \texttt{,}\ exprow \rangle \Rightarrow \{lab \mapsto \tau\}\langle + \varrho \rangle} \tag{10}$$

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

$$\frac{C \vdash atexp \Rightarrow \tau}{C \vdash atexp \Rightarrow \tau} \tag{11}$$

$$\frac{C \vdash exp \Rightarrow \tau' \to \tau \qquad C \vdash atexp \Rightarrow \tau'}{C \vdash exp\ atexp \Rightarrow \tau} \tag{12}$$

$$\frac{C \vdash exp \Rightarrow \tau \qquad C \vdash ty \Rightarrow \tau}{C \vdash exp\ \texttt{:}\ ty \Rightarrow \tau} \tag{13}$$

$$\frac{C \vdash exp \Rightarrow \tau \qquad C \vdash match \Rightarrow \texttt{exn} \to \tau}{C \vdash exp\ \texttt{handle}\ match \Rightarrow \tau} \tag{14}$$

$$\frac{C \vdash exp \Rightarrow \texttt{exn}}{C \vdash \texttt{raise}\ exp \Rightarrow \tau} \tag{15}$$

$$\frac{C \vdash match \Rightarrow \tau}{C \vdash \texttt{fn}\ match \Rightarrow \tau} \tag{16}$$

Comments:

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

(13) Here τ is determined by C and *ty*. Notice that type variables in *ty* cannot be instantiated in obtaining τ; thus the expression `1:'a` will not elaborate successfully, nor will the expression `(fn x=>x):'a->'b`. The effect of type variables in an explicitly typed expression is to indicate exactly the degree of polymorphism present in the expression.

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

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

$$\frac{C \vdash \mathit{mrule} \Rightarrow \tau \qquad \langle C \vdash \mathit{match} \Rightarrow \tau \rangle}{C \vdash \mathit{mrule}\ \langle\ \texttt{|}\ \mathit{match}\rangle \Rightarrow \tau} \tag{17}$$

Match Rules $\boxed{C \vdash \mathit{mrule} \Rightarrow \tau}$

$$\frac{C \vdash \mathit{pat} \Rightarrow (\mathit{VE}, \tau) \qquad C + \mathit{VE} \vdash \mathit{exp} \Rightarrow \tau'}{C \vdash \mathit{pat}\ \texttt{=>}\ \mathit{exp}\ \Rightarrow \tau \to \tau'} \tag{18}$$

Comment: This rule allows new free type variables to enter the context. These new type variables will be chosen, in effect, during the elaboration of *pat* (i.e., in the inference of the first hypothesis). In particular, their choice may have to be made to agree with type variables present in any explicit type expression occurring within *exp* (see rule 13).

Declarations $\boxed{C \vdash \mathit{dec} \Rightarrow \exists T.E}$

$$\frac{\begin{array}{ll} C \vdash \mathit{tyidseq} \Rightarrow (\alpha_1, \cdots, \alpha_k), \mathit{IE} & C + \mathit{IE} \vdash \mathit{valbind} \Rightarrow \mathit{VE} \\ \mathit{VE}' = \mathrm{Clos}_{C,\mathit{valbind}}\mathit{VE} & \{\alpha_1, \cdots, \alpha_k\} \cap \mathrm{tyvars}\,\mathit{VE}' = \emptyset \end{array}}{C \vdash \texttt{val}\ \mathit{tyidseq}\ \mathit{valbind} \Rightarrow \exists\emptyset.\mathit{VE}' \text{ in Env}} \tag{19}$$

$$\frac{C \vdash \mathit{typbind} \Rightarrow \mathit{TE}}{C \vdash \texttt{type}\ \mathit{typbind} \Rightarrow \exists\emptyset.\mathit{TE} \text{ in Env}} \tag{20}$$

$$\frac{\begin{array}{l} C + \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \\ T = \{\mathrm{t}; (\mathrm{t}, \mathit{VE}') \in \mathrm{Ran}\,\mathit{TE}\} \\ T \cap \mathrm{tynames}\,C = \emptyset \\ \mathit{TE} \text{ maximises equality} \end{array}}{C \vdash \texttt{datatype}\ \mathit{datbind} \Rightarrow \exists T.(\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{21}$$

$$\frac{\begin{array}{c} C \vdash \mathit{tyconpath} \Rightarrow (\theta, \mathit{VE}) \\ \mathit{TE} = \{\mathit{tycon} \mapsto (\theta, \mathit{VE})\} \end{array}}{\begin{array}{c} C \vdash \texttt{datatype}\ \mathit{tycon}\ \texttt{=}\ \texttt{datatype}\ \mathit{tyconpath} \Rightarrow \\ \exists\emptyset.(\mathit{VE}, \mathit{TE})\ \text{in Env} \end{array}} \tag{22}$$

$$\frac{\begin{array}{ll} C + \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} & T = \{\mathrm{t}; (\mathrm{t}, \mathit{VE}) \in \operatorname{Ran} \mathit{TE}\} \\ \mathit{TE}\ \text{maximises equality} & T \cap \operatorname{tynames} C = \emptyset \\ C + (\mathit{VE}, \mathit{TE}) \vdash \mathit{dec} \Rightarrow \exists T'.E & T' \cap T = \emptyset \\ \operatorname{Abs}(\mathit{TE}, E) = \exists T''.E' & T'' \cap T' = \emptyset \end{array}}{C \vdash \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\ \texttt{end} \Rightarrow \exists T' \cup T''.\operatorname{Abs}(\mathit{TE}, E)} \tag{23}$$

$$\frac{C \vdash \mathit{exbind} \Rightarrow \mathit{VE}}{C \vdash \texttt{exception}\ \mathit{exbind} \Rightarrow \exists\emptyset.\mathit{VE}\ \text{in Env}} \tag{24}$$

$$\frac{\begin{array}{ll} C \vdash \mathit{dec}_1 \Rightarrow \exists T_1.E_1 & T_1 \cap \operatorname{tynames} C = \emptyset \\ C + E_1 \vdash \mathit{dec}_2 \Rightarrow \exists T_2.E_2 & T_2 \cap T_1 = \emptyset \end{array}}{C \vdash \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \Rightarrow \exists T_1 \cup T_2.E_2} \tag{25}$$

$$\frac{\begin{array}{c} C \vdash \mathit{longstrid}_1 \Rightarrow \mathit{RS}_1 \quad S_1 = S\ \text{of}\ \mathit{RS}_1 \\ \vdots \\ C \vdash \mathit{longstrid}_n \Rightarrow \mathit{RS}_n \quad S_n = S\ \text{of}\ \mathit{RS}_n \end{array}}{C \vdash \texttt{open}\ \mathit{longstrid}_1 \cdots \mathit{longstrid}_n \Rightarrow \exists\emptyset.(S_1 + \cdots + S_n)\ \text{in Env}} \tag{26}$$

$$\frac{C \vdash \mathit{strbind} \Rightarrow \exists T.\mathit{SE}}{C \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow \exists T.\mathit{SE}\ \text{in Env}} \tag{27}$$

$$\frac{C \vdash \mathit{funbind} \Rightarrow \exists T.\mathit{FE}}{C \vdash \texttt{functor}\ \mathit{funbind} \Rightarrow \exists T.\mathit{FE}\ \text{in Env}} \tag{28}$$

$$\frac{C \vdash \mathit{sigbind} \Rightarrow \mathit{GE}}{C \vdash \texttt{signature}\ \mathit{sigbind} \Rightarrow \exists\emptyset.\mathit{GE}\ \text{in Env}} \tag{29}$$

$$\frac{}{C \vdash \qquad \Rightarrow \exists\emptyset.\{\}\ \text{in Env}} \tag{30}$$

$$\frac{\begin{array}{ll} C \vdash \mathit{dec}_1 \Rightarrow \exists T_1.E_1 & T_1 \cap \operatorname{tynames} C = \emptyset \\ C + E_1 \vdash \mathit{dec}_2 \Rightarrow \exists T_2.E_2 & T_2 \cap (T_1 \cup \operatorname{tynames} E_1) = \emptyset \end{array}}{C \vdash \mathit{dec}_1\ \langle ; \rangle\ \mathit{dec}_2 \Rightarrow \exists T_1 \cup T_2.E_1 + E_2} \tag{31}$$

Comments:

(19) Here *VE* will contain types rather than general type schemes. The closure of *VE* allows value identifiers to be used polymorphically, via rule 4.

The side-condition on $\{\alpha_1, \cdots, \alpha_k\}$ ensures that the type variables bound to *tyidseq* are bound by the closure operation, if they occur in the range of *VE*.

On the other hand, if the phrase `val` *tyidseq valbind* occurs inside some larger value binding `val` $tyidseq_0$ $valbind_0$ then no type variable α bound to a type identifier listed in $tyidseq_0$ will become bound by the $\mathrm{Clos}_{C,valbind}(VE)$ operation; for α must be in *IE* of C and hence excluded from closure by the definition of the closure operation (Section 5.7, page 26) since $\mathrm{tyvars}(IE \text{ of } C) \subseteq \mathrm{tyvars}\, C$.

(21),(23) The side conditions express that the elaboration of each datatype binding generates new type names and that as many of these new names as possible admit equality. Adding *TE* to the context on the left of the $\vdash$ captures the recursive nature of the binding.

(22) Note that no new type name is generated (i.e., datatype replication is not generative).

(23) The Abs operation was defined in Section 5.9, page 27.

(24) No closure operation is used here, as this would make the type system unsound. Example: `exception E of 'a; val it = (raise E 5) handle E f => f(2)` .

Value Bindings

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

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C \vdash exp \Rightarrow \tau \qquad \langle C \vdash valbind \Rightarrow VE' \rangle}{C \vdash pat \; \texttt{=} \; exp \; \langle \texttt{and} \; valbind \rangle \Rightarrow VE \; \langle + \; VE' \rangle} \tag{32}$$

$$\frac{C + VE \vdash valbind \Rightarrow VE}{C \vdash \texttt{rec} \; valbind \Rightarrow VE} \tag{33}$$

Comments:

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

(33) Modifying C by VE on the left captures the recursive nature of the binding. From rule 32 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. Also note that C + VE may overwrite identifier status. For example, the program `datatype t = f; val rec f = fn x => x;` is legal.

Type Bindings

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

$$\frac{C \vdash \mathit{tyidseq} \Rightarrow (\alpha^{(k)}, \mathit{IE}) \quad C + \mathit{IE} \vdash \mathit{ty} \Rightarrow \tau \quad \langle C \vdash \mathit{typbind} \Rightarrow \mathit{TE}\rangle}{\begin{array}{c} C \vdash \mathit{tyidseq}\ \mathit{tycon} \texttt{ = } \mathit{ty}\ \langle\texttt{and}\ \mathit{typbind}\rangle \Rightarrow \\ \{\mathit{tycon} \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\}\ \langle + \mathit{TE}\rangle \end{array}} \tag{34}$$

Comment: The syntactic restrictions ensure that the type function $\Lambda\alpha^{(k)}.\tau$ satisfies the well-formedness constraints of Section 5.4 and they ensure $\mathit{tycon} \notin \mathrm{Dom}\,\mathit{TE}$.

Data Type Bindings

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

$$\frac{\begin{array}{c} C \vdash \mathit{tyidseq} \Rightarrow (\alpha^{(k)}, \mathit{IE}) \\ C + \mathit{IE}, \alpha^{(k)}\ \mathrm{t} \vdash \mathit{conbind} \Rightarrow \mathit{VE} \\ \langle C \vdash \mathit{datbind} \Rightarrow \mathit{VE}', \mathit{TE}' \qquad \forall(\mathrm{t}', \mathit{VE}'') \in \mathrm{Ran}\,\mathit{TE}, \mathrm{t} \neq \mathrm{t}'\rangle \end{array}}{\begin{array}{c} C \vdash \mathit{tyidseq}\ \mathit{tycon} \texttt{ = } \mathit{conbind}\ \langle\texttt{and}\ \mathit{datbind}\rangle \Rightarrow \\ \mathrm{Clos}_C\mathit{VE}\langle + \mathit{VE}'\rangle,\ \{\mathit{tycon} \mapsto (\mathrm{t}, \mathrm{Clos}_C\mathit{VE})\}\ \langle + \mathit{TE}'\rangle \end{array}} \tag{35}$$

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

Constructor Bindings

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

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau'\rangle \qquad \langle\langle C, \tau \vdash \mathit{conbind} \Rightarrow \mathit{VE}\rangle\rangle}{\begin{array}{c} C, \tau \vdash \mathit{vid}\ \langle\texttt{of}\ \mathit{ty}\rangle\ \langle\langle\ \texttt{|}\ \mathit{conbind}\rangle\rangle \Rightarrow \\ \{\mathit{vid} \mapsto (\tau, \mathtt{c})\}\ \langle + \{\mathit{vid} \mapsto (\tau' \to \tau, \mathtt{c})\}\ \rangle\ \langle\langle + \mathit{VE}\rangle\rangle \end{array}} \tag{36}$$

Comment: By the syntactic restrictions $\mathit{vid} \notin \mathrm{Dom}\,\mathit{VE}$.

Exception Bindings

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

$$\frac{\langle C \vdash ty \Rightarrow \tau \rangle \qquad \langle\langle C \vdash exbind \Rightarrow VE \rangle\rangle}{\begin{array}{l} C \vdash vid\ \langle \texttt{of}\ ty \rangle\ \langle\langle \texttt{and}\ exbind \rangle\rangle \Rightarrow \\ \{vid \mapsto (\texttt{exn}, \texttt{e}\}\ \langle + \{vid \mapsto (\tau \to \texttt{exn}, \texttt{e})\}\ \rangle\ \langle\langle + VE \rangle\rangle \end{array}} \tag{37}$$

$$\frac{C \vdash longvid \Rightarrow (\tau, \texttt{e}) \qquad \langle C \vdash exbind \Rightarrow VE \rangle}{C \vdash vid\ \texttt{=}\ longvid\ \langle \texttt{and}\ exbind \rangle \Rightarrow \{vid \mapsto (\tau, \texttt{e})\}\ \langle + VE \rangle} \tag{38}$$

Comments:

(37) Notice that τ may contain type variables.

(37),(38) For each C and *exbind*, there is at most one VE satisfying $C \vdash exbind \Rightarrow VE$.

Atomic Patterns

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

$$\frac{}{C \vdash \_ \Rightarrow (\{\}, \tau)} \tag{39}$$

$$\frac{}{C \vdash scon \Rightarrow (\{\}, \mathrm{type}(scon))} \tag{40}$$

$$\frac{vid \notin \mathrm{Dom}(C) \text{ or } is \text{ of } C(vid) = \texttt{v}}{C \vdash vid \Rightarrow (\{vid \mapsto (\tau, \texttt{v}\}, \tau)} \tag{41}$$

$$\frac{C \vdash longvid \Rightarrow (\sigma, is) \quad is \neq \texttt{v} \quad \sigma \succ \tau^{(k)}\ \vartheta}{C \vdash longvid \Rightarrow (\{\}, \tau^{(k)}\ \vartheta)} \tag{42}$$

$$\frac{\langle C \vdash patrow \Rightarrow (VE, \varrho) \rangle}{C \vdash \texttt{\{}\ \langle patrow \rangle\ \texttt{\}} \Rightarrow (\ \{\}\langle + VE \rangle,\ \{\}\langle + \varrho \rangle \text{ in Type }\ )} \tag{43}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau)}{C \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow (VE, \tau)} \tag{44}$$

Comments:

(41,42) The context C determines which of these two rules applies. In rule 41, note that *vid* can assume a type, not a general type scheme.

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

$$\frac{}{C \vdash \ldots \Rightarrow (\{\}, \varrho)} \tag{45}$$

$$\frac{\begin{array}{c} C \vdash pat \Rightarrow (VE, \tau) \\ \langle C \vdash patrow \Rightarrow (VE', \varrho) \qquad lab \notin \mathrm{Dom}\,\varrho \rangle \end{array}}{C \vdash lab\ \texttt{=}\ pat\ \langle\ \texttt{,}\ patrow \rangle \Rightarrow (VE\langle + \ VE'\rangle,\ \{lab \mapsto \tau\}\langle + \ \varrho\rangle)} \tag{46}$$

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

$$\frac{C \vdash atpat \Rightarrow (VE, \tau)}{C \vdash atpat \Rightarrow (VE, \tau)} \tag{47}$$

$$\frac{C \vdash longvid \Rightarrow (\sigma, is) \quad is \neq \mathtt{v} \quad \sigma \succ \tau' \to \tau \quad C \vdash atpat \Rightarrow (VE, \tau')}{C \vdash longvid\ atpat \Rightarrow (VE, \tau)} \tag{48}$$

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C \vdash ty \Rightarrow \tau}{C \vdash pat\ \texttt{:}\ \ ty \Rightarrow (VE, \tau)} \tag{49}$$

$$\frac{\begin{array}{c} vid \notin \mathrm{Dom}(C) \text{ or } is \text{ of } C(vid) = \mathtt{v} \\ \langle C \vdash ty \Rightarrow \tau \rangle \qquad C \vdash pat \Rightarrow (VE, \tau) \qquad vid \notin \mathrm{Dom}\,VE \end{array}}{C \vdash vid\langle \texttt{:}\ \ ty \rangle\ \texttt{as}\ pat \Rightarrow (\{vid \mapsto (\tau, \mathtt{v})\} + VE, \tau)} \tag{50}$$

Long Type Constructors $\boxed{C \vdash longtycon \Rightarrow (\theta^k, VE)}$

$$\frac{tycon \in \mathrm{Dom}\,C}{C \vdash tycon \Rightarrow C(tycon)} \tag{51}$$

$$\frac{C \vdash longstrid \Rightarrow RS \quad S = S \text{ of } RS \quad tycon \in \mathrm{Dom}\,S}{C \vdash longstrid.tycon \Rightarrow S(tycon)} \tag{52}$$

Type Constructor Paths

$$\boxed{C \vdash \mathit{tyconpath} \Rightarrow (\theta^k, \mathit{VE})}$$

$$\frac{C \vdash \mathit{longtycon} \Rightarrow (\theta, \mathit{VE})}{C \vdash \mathit{longtycon} \Rightarrow (\theta, \mathit{VE})} \tag{53}$$

$$\frac{\begin{array}{l} C \vdash^{\mathsf{s}} \mathit{modexp} \Rightarrow \exists T.\mathit{RS} \\ C + \{\mathit{strid} \mapsto \mathit{RS}\} \vdash \mathit{longtycon} \Rightarrow (\theta, \mathit{VE}) \\ T \cap (\mathrm{tynames}(C) \cup \mathrm{tynames}((\theta, \mathit{VE}))) = \emptyset \end{array}}{C \vdash \mathit{longtycon}\ \texttt{where}\ \mathit{strid} = \mathit{modexp} \Rightarrow (\theta, \mathit{VE})} \tag{54}$$

Type Identifier Sequences

$$\boxed{C \vdash \mathit{tyidseq} \Rightarrow (\alpha^{(k)}, \mathit{IE})}$$

$$\frac{\alpha \text{ admits equality iff } \mathit{tyid} \in \mathrm{ETyId} \quad \alpha \notin (\mathrm{tyvars}\, C)}{C \vdash \mathit{tyid} \Rightarrow ((\alpha), \{\mathit{tyid} \mapsto (\alpha)\})} \tag{55}$$

$$\frac{}{C \vdash \quad \Rightarrow ((), \{\})} \tag{56}$$

$$\frac{\begin{array}{c} \alpha_i \text{ admits equality iff } \mathit{tyid}_i \in \mathrm{ETyId}, i = 1..k \\ \alpha_i \notin (\mathrm{tyvars}\, C) \cup \{\alpha_1, \ldots, \alpha_{(i-1)}\}, i = 1..k \end{array}}{C \vdash (\mathit{tyid}_1, \cdots, \mathit{tyid}_k) \Rightarrow ((\alpha_1, \cdots, \alpha_k), \{\mathit{tyid}_1 \mapsto \alpha_1\} + \cdots + \{\mathit{tyid}_k \mapsto \alpha_k\})} \tag{57}$$

Type Expressions

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

$$\frac{\mathit{tyid} \in \mathrm{Dom}\ C}{C \vdash \mathit{tyid} \Rightarrow C(\mathit{tyid})} \tag{58}$$

$$\frac{\langle C \vdash \mathit{tyrow} \Rightarrow \varrho\rangle}{C \vdash \texttt{\{}\ \langle \mathit{tyrow}\rangle\ \texttt{\}} \Rightarrow \{\}\langle + \varrho\rangle \text{ in Type}} \tag{59}$$

$$\frac{\begin{array}{c} \mathit{tyseq} = \mathit{ty}_1 \cdots \mathit{ty}_k \qquad C \vdash \mathit{ty}_i \Rightarrow \tau_i\ (1 \le i \le k) \\ C \vdash \mathit{tyconpath} \Rightarrow (\theta^k, \mathit{VE}) \end{array}}{C \vdash \mathit{tyseq}\ \mathit{tyconpath} \Rightarrow \tau^{(k)}\ \theta^k} \tag{60}$$

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau \qquad C \vdash \mathit{ty}' \Rightarrow \tau'}{C \vdash \mathit{ty}\ \texttt{->}\ \mathit{ty}' \Rightarrow \tau \to \tau'} \tag{61}$$

$$\frac{C \vdash sigexp \Rightarrow \Lambda T.M}{C \vdash \texttt{[}\ sigexp\ \texttt{]} \Rightarrow [\exists T.M]} \tag{62}$$

$$\frac{C \vdash ty \Rightarrow \tau}{C \vdash \texttt{(}\ ty\ \texttt{)} \Rightarrow \tau} \tag{63}$$

Comments:

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

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

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash tyrow \Rightarrow \varrho\rangle}{C \vdash lab\ \texttt{:}\ ty\ \langle\ \texttt{,}\ tyrow\rangle \Rightarrow \{lab \mapsto \tau\}\langle + \varrho\rangle} \tag{64}$$

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

5.11 Further Restrictions

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

1. For each occurrence of a record pattern containing a record wildcard, i.e. of the form `{`lab_1`=`pat_1`,`$\cdots$`,`lab_m`=`pat_m`,...}` the program context must determine uniquely the domain $\{lab_1, \cdots, lab_n\}$ of its row 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.

2. In a match of the form pat_1 `=>` exp_1`|` $\cdots$ `|` `pat`$_n$ `=>` exp_n the pattern sequence $pat_1, \ldots, pat_n$ should be *irredundant*; that is, each pat_j must match some value (of the right type) which is not matched by pat_i for any $i < j$. In the context `fn` *match*, the *match* must also be *exhaustive*; that is, every value (of the right type) must be matched by some pat_i. The compiler must give warning on violation of these restrictions, but should still compile the match. The restrictions are inherited by derived forms; in particular, this means that in the function-value binding $vid\ atpat_1\ \cdots\ atpat_n\langle\texttt{:}\ ty\rangle$ `=` exp (consisting of one clause only), each separate $atpat_i$ should be exhaustive by itself.

3. For each value binding *pat* `=` *exp* the compiler must issue a report (but still compile) if *pat* is not exhaustive. This will detect a mistaken declaration like `val nil =` *exp* in which the user expects to declare a new variable `nil` (whereas the language dictates that `nil` is here a constant pattern, so no variable gets declared). However, this warning should not be given when the binding is a component of a top-level declaration `val` *tyidseq valbind*; e.g. `val x::l =` exp_1 `and y =` exp_2 is not faulted by the compiler at top level, but may of course generate a `Bind` exception (see Section **??**).

6 Static Semantics for Modules

6.1 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 12.

$$
\begin{array}{rcl}
S \text{ or } (FE, SE, TE, VE) & \in & \mathrm{Str} = \mathrm{FunEnv} \times \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
RS \text{ or } S \text{ or } (RS_1, RS_2) & \in & \mathrm{RecStr} = \mathrm{Str} \ \cup (\mathrm{RecStr} \times \mathrm{RecStr}) \\
M \text{ or } RS \text{ or } F & \in & \mathrm{Mod} = \mathrm{RecStr} \ \cup \mathrm{Fun} \\
F \text{ or } \forall T.M \rightarrow X & \in & \mathrm{Fun} = \mathrm{TyNameSet} \times \mathrm{Mod} \times \mathrm{ExMod} \\
X \text{ or } \exists T.M & \in & \mathrm{ExMod} = \mathrm{Ex}(\mathrm{Mod}) = \mathrm{TyNameSet} \times \mathrm{Mod} \\
G \text{ or } \Lambda T.M & \in & \mathrm{Sig} = \mathrm{Par}(\mathrm{Mod}) = \mathrm{TyNameSet} \times \mathrm{Mod} \\
GE & \in & \mathrm{SigEnv} = \mathrm{SigId} \stackrel{\mathrm{fin}}{\rightarrow} \mathrm{Sig} \\
FE & \in & \mathrm{FunEnv} = \mathrm{FunId} \stackrel{\mathrm{fin}}{\rightarrow} \mathrm{Fun} \\
SE & \in & \mathrm{StrEnv} = \mathrm{StrId} \stackrel{\mathrm{fin}}{\rightarrow} \mathrm{RecStr}
\end{array}
$$

Figure 12: Further Compound Semantic Objects

The prefixes $\Lambda T.\_$, $\exists T, _.a$nd $\forall T.\_ \rightarrow \_$ in parameterised objects, existential objects and functors bind type names. Certain operations require a change of bound names in semantic objects; see for example Section 6.2. When bound type names are changed, we demand that all of their attributes (i.e. equality and kind) are preserved.

The operations of projection, injection and modification are as for the Core, with the following additions:

For a recursive structure RS, we define $(S \text{ of } RS) = S$ if $RS = S$ and $(S \text{ of } RS) = (S \text{ of } RS_2)$ if $RS = (RS_1, RS_2)$. The operation projects the type of the body of a recursive structure.

We overload the notation for environment modification $C + \{modid \mapsto M\}$ to mean $C + \{strid \mapsto RS\}$ if $M = RS$ and where $strid = modid$; and $C + \{funid \mapsto RS\}$ if $M = F$ and where $funid = modid$. The former extends the structure environment of C, interpreting the module identifier as a structure identifier, the latter extends the functor environment of C, interpreting the module identifier as a functor identifier. Which interpretation to apply is uniquely determined by the form of M.

6.2 Type Realisation

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

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

Realisations φ are extended to apply to all semantic objects; their effect is to replace each name t by $\varphi(\mathrm{t})$ (performing β-reductions as necessary to preserve the structure of constructed types, type applications and type functions). In applying φ to an object with bound names, such as a signature $\Lambda T.M$, first bound names must be changed so that, for each binding prefix ($\Lambda T.\_$, $\exists T.\_$ and $\forall T.\_ \to \_$),

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

6.3 Signature Instantiation

A module M *is an instance of* a signature $G = \Lambda T.M'$, written $G \geq M$, if there exists a realisation φ such that $\varphi(M') = M$ and $\mathrm{Supp}\,\varphi \subseteq T$.

6.4 Functor Instantiation

An object $M \to X$ is called a *functor instance*. Given $F = \forall T_1.M_1 \to X_1$, a functor instance $M_2 \to X_2$ is an *instance* of F, written $F \geq M_2 \to X_2$, if there exists a realisation φ such that $\varphi(M_1 \to X_1) = M_2 \to X_2$ and $\mathrm{Supp}\,\varphi \subseteq T_1$.

6.5 Enrichment

In matching a (recursive) 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. In matching an functor to a signature, the functor will be allowed to be more polymorphic, have a less rich domain, and have a richer range than (an instance of) the signature.

Precisely, we define enrichment of structures, recursive structures, functors, modules, existential modules, and type structures by mutual recursion as follows.

- A structure $S_1 = (FE_1, SE_1, TE_1, VE_1)$ *enriches* another structure $S_2 = (FE_2, SE_2, TE_2, VE_2)$, written $S_1 \succ S_2$, if
  1. $\mathrm{Dom}\, FE_1 \supseteq \mathrm{Dom}\, FE_2$, and $FE_1(\mathit{funid}) \succ FE_2(\mathit{funid})$ for all $\mathit{funid} \in \mathrm{Dom}\, FE_2$,
  2. $\mathrm{Dom}\, SE_1 \supseteq \mathrm{Dom}\, SE_2$, and $SE_1(\mathit{strid}) \succ SE_2(\mathit{strid})$ for all $\mathit{strid} \in \mathrm{Dom}\, SE_2$,
  3. $\mathrm{Dom}\, TE_1 \supseteq \mathrm{Dom}\, TE_2$, and $TE_1(\mathit{tycon}) \succ TE_2(\mathit{tycon})$ for all $\mathit{tycon} \in \mathrm{Dom}\, TE_2$, and
  4. $\mathrm{Dom}\, VE_1 \supseteq \mathrm{Dom}\, VE_2$, and $VE_1(\mathit{vid}) \succ VE_2(\mathit{vid})$ for all $\mathit{vid} \in \mathrm{Dom}\, VE_2$.
- A recursive structure RS_1 *enriches* another recursive structure RS_2, written $RS_1 \succ RS_2$, if
  1. $RS_2 = S_2$ and $(S \text{ of } RS_1) \succ S_2$ for some (non-recursive) structure S_2, or
  2. $RS_2 = (RS_3, RS_4)$ and $(S \text{ of } RS_1) \succ RS_3$ and $(S \text{ of } RS_1) \succ RS_4$ for some recursive structures RS_3 and RS_4.
- A functor $F_1 = \forall T_1.M_1 \to X_1$ *enriches* another functor $F_2 = \forall T_2.M_2 \to X_2$, written $F_1 \succ F_2$, if there exists a realisation φ such that:
  1. $T_2 \cap \mathrm{tynames}(F_1) = \emptyset$,
  2. $M_2 \succ \varphi(M_1)$,
  3. $\varphi(X_1) \succ X_2$, and
  4. $\mathrm{Supp}\, \varphi \subseteq T_1$.
- A module M_1 *enriches* another module M_2, written $M_1 \succ M_2$, if:
  1. $M_1 = RS_1$, $M_2 = RS_2$ and $RS_1 \succ RS_2$ for some recursive structures RS_1 and RS_2, or
  2. $M_1 = F_1$, $M_2 = F_2$ and $F_1 \succ F_2$ for some functors F_1 and F_2.
- An existential module $X_1 = \exists T_1.M_1$ *enriches* another existential module $X_2 = \exists T_2.M_2$, written $X_1 \succ X_2$, if:
  1. $T_1 \cap \mathrm{tynames}\,(X_2) = \emptyset$ and $M_1 \succ \varphi_2(M_2)$ for some realisation φ_2 with $\mathrm{Supp}\, \varphi_2 \subseteq T_2$.

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

  1. θ_1 and θ_2 have the same kind,
  2. $\theta_1 = \theta_2$, and
  3. Either $VE_1 = VE_2$ or $VE_2 = \{\}$.

6.6 Signature Matching

A module M *matches* a signature G if there exists a module M^- such that $G \geq M^- \prec M$. Thus matching is a combination of instantiation and enrichment. For any G and M that must be matched during elaboration from the initial context, there will be at most one such M^-.

6.7 Equivalence of Package Types

We identify package types $[X] \in \mathrm{PackType}$ that differ only in a kind and attribute preserving renaming of bound type names. Moreover, since we do not want to distinguish between package types that differ merely in a reordering of components, we identify package types that are equivalent according to the following definition.

Two package types $[X_1]$ and $[X_2]$ are *equivalent* if, and only if, $X_1 \succ X_2$ and $X_2 \succ X_1$ (each enriches the other).

6.8 Applicative Module Expressions

To preserve the type soundness property in the presence of both applicative functors and first-class modules (Core expressions of package type), the set ModExp of module expressions is divided into a further subclass, the set of *applicative* module expressions. Informally, a module expression *modexp* is applicative only if it contains no structure or functor binding of the form *strid* `as` *sigexp* `=` *exp* or *funid* `as` *sigexp* `=` *exp*, unless that binding occurs within the declaration *dec* of a Core sub-expression of the form `let` *dec* `in` *exp* `end`. Formally, a module expression is *applicative* if, after replacing derived forms by their equivalent forms, it can be generated by the following grammar from the non-terminal *appmodexp* in Figure 13:

appdec	::=	`val` *tyidseq valbind*
		`type` *typbind*
		`datatype` *datbind*
		`datatype` *tycon* `=` `datatype` *tyconpath*
		`abstype` *datbind* `with` *appdec* `end`
		`exception` *exbind*
		`local` $appdec_1$ `in` $appdec_2$ `end`
		`open` $longstrid_1 \cdots longstrid_n$
		`structure` *appstrbind*
		`functor` *appfunbind*
		`signature` *sigbind*
		$appdec_1$ ⟨;⟩ $appdec_2$
		`infix` ⟨*d*⟩ $vid_1 \cdots vid_n$
		`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$
		`nonfix` $vid_1 \cdots vid_n$
appstrbind	::=	*strid* `=` *appmodexp* ⟨`and` *appstrbind*⟩
appfunbind	::=	*funid* `=` *appmodexp* ⟨`and` *appfunbind*⟩
appatmodexp	::=	`struct` *appdec* `end`
		⟨`op`⟩*longmodid*
		`let` *appdec* `in` *modexp* `end`
		`(` *appmodexp* `)`
appmodexp	::=	*appatmodexp*
		appmodexp appatmodexp
		appmodexp `:` *sigexp*
		appmodexp `:>` *sigexp*
		`functor` `(`*modid*`:`*sigexp*`)` `=>` *modexp*
		`functor` *modid*`:`*sigexp* `=>` *modexp*
		`rec` `(`*strid*`:`*sigexp*`)` *appmodexp*

Figure 13: Applicative Module Expressions

Type soundness is preserved by the inference rules by ensuring that if *modexp* occurs as the body of a functor, i.e. a phrases of the form

`functor` *modid*: *sigexp* `=>` *modexp*

or

`functor (`*modid*: *sigexp*`) =>` *modexp*,

then *modexp* must be applicative.

This restriction applies only to functor bodies; in particular, it does *not* preclude declarations of the form *strid* `as` *sigexp* `=` *exp* or *funid* `as` *sigexp* `=` *exp* from occurring in top-level declarations or structures, nor does it preclude them from occurring within Core expressions of the form `let` *dec* `in` *exp* `end`.

6.9 Resolution of Long Module Identifiers

Although StrId and FunId are regarded as disjoint by the semantics, in the sense that structures and functors reside in separate name-spaces, the syntax of structure and functor identifiers is, in fact, shared. *A priori*, the module expression ⟨`op`⟩*longmodid* may refer to a either a functor or a structure so the semantics must dictate how to resolve this ambiguity. Fortunately, the context of the phrase often rules out one alternative, on the grounds that choosing that alternative would force elaboration to fail. In particular, if ⟨`op`⟩*longmodid* occurs as the right hand side of a structure (functor) binding, then *longmodid* must be interpreted as an element of longStrId (longFunId); if ⟨`op`⟩*longmodid* occurs in the functor position of an application, then *longmodid* must be interpreted as an element of longFunId; if ⟨`op`⟩*longmodid* is constrained by a signature then the signature forces a unique interpretation on *longmodid* (depending on whether the signature specificies a structure or functor). Similarly, if ⟨`op`⟩*longmodid* occurs as the argument of a functor application, then the functor's domain forces a unique interpretation on *longmodid*. Indeed, the only ambiguity that remains occurs when ⟨`op`⟩*longmodid* is the body of a functor. In this case, the optional prefix ⟨`op`⟩ is used to resolve the ambiguity: the *absence* of `op` signals that *longmodid* refers to structure; the *presence* of `op` signals that `op` *longmodid* refers to a functor. When the interpretation of ⟨`op`⟩*longmodid* is already determined by the context, the optional prefix ⟨`op`⟩ has *no* effect.

Since this method of disambiguation relies on type informatifon it is formalised within the static semantic rules. In the rules for elaborating module expressions, the context of the current phrase is summarized by an *expectation*:

$$ex \in \text{Expectation} = \{\texttt{s}, \texttt{f}, \texttt{m}\}$$

Of the three values: `s` indicates that a structure is expected; `f` indicates that a functor is expected; finally, `m` indicates that a structure or a functor is expected, so that the status of $\langle\texttt{op}\rangle$*longmodid* must be resolved by $\langle\texttt{op}\rangle$.

The inference rules rely on two auxilliary functions that determine the expectation for a subderivation depending on the form of a signature or the domain of a functor:

$$\begin{array}{rcl}
\text{expect}(_) & \in & \text{Sig} \to \text{Expectation} \\
\text{expect}(G) & = & \begin{cases} \texttt{s} & \text{if } G = \Lambda T.RS \\ \texttt{f} & \text{if } G = \Lambda T.F \end{cases}
\end{array}$$

$$\begin{array}{rcl}
\text{expect}(_) & \in & \text{Fun} \to \text{Expectation} \\
\text{expect}(F) & = & \begin{cases} \texttt{s} & \text{if } F = \forall T.RS \to X \\ \texttt{f} & \text{if } F = \forall T.F' \to X \end{cases}
\end{array}$$

6.10 Inference Rules

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

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

to be inferred, where A is typically a context, *phrase* is a phrase of the Modules language, and A' is a semantic object. The convention for options is as in the Core semantics.

Long Structure Identifiers $\boxed{C \vdash \mathit{longstrid} \Rightarrow \mathit{RS}}$

$$\frac{\mathit{strid} \in \operatorname{Dom} C}{C \vdash \mathit{strid} \Rightarrow C(\mathit{strid})} \tag{65}$$

$$\frac{C \vdash \mathit{longstrid} \Rightarrow \mathit{RS} \quad \mathit{SE} = \mathit{SE} \text{ of } (S \text{ of } \mathit{RS}) \quad \mathit{strid} \in \operatorname{Dom} \mathit{SE}}{C \vdash \mathit{longstrid}.\mathit{strid} \Rightarrow \mathit{SE}(\mathit{strid})} \tag{66}$$

Long Functor Identifiers

$$\boxed{C \vdash \mathit{longfunid} \Rightarrow F}$$

$$\frac{\mathit{funid} \in \mathrm{Dom}\, C}{C \vdash \mathit{funid} \Rightarrow C(\mathit{funid})} \tag{67}$$

$$\frac{C \vdash \mathit{longstrid} \Rightarrow \mathit{RS} \quad \mathit{FE} = \mathit{FE} \text{ of } (S \text{ of } \mathit{RS}) \quad \mathit{funid} \in \mathrm{Dom}\, \mathit{FE}}{C \vdash \mathit{longstrid}.\mathit{funid} \Rightarrow \mathit{FE}(\mathit{funid})} \tag{68}$$

Long Module Identifiers

$$\boxed{C \vdash^{ex} \langle \texttt{op} \rangle\ \mathit{longmodid} \Rightarrow M}$$

$$\frac{C \vdash \mathit{longstrid} \Rightarrow \mathit{RS} \quad \mathit{longstrid} = \mathit{longmodid}}{C \vdash^{\texttt{s}} \langle \texttt{op} \rangle\ \mathit{longmodid} \Rightarrow \mathit{RS}} \tag{69}$$

$$\frac{C \vdash \mathit{longfunid} \Rightarrow F \quad \mathit{longfunid} = \mathit{longmodid}}{C \vdash^{\texttt{f}} \langle \texttt{op} \rangle\ \mathit{longmodid} \Rightarrow F} \tag{70}$$

$$\frac{C \vdash \mathit{longstrid} \Rightarrow \mathit{RS} \quad \mathit{longstrid} = \mathit{longmodid}}{C \vdash^{\texttt{m}} \mathit{longmodid} \Rightarrow \mathit{RS}} \tag{71}$$

$$\frac{C \vdash \mathit{longfunid} \Rightarrow F \quad \mathit{longfunid} = \mathit{longmodid}}{C \vdash^{\texttt{m}} \texttt{op}\ \mathit{longmodid} \Rightarrow F} \tag{72}$$

Atomic Module Expressions

$$\boxed{C \vdash^{ex} \mathit{atmodexp} \Rightarrow X}$$

$$\frac{C \vdash \mathit{dec} \Rightarrow \exists T.E}{C \vdash^{ex} \texttt{struct}\ \mathit{dec}\ \texttt{end} \Rightarrow \exists T.(\mathit{FE} \text{ of } E, \mathit{SE} \text{ of } E, \mathit{TE} \text{ of } E, \mathit{VE} \text{ of } E)} \tag{73}$$

$$\frac{C \vdash^{ex} \langle \texttt{op} \rangle\ \mathit{longmodid} \Rightarrow M}{C \vdash^{ex} \langle \texttt{op} \rangle\ \mathit{longmodid} \Rightarrow \exists \emptyset.M} \tag{74}$$

$$\frac{\begin{array}{ll} C \vdash \mathit{dec} \Rightarrow \exists T_1.E & T_1 \cap \mathrm{tynames}\, C = \emptyset \\ C + E \vdash^{ex} \mathit{modexp} \Rightarrow \exists T_2.M & T_2 \cap T_1 = \emptyset \end{array}}{C \vdash^{ex} \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{modexp}\ \texttt{end} \Rightarrow \exists T_1 \cup T_2.M} \tag{75}$$

$$\frac{C \vdash^{ex} \mathit{modexp} \Rightarrow X}{C \vdash^{ex} \texttt{(}\ \mathit{modexp}\ \texttt{)} \Rightarrow X} \tag{76}$$

Comments:

(73) The resulting structure contains the functor, structure, type and value components of E. Signatures declared in *dec* are local to *dec* and not visible from outside the structure.

(75) The side condition $T_1 \cap \text{tynames}\, C$, here and elsewhere, ensures that eliminating the first existential quantifier does not capture type names occurring free in the context. The side condition can always be satisfied by renaming bound names in $\exists T_1.E$. Existentially quantifying over both T_1 and T_2 in the result ensures that the hypothetical type names in T_1 do not escape their scope.

Module Expressions

$$\boxed{C \vdash^{ex} \mathit{modexp} \Rightarrow X}$$

$$\frac{C \vdash^{ex} \mathit{atmodexp} \Rightarrow X}{C \vdash^{ex} \mathit{atmodexp} \Rightarrow X} \tag{77}$$

$$\frac{\begin{array}{lll} C \vdash^{\mathtt{f}} \mathit{modexp} \Rightarrow \exists T.F & & C \vdash^{\mathrm{expect}(F)} \mathit{atmodexp} \Rightarrow \exists T'.M \\ T \cap (T' \cup \text{tynames}\, M) = \emptyset & & T' \cap \text{tynames}\, F = \emptyset \\ F \geq M' \to \exists T''.M'' & M \succ M' & T'' \cap (T \cup T') = \emptyset \end{array}}{C \vdash^{ex} \mathit{modexp}\ \mathit{atmodexp} \Rightarrow \exists T \cup T' \cup T''.M''} \tag{78}$$

$$\frac{\begin{array}{ll} C \vdash \mathit{sigexp} \Rightarrow G & C \vdash^{\mathrm{expect}(G)} \mathit{modexp} \Rightarrow \exists T.M \\ T \cap \text{tynames}\, G = \emptyset & G \geq M' \succ M \end{array}}{C \vdash^{ex} \mathit{modexp}\ :\ \mathit{sigexp} \Rightarrow \exists T.M'} \tag{79}$$

$$\frac{\begin{array}{ll} C \vdash \mathit{sigexp} \Rightarrow \Lambda T'.M' & C \vdash^{\mathrm{expect}(\Lambda T'.M')} \mathit{modexp} \Rightarrow \exists T.M \\ T \cap \text{tynames}(\Lambda T'.M') = \emptyset & \Lambda T'.M' \geq M'' \succ M \end{array}}{C \vdash^{ex} \mathit{modexp}\ \texttt{:>}\ \mathit{sigexp} \Rightarrow \exists T'.M'} \tag{80}$$

$$\frac{\begin{array}{l} \mathit{modexp} \text{ is applicative} \\ C \vdash \mathit{sigexp} \Rightarrow \Lambda T.M \\ T \cap \text{tynames}\, C = \emptyset \\ C + \{\mathit{modid} \mapsto M\} \vdash^{\mathtt{m}} \mathit{modexp} \Rightarrow X \end{array}}{C \vdash^{ex} \texttt{functor (}\mathit{modid}\texttt{:}\,\mathit{sigexp}\texttt{) =>}\ \mathit{modexp} \Rightarrow \exists\emptyset.(\forall T.M \to X)} \tag{81}$$

$$\frac{\begin{array}{l} \mathit{modexp} \text{ is applicative} \\ C \vdash \mathit{sigexp} \Rightarrow \Lambda T.M \\ T \cap \text{tynames}\, C = \emptyset \quad T = \{\text{t}_1^{K_1}, \ldots, \text{t}_n^{K_n}\} \\ C + \{\mathit{modid} \mapsto M\} \vdash^{\mathtt{m}} \mathit{modexp} \Rightarrow \exists T'.M' \\ T'' \cap (T \cup \text{tynames}\, M \cup \text{tynames}(\exists T'.M')) = \emptyset \\ \varphi = \{\text{u}^K \mapsto \text{u}^{K_1 \to \cdots K_n \to K}\ \text{t}_1 \cdots \text{t}_n ; \text{u}^K \in T'\} \\ T'' = \{\text{u}^{K_1 \to \cdots K_n \to K} ; \text{u}^K \in T'\} \end{array}}{C \vdash^{ex} \mathtt{functor}\ \mathit{modid} \colon \mathit{sigexp}\ \mathtt{=>}\ \mathit{modexp} \Rightarrow \exists T''.(\forall T.M \to \exists\emptyset.\varphi(M'))} \quad (82)$$

$$\frac{\begin{array}{l} C \vdash \mathit{sigexp} \Rightarrow \Lambda T.\mathit{RS} \\ T \cap \text{tynames}\, C = \emptyset \\ C + \{\mathit{strid} \mapsto \mathit{RS}\} \vdash^{\mathtt{s}} \mathit{modexp} \Rightarrow \exists T'.\mathit{RS} \\ T' \cap (T \cup \text{tynames}\, \mathit{RS}) = \emptyset \\ \mathit{RS}' \succ \mathit{RS} \end{array}}{C \vdash^{ex} \mathtt{rec}\ (\mathit{strid} \colon \mathit{sigexp})\, \mathit{modexp} \Rightarrow \exists T \cup T'.(\mathit{RS}, \mathit{RS}')} \quad (83)$$

Comments:

(**??**) The side conditions on T, T' and T'' can always be satisfied by renaming bound names. T is the set of existential type names introduced by the functor. T' is the set of existential type names introduced by the argument. T'' is the set of existential type names introduced by the functor body. The side conditions on T, T', and T'' ensure that eliminating the existential quantifiers does not capture any free or hypothetical type names. Existentially quantifying over $T \cup T'$ as well as T'' in the result type M'' prevents any hypothetical type names, that may occur free in actual range of the application, from escaping their scope.

Let $F = \forall T_F.M_F \to X_F$. Let φ be a realisation such that $\varphi(M_F \to X_F) = M' \to \exists T''.M''$ (with $\text{Supp}\, \varphi \subseteq T_F$). Sharing between the formal domain and the formal range of the functor is represented by occurrences of the same type name of T_F in both M_F and X_F. These shared occurrences are preserved by φ, yielding sharing between the actual domain M and the actual range type $\exists T''.M''$ of this functor application.

(81),(82) In both rules, the functor body *modexp* is elaborated in the extended context $C + \{\mathit{modid} \mapsto M\}$. The side condition $T \cap \text{tynames}\, C =$

$\emptyset$, which may always be satisfied by a renaming of bound names in $\Lambda T.M$, ensures that the type names in T are treated as parameters during elaboration of the body, so that M represents a generic instance of the signature. Thus the functor may be applied at any realisation of these parameters and, in particular, to any argument whose type matches the signature $\Lambda T.M$.

(81) The type of the functor body is an existentially quantified module type X of the form $\exists T'.M'$. This type determines the range of the functor $\forall T.M \rightarrow X = \forall T.M \rightarrow \exists T'.M'$. Observe that the scope of the existential quantifier implies that distinct applications of this functor will introduce distinct abstract types (even when the functor is applied at the same realisation). Thus functors of this form have a *generative* semantics.

(82) Elaborating the body introduces existential type names T. In general, because *modexp* is elaborated in the extended context $C + \{modid \mapsto M\}$, names in T' may have hidden functional dependencies on the type parameters T of the formal argument *modid*. These dependencies are made explicit by applying the realisation φ to M'. This effectively *skolemises* each occurrence in M' of a name $\mathrm{u} \in T'$ by the names in T. The kinds of names in T' must be adjusted to reflect this, yielding the set T''. Having parameterised names in T' by their implicit arguments, the existential quantifier can be moved from its scope within the functor range, i.e. $\exists T'.M'$, to a scope that encloses the entire functor, yielding the existential module $\exists T''.(\forall T.M \rightarrow \exists\emptyset.\varphi(M'))$.

Observe that the scope of the existential quantifier implies that distinct applications of this functor, at equivalent realisations, will yield equivalent abstract types. Thus functors of this form have an *applicative* semantics.

Structure Bindings $\boxed{C \vdash \mathit{strbind} \Rightarrow \exists T.\mathit{SE}}$

$$\frac{\begin{array}{l} C \vdash^{\mathtt{s}} \mathit{modexp} \Rightarrow \exists T.\mathit{RS} \\ \langle C \vdash \mathit{strbind} \Rightarrow \exists T'.\mathit{SE}\rangle \\ \langle T \cap (T' \cup \mathrm{tynames}\,\mathit{SE}) = \emptyset\rangle \\ \langle T' \cap (\mathrm{tynames}\,\mathit{RS}) = \emptyset\rangle \end{array}}{\begin{array}{c} C \vdash \mathit{strid}\ \mathtt{=}\ \mathit{modexp}\ \langle\mathtt{and}\ \mathit{strbind}\rangle \Rightarrow \\ \exists T\langle\cup T'\rangle.\{\mathit{strid} \mapsto \mathit{RS}\}\ \langle +\ \mathit{SE}\rangle \end{array}} \tag{84}$$

$$\frac{\begin{array}{l} C \vdash \mathit{sigexp} \Rightarrow \Lambda T.\mathit{RS} \\ C \vdash \mathit{exp} \Rightarrow [\exists T.\mathit{RS}] \\ \langle C \vdash \mathit{strbind} \Rightarrow \exists T'.\mathit{SE}\rangle \\ \langle T \cap (T' \cup \mathrm{tynames}\,\mathit{SE}) = \emptyset\rangle \\ \langle T' \cap (\mathrm{tynames}\,\mathit{RS}) = \emptyset\rangle \end{array}}{\begin{array}{c} C \vdash \mathit{strid}\ \mathtt{as}\ \mathit{sigexp}\ \mathtt{=}\ \mathit{exp}\ \langle\mathtt{and}\ \mathit{strbind}\rangle \Rightarrow \\ \exists T\langle\cup T'\rangle.\{\mathit{strid} \mapsto \mathit{RS}\}\ \langle +\ \mathit{SE}\rangle \end{array}} \tag{85}$$

Functor Bindings $\boxed{C \vdash \mathit{funbind} \Rightarrow \exists T.\mathit{FE}}$

$$\frac{\begin{array}{l} C \vdash^{\mathtt{f}} \mathit{modexp} \Rightarrow \exists T.F \\ \langle C \vdash \mathit{funbind} \Rightarrow \exists T'.\mathit{FE}\rangle \\ \langle T \cap (T' \cup \mathrm{tynames}\,\mathit{FE}) = \emptyset\rangle \\ \langle T' \cap (\mathrm{tynames}\,F) = \emptyset\rangle \end{array}}{\begin{array}{c} C \vdash \mathit{funid}\ \mathtt{=}\ \mathit{modexp}\ \langle\mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \exists T\langle\cup T'\rangle.\{\mathit{funid} \mapsto F\}\ \langle +\ \mathit{FE}\rangle \end{array}} \tag{86}$$

$$\frac{\begin{array}{l} C \vdash \mathit{sigexp} \Rightarrow \Lambda T.F \\ C \vdash \mathit{exp} \Rightarrow [\exists T.F] \\ \langle C \vdash \mathit{funbind} \Rightarrow \exists T'.\mathit{FE}\rangle \\ \langle T \cap (T' \cup \mathrm{tynames}\,\mathit{FE}) = \emptyset\rangle \\ \langle T' \cap (\mathrm{tynames}\,F) = \emptyset\rangle \end{array}}{\begin{array}{c} C \vdash \mathit{funid}\ \mathtt{as}\ \mathit{sigexp}\ \mathtt{=}\ \mathit{exp}\ \langle\mathtt{and}\ \mathit{funbind}\rangle \Rightarrow \\ \exists T\langle\cup T'\rangle.\{\mathit{funid} \mapsto F\}\ \langle +\ \mathit{FE}\rangle \end{array}} \tag{87}$$

Signature Bindings $\boxed{C \vdash \mathit{sigbind} \Rightarrow \mathit{GE}}$

$$\frac{C \vdash \mathit{sigexp} \Rightarrow G \quad \langle C \vdash \mathit{sigbind} \Rightarrow \mathit{GE}\rangle}{C \vdash \mathit{sigid}\ \mathtt{=}\ \mathit{sigexp}\ \langle\mathtt{and}\ \mathit{sigbind}\rangle \Rightarrow \{\mathit{sigid} \mapsto G\}\ \langle +\ \mathit{GE}\rangle} \tag{88}$$

Signature Expressions

$$\boxed{C \vdash sigexp \Rightarrow G}$$

$$\frac{C \vdash spec \Rightarrow \Lambda T.(GE, FE, SE, TE, VE)}{C \vdash \mathtt{sig}\ spec\ \mathtt{end} \Rightarrow \Lambda T.(FE, SE, TE, VE)} \tag{89}$$

$$\frac{sigid \in \mathrm{Dom}\, C}{C \vdash sigid \Rightarrow C(sigid)} \tag{90}$$

$$\frac{\begin{array}{c} C \vdash sigexp \Rightarrow \Lambda T.RS \qquad S = S \text{ of } RS \\ C \vdash tyidseq \Rightarrow (\alpha^{(k)}, IE) \quad C + IE \vdash ty \Rightarrow \tau \\ (\{\}, (\{\}, FE \text{ of } S, SE \text{ of } S, TE \text{ of } S, VE \text{ of } S)) \vdash longtycon \Rightarrow (\mathrm{t}, VE) \\ \mathrm{t} \in T \quad \mathrm{t} \text{ has arity } k \\ (T \setminus \{\mathrm{t}\}) \cap \mathrm{tynames}\, \Lambda\alpha^{(k)}.\tau = \emptyset \\ \varphi = \{\mathrm{t} \mapsto \Lambda\alpha^{(k)}.\tau\} \quad \Lambda\alpha^{(k)}.\tau \text{ admits equality, if t does} \\ \varphi(S) \text{ well-formed} \end{array}}{C \vdash sigexp\ \mathtt{where\ type}\ tyidseq\ longtycon\ \mathtt{=}\ ty \Rightarrow \Lambda T \setminus \{\mathrm{t}\}.\varphi(RS)} \tag{91}$$

$$\frac{\begin{array}{l} C \vdash sigexp_1 \Rightarrow \Lambda T.M \\ T \cap \mathrm{tynames}\, C = \emptyset \\ C + \{modid \mapsto M\} \vdash sigexp_2 \Rightarrow \Lambda T'.M' \end{array}}{C \vdash \mathtt{functor}\ \mathtt{(}modid\ \mathtt{:}\ sigexp_1\mathtt{)}\ \mathtt{->}\ sigexp_2 \Rightarrow \Lambda\emptyset.(\forall T.M \to \exists T'.M')} \tag{92}$$

$$\frac{\begin{array}{l} C \vdash sigexp_1 \Rightarrow \Lambda T.M \\ T \cap \mathrm{tynames}\, C = \emptyset \quad T = \{\mathrm{t}_1^{K_1}, \ldots, \mathrm{t}_n^{K_n}\} \\ C + \{modid \mapsto M\} \vdash sigexp_2 \Rightarrow \Lambda T'.M' \\ T'' \cap (T \cup \mathrm{tynames}\, M \cup \mathrm{tynames}(\Lambda T'.M')) = \emptyset \\ \varphi = \{\mathrm{u}^K \mapsto \mathrm{u}^{K_1 \to \cdots K_n \to K}\ \mathrm{t}_1\ \cdots\ \mathrm{t}_n; \mathrm{u}^K \in T'\} \\ T'' = \{\mathrm{u}^{K_1 \to \cdots K_n \to K}; \mathrm{u}^K \in T'\} \end{array}}{C \vdash \mathtt{functor}\ modid\ \mathtt{:}\ sigexp_1\ \mathtt{->}\ sigexp_2 \Rightarrow \Lambda T''.(\forall T.M \to \exists\emptyset.\varphi(M'))} \tag{93}$$

$$\frac{\begin{array}{l} C \vdash sigexp_1 \Rightarrow \Lambda T.RS \\ T \cap \mathrm{tynames}\, C = \emptyset \\ C + \{strid \mapsto RS\} \vdash sigexp_2 \Rightarrow \Lambda T'.RS' \\ T' \cap (T \cup \mathrm{tynames}\, RS) = \emptyset \\ \varphi(RS') \succ \varphi(RS) \\ \mathrm{Supp}\, \varphi = T \\ T \cap \mathrm{Yield}(\varphi) = \emptyset \end{array}}{C \vdash \texttt{rec (}strid\texttt{:}\, sigexp\texttt{)}\, sigexp \Rightarrow \Lambda T'.\varphi(RS, RS')} \tag{94}$$

Comments:

(89) The resulting signature contains the functor, structure, type and value components of E. Signatures declared in *spec* are local to *spec* and not visible from the signature.

(94) An opaque functor signature specifies a set of functors. A functor that "matches" the signature `functor (`*modid* `:` $sigexp_1$`) ->` $sigexp_2$ must be applicable to any actual argument whose type "matches" $sigexp_1$. Thus $sigexp_1$ specifies the type parameters T and domain M of any matching functor. The signature expression $sigexp_2$, which is elaborated in the extended context $C + \{modid \mapsto M\}$, specifies the range of the matching functor, up to some opaque realisation of T'.

In this way, the type parameters arising $sigexp_1$ determine the polymorphism of the specified functors, while the type parameters arising from $sigexp_2$ hide variation in the range of the specified functors.

(93) A transparent functor signature specifies a family of functors. A functor that "matches" the signature `functor` *modid* `:` $sigexp_1$ `->` $sigexp_2$ must be applicable to any actual argument whose type "matches" $sigexp_1$. Thus $sigexp_1$ specifies the type parameters T and domain M of any matching functor. The range signature expression $sigexp_2$, which is elaborated in the extended context $C + \{modid \mapsto M\}$, specifies the result of applying such a functor.

If $sigexp_2$ elaborates to a signature $\Lambda T'.M'$, then any type names in T' represent types that have an unspecified realisation in $sigexp_2$. Because the types declared in the body of a matching functor may depend on the functor's type parameters, the rule allows type names in T' to have a functional dependency on the type parameters in T. Applying the realisation φ to M' caters for these dependencies. The realisation parameterises each occurrence in M' of a name $\mathrm{u} \in T'$ by the names in T. The kinds of names in T' must be adjusted to reflect this, resulting in the name set T''. Having modified names in T' to take account of their implicit dependencies on T, the scope of the parameterisation over T' can be extended from the range, i.e. $\Lambda T'.M'$, to a scope that encloses the entire functor, yielding the signature $\Lambda T''.(\forall T.M \to \exists\emptyset.\varphi(M'))$.

In this way, the type parameters arising *sigexp*$_1$ determine the polymorphism of the specified functors, while the type parameters arising from *sigexp*$_2$ index variations in the range of the specified functors. These parameters represent unspecified argument-result type dependencies.

Specifications

$$\boxed{C \vdash \mathit{spec} \Rightarrow \Lambda T.E}$$

$$\frac{C \vdash \mathit{tyidseq} \Rightarrow (\alpha_1, \cdots, \alpha_k), \mathit{IE} \quad C + \mathit{IE} \vdash \mathit{valdesc} \Rightarrow \mathit{VE}}{C \vdash \texttt{val}\ \mathit{tyidseq}\ \mathit{valdesc} \Rightarrow \Lambda\emptyset.\mathrm{Clos}_C \mathit{VE} \text{ in Env}} \tag{95}$$

$$\frac{C \vdash \mathit{typdesc} \Rightarrow \Lambda T.\mathit{TE} \quad \forall(\mathrm{t}, \mathit{VE}) \in T,\ \mathrm{t} \text{ does not admit equality}}{C \vdash \texttt{type}\ \mathit{typdesc} \Rightarrow \Lambda T.\mathit{TE} \text{ in Env}} \tag{96}$$

$$\frac{C \vdash \mathit{typdesc} \Rightarrow \Lambda T.\mathit{TE} \quad \forall(\mathrm{t}, \mathit{VE}) \in T,\ \mathrm{t} \text{ admits equality}}{C \vdash \texttt{eqtype}\ \mathit{typdesc} \Rightarrow \Lambda T.\mathit{TE} \text{ in Env}} \tag{97}$$

$$\frac{\begin{array}{l} C + \mathit{TE} \vdash \mathit{datdesc} \Rightarrow \mathit{VE}, \mathit{TE} \\ T = \{\mathrm{t}; (\mathrm{t}, \mathit{VE}') \in \mathrm{Ran}\, \mathit{TE}\} \\ T \cap \mathrm{tynames}\, C = \emptyset \\ \mathit{TE} \text{ maximises equality} \end{array}}{C \vdash \texttt{datatype}\ \mathit{datdesc} \Rightarrow \Lambda T.(\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{98}$$

$$\frac{\begin{array}{c} C \vdash \mathit{tyconpath} \Rightarrow (\theta, \mathit{VE}) \\ \mathit{TE} = \{\mathit{tycon} \mapsto (\theta, \mathit{VE})\} \end{array}}{\begin{array}{c} C \vdash \texttt{datatype}\ \mathit{tycon}\ \texttt{=}\ \texttt{datatype}\ \mathit{tyconpath} \Rightarrow \\ \Lambda\emptyset.(\mathit{VE}, \mathit{TE}) \text{ in Env} \end{array}} \tag{99}$$

$$\frac{C \vdash \mathit{exdesc} \Rightarrow \mathit{VE}}{C \vdash \texttt{exception}\ \mathit{exdesc} \Rightarrow \Lambda\emptyset.\mathit{VE} \text{ in Env}} \tag{100}$$

$$\frac{C \vdash \mathit{strdesc} \Rightarrow \Lambda T.\mathit{SE}}{C \vdash \texttt{structure}\ \mathit{strdesc} \Rightarrow \Lambda T.\mathit{SE}\ \text{in Env}} \tag{101}$$

$$\frac{C \vdash \mathit{fundesc} \Rightarrow \Lambda T.\mathit{FE}}{C \vdash \texttt{functor}\ \mathit{fundesc} \Rightarrow \Lambda T.\mathit{FE}\ \text{in Env}} \tag{102}$$

$$\frac{C \vdash \mathit{sigbind} \Rightarrow \mathit{GE}}{C \vdash \texttt{signature}\ \mathit{sigbind} \Rightarrow \Lambda\emptyset.\mathit{GE}\ \text{in Env}} \tag{103}$$

$$\frac{C \vdash \mathit{sigexp} \Rightarrow \Lambda T.(\mathit{FE}, \mathit{SE}, \mathit{TE}, \mathit{VE})}{C \vdash \texttt{include}\ \mathit{sigexp} \Rightarrow \Lambda T.(\{\}, \mathit{FE}, \mathit{SE}, \mathit{TE}, \mathit{VE})} \tag{104}$$

$$\frac{}{C \vdash \quad \Rightarrow \Lambda\emptyset.(\{\},\{\},\{\},\{\},\{\})} \tag{105}$$

$$\frac{\begin{array}{ll} C \vdash \mathit{spec}_1 \Rightarrow \Lambda T_1.E_1 & T_1 \cap \text{tynames}\, C = \emptyset \\ C + E_1 \vdash \mathit{spec}_2 \Rightarrow \Lambda T_2.E_2 & \text{Dom}\, E_1 \cap \text{Dom}\, E_2 = \emptyset \\ T_2 \cap (T_1 \cup \text{tynames}\, E_1) = \emptyset & \end{array}}{C \vdash \mathit{spec}_1\ \langle\texttt{;}\rangle\ \mathit{spec}_2 \Rightarrow \Lambda T_1 \cup T_2.E_1 + E_2} \tag{106}$$

$$\frac{\begin{array}{c} C \vdash \mathit{spec} \Rightarrow \Lambda T.E \\ (\{\}, E) \vdash \mathit{longtycon}_i \Rightarrow (\mathrm{t}_i, \mathit{VE}_i),\ i = 1..n \\ \mathrm{t}_1, \ldots, \mathrm{t}_n \text{ have arity } k \\ \mathrm{t} \in \{\mathrm{t}_1, \ldots, \mathrm{t}_n\} \quad \mathrm{t} \text{ admits equality, if some } \mathrm{t}_i \text{ does} \\ \{\mathrm{t}_1, \ldots, \mathrm{t}_n\} \subseteq T \qquad \varphi = \{\mathrm{t}_1 \mapsto \mathrm{t}, \ldots, \mathrm{t}_n \mapsto \mathrm{t}\} \end{array}}{\begin{array}{c} C \vdash \mathit{spec}\ \texttt{sharing type}\ \mathit{longtycon}_1\ \texttt{=} \cdots \texttt{=}\ \mathit{longtycon}_n \Rightarrow \\ \Lambda T \setminus (\{\mathrm{t}_1, \ldots, \mathrm{t}_n\} \setminus \{\mathrm{t}\}).\varphi(E) \end{array}} \tag{107}$$

Comments:

(95) VE is determined by C and *valdesc*.

(96)–(98) The type names in T are bound and thus parameters of the resulting signature.

(100) VE is determined by C and *exdesc* and contains monotypes only.

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

Value Descriptions

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

$$\frac{C \vdash ty \Rightarrow \tau \qquad \langle C \vdash valdesc \Rightarrow VE \rangle}{C \vdash vid\ \texttt{:}\ ty\ \langle \texttt{and}\ valdesc \rangle \Rightarrow \{vid \mapsto (\tau, \texttt{v})\}\ \langle +\ VE \rangle} \tag{108}$$

Type Descriptions

$$\boxed{C \vdash typdesc \Rightarrow \Lambda T.TE}$$

$$\frac{\begin{array}{c} C \vdash tyidseq \Rightarrow (\alpha^{(k)}, VE) \\ \mathrm{t}\ \text{has arity}\ k \\ \langle\ C \vdash typdesc \Rightarrow \Lambda T.TE \quad \mathrm{t} \notin T\ \rangle \end{array}}{C \vdash tyidseq\ tycon\ \langle \texttt{and}\ typdesc \rangle \Rightarrow \Lambda\{\mathrm{t}\}\langle \cup T \rangle.\{tycon \mapsto (\mathrm{t}, \{\})\}\langle +TE \rangle} \tag{109}$$

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

Datatype Descriptions

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

$$\frac{\begin{array}{c} C \vdash tyidseq \Rightarrow (\alpha^{(k)}, IE) \\ C + IE, \alpha^{(k)}\ \mathrm{t} \vdash condesc \Rightarrow VE \\ \langle C \vdash datdesc \Rightarrow VE', TE' \qquad \forall (\mathrm{t}', VE'') \in \mathrm{Ran}\, TE, \mathrm{t} \neq \mathrm{t}' \rangle \end{array}}{\begin{array}{l} C \vdash tyidseq\ tycon\ \texttt{=}\ condesc\ \langle \texttt{and}\ datdesc \rangle \Rightarrow \\ \qquad \mathrm{Clos}_C VE \langle +\ VE' \rangle, \{tycon \mapsto (\mathrm{t}, \mathrm{Clos}_C VE)\}\ \langle +\ TE' \rangle \end{array}} \tag{110}$$

Constructor Descriptions

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

$$\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash condesc \Rightarrow VE \rangle\rangle}{\begin{array}{l} C, \tau \vdash vid\ \langle \texttt{of}\ ty \rangle\ \langle\langle\ \texttt{|}\ condesc \rangle\rangle \Rightarrow \\ \qquad \{vid \mapsto (\tau, \texttt{c})\}\ \langle +\ \{vid \mapsto (\tau' \to \tau, \texttt{c})\}\ \rangle\ \langle\langle +\ VE \rangle\rangle \end{array}} \tag{111}$$

Exception Descriptions

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

$$\frac{\langle C \vdash ty \Rightarrow \tau \rangle \qquad \langle\langle C \vdash exdesc \Rightarrow VE \rangle\rangle}{\begin{array}{l} C \vdash vid\ \langle \texttt{of}\ ty \rangle\ \ \langle\langle \texttt{and}\ exdesc \rangle\rangle \Rightarrow \\ \qquad \{vid \mapsto (\texttt{exn}, \texttt{e})\}\ \langle +\ \{vid \mapsto (\tau \to \texttt{exn}, \texttt{e})\} \rangle\ \langle\langle +\ VE \rangle\rangle \end{array}} \tag{112}$$

Structure Descriptions

$$\boxed{C \vdash \mathit{strdesc} \Rightarrow \Lambda T.\mathit{SE}}$$

$$\frac{\begin{array}{l} C \vdash \mathit{sigexp} \Rightarrow \Lambda T.\mathit{RS} \\ \langle C \vdash \mathit{strdesc} \Rightarrow \Lambda T'.\mathit{SE} \rangle \\ \langle T \cap (T' \cup \mathrm{tynames}\,\mathit{SE}) = \emptyset \rangle \\ \langle T' \cap (\mathrm{tynames}\,\mathit{RS}) = \emptyset \rangle \end{array}}{\begin{array}{c} C \vdash \mathit{strid} \ : \ \mathit{sigexp} \ \langle \texttt{and} \ \mathit{strdesc} \rangle \Rightarrow \\ \Lambda T \langle \cup T' \rangle . \{ \mathit{strid} \mapsto \mathit{RS} \} \ \langle + \ \mathit{SE} \rangle \end{array}} \tag{113}$$

Functor Descriptions

$$\boxed{C \vdash \mathit{fundesc} \Rightarrow \Lambda T.\mathit{FE}}$$

$$\frac{\begin{array}{l} C \vdash \mathit{sigexp} \Rightarrow \Lambda T.F \\ \langle C \vdash \mathit{fundesc} \Rightarrow \Lambda T'.\mathit{FE} \rangle \\ \langle T \cap (T' \cup \mathrm{tynames}\,\mathit{FE}) = \emptyset \rangle \\ \langle T' \cap (\mathrm{tynames}\,F) = \emptyset \rangle \end{array}}{\begin{array}{c} C \vdash \mathit{funid} \ : \ \mathit{sigexp} \ \langle \texttt{and} \ \mathit{fundesc} \rangle \Rightarrow \\ \Lambda T \langle \cup T' \rangle . \{ \mathit{funid} \mapsto F \} \ \langle + \ \mathit{FE} \rangle \end{array}} \tag{114}$$

A Appendix: Derived Forms

Several derived grammatical forms are provided in the Core; they are presented in Figures 14, 15 and 16. Each derived form is given with its equivalent form. Thus, each row of the tables should be considered as a rewriting rule

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

and these rules may be applied repeatedly to a phrase until it is transformed into a phrase of the bare language. See Appendix **??** 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` *tyidseq fvalbind* . The mixed forms `val` *tyidseq* `rec` *fvalbind* , `val` *tyidseq fvalbind* and `fun` *tyidseq valbind* are not allowed – though the first form arises during translation into the bare language.

The following notes refer to Figure 16:

- There is a version of the derived form for function-value binding which allows the function identifier to be infixed; see Figure **??** in Appendix **??**.

- 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

  $$tyidseq_1\ tycon_1 \texttt{=} ty_1\ \texttt{and} \cdots \texttt{and}\ tyidseq_n\ tycon_n \texttt{=} ty_n$$

  then *datbind′* is the result of simultaneous replacement (in *datbind*) of every type expression $tyseq_i\ tycon_i$ $(1 \leq i \leq n)$ by the corresponding defining expression

  $$ty_i\{tyseq_i/tyidseq_i\}$$

Figure 17 shows derived forms for functors. They allow functors to take, say, a single type or value as a parameter, in cases where it would seem clumsy to "wrap up" the argument as a structure expression.

Finally, Figure 18 shows the derived forms for specifications and signature expressions. The last derived form for specifications allows sharing between structure identifiers as a shorthand for type sharing specifications. The phrase

$$\textit{spec}\ \texttt{sharing}\ \textit{longstrid}_1\ \texttt{=}\ \cdots\ \texttt{=}\ \textit{longstrid}_k$$

is a derived form whose equivalent form is

spec
 `sharing type` $\textit{longtycon}_1$ `=` $\textit{longtycon}'_1$
 $\cdots$
 `sharing type` $\textit{longtycon}_m$ `=` $\textit{longtycon}'_m$

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

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

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

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

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

`while` exp_1 `do` exp_2	`let val rec` *vid* `= fn () =>`
	 `if` exp_1 `then (`exp_2`;`*vid*`()) else ()`
	 `in` *vid*`() end`
		(*vid* new)

`[`exp_1 `,` $\cdots$ `,` exp_n`]`	exp_1 `::` $\cdots$ `::` exp_n `:: nil`	$(n \geq 0)$

Figure 14: Derived forms of Expressions

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

Figure 15: Derived forms of Patterns and Type Expressions

Derived Form	Equivalent Form

Function-value Bindings *fvalbind*

$\langle$`op`$\rangle$*vid* $atpat_{11} \cdots atpat_{1n} \langle$`:` $ty\rangle$ `=` exp_1 `|`$\langle$`op`$\rangle$*vid* $atpat_{21} \cdots atpat_{2n} \langle$`:` $ty\rangle$ `=` exp_2 `|` $\cdots$ $\cdots$ `|`$\langle$`op`$\rangle$*vid* $atpat_{m1} \cdots atpat_{mn} \langle$`:` $ty\rangle$ `=` exp_m $\langle$`and` *fvalbind*$\rangle$	$\langle$`op`$\rangle$*vid* `=` `fn` vid_1`=>` $\cdots$ `fn` vid_n`=>` `case` `(`vid_1`,` $\cdots$ `,` vid_n`)` `of` `(`$atpat_{11}$`,`$\cdots$`,`$atpat_{1n}$ `)=>`$exp_1\langle$`:` $ty\rangle$ `|(`$atpat_{21}$`,`$\cdots$`,`$atpat_{2n}$ `)=>`$exp_2\langle$`:` $ty\rangle$ `|` $\cdots$ $\cdots$ `|(`$atpat_{m1}$`,`$\cdots$`,`$atpat_{mn}$ `)=>`$exp_m\langle$`:` $ty\rangle$ $\langle$`and` *fvalbind*$\rangle$

($m, n \geq 1$; $vid_1, \cdots, vid_n$ distinct and new)

Declarations *dec*

`fun` *tyidseq fvalbind*	`val` *tyidseq* `rec` *fvalbind*
`datatype` *datbind* `withtype` *typbind*	`datatype` *datbind′* `;` `type` *typbind*
`abstype` *datbind* `withtype` *typbind* `with` *dec* `end`	`abstype` *datbind′* `with` `type` *typbind* `;` *dec* `end`

(see note in text concerning *datbind′*)

Figure 16: Derived forms of Function-value Bindings and Declarations

Derived Form	Equivalent Form

Structure Bindings *strbind*

strid `:` *sigexp* `=` *modexp* ⟨`and` *strbind*⟩	*strid* `=` *modexp* `:` *sigexp* ⟨`and` *strbind*⟩
strid `:>` *sigexp* `=` *modexp* ⟨`and` *strbind*⟩	*strid* `=` *modexp* `:>` *sigexp* ⟨`and` *strbind*⟩

Module Expressions *modexp*

`(` *dec* `)`	`( struct` *dec* `end )`

Functor Bindings *funbind*

funid $\langle(\rangle_1 modid_1 : sigexp_1 \langle)\rangle_1$ $\vdots$ $\langle(\rangle_n modid_n : sigexp_n \langle)\rangle_n$ `=` *modexp* ⟨`and` *funbind*⟩	*funid* `= functor` $\langle(\rangle_1 modid_1 : sigexp_1 \langle)\rangle_1$ `=>` $\vdots$ `functor` $\langle(\rangle_n modid_n : sigexp_n \langle)\rangle_n$ `=>` *modexp* ⟨`and` *funbind*⟩

funid $\langle(\rangle_1 modid_1 : sigexp_1 \langle)\rangle_1$
$\vdots$
$\langle(\rangle_m modid_m : sigexp_m \langle)\rangle_m$
`:` $sigexp'$ `=` *modexp*
⟨`and` *funbind*⟩
	funid `= functor` $\langle(\rangle_1 modid_1 : sigexp_1 \langle)\rangle_1$ `=>`
	$\vdots$
	`functor` $\langle(\rangle_m modid_m : sigexp_m \langle)\rangle_m$ `=>`
	`(`*modexp*`:`$sigexp'$`)`
	⟨`and` *funbind*⟩

funid $\langle(\rangle_1 modid_1 : sigexp_1 \langle)\rangle_1$
$\vdots$
$\langle(\rangle_m modid_m : sigexp_m \langle)\rangle_m$
`:>` $sigexp'$ `=` *modexp*
⟨`and` *funbind*⟩
	funid `= functor` $\langle(\rangle_1 modid_1 : sigexp_1 \langle)\rangle_1$ `=>`
	$\vdots$
	`functor` $\langle(\rangle_m modid_m : sigexp_m \langle)\rangle_m$ `=>`
	`(`*modexp*`:>`$sigexp'$`)`
	⟨`and` *funbind*⟩

funid `(` *spec* `)` ⟨`:` *sigexp*⟩ `=` *modexp* ⟨`and` *funbind*⟩	*funid* `(` $strid_\nu$ `: sig` *spec* `end ) =` `let open` $strid_\nu$ `in` *modexp*⟨`:` *sigexp*⟩ `end` ⟨`and` *funbind*⟩
funid `(` *spec* `)` ⟨`:>` *sigexp*⟩ `=` *modexp* ⟨`and` *funbind*⟩	*funid* `(` $strid_\nu$ `: sig` *spec* `end ) =` `let open` $strid_\nu$ `in` *modexp*⟨`:>`*sigexp*⟩ `end` ⟨`and` *funbind*⟩

($n \geq 1$, $m \geq 0$, $strid_\nu$ new)

Programs *program*

exp `;`⟨*program*⟩	`val it =` *exp* `;`⟨*program*⟩

Figure 17: Derived forms of Functors, Structure Bindings and Programs

Derived Form	Equivalent Form

Specifications *spec*

`type` *tyidseq* *tycon* = *ty*	`include` `sig type` *tyidseq* *tycon* `end where type` *tyidseq* *tycon* = *ty*
`type` $tyidseq_1$ $tycon_1$ `=` ty_1 `and` $\cdots$ $\cdots$ `and` $tyidseq_n$ $tycon_n$ `=` ty_n	`type` $tyidseq_1$ $tycon_1$ `=` ty_1 `type` $\cdots$ $\cdots$ `type` $tyidseq_n$ $tycon_n$ `=` ty_n
`include` $sigid_1 \cdots sigid_n$	`include` $sigid_1$; $\cdots$; `include` $sigid_n$

spec `sharing` $longstrid_1$ `=` $\cdots$
`=` $longstrid_k$
	spec
	 `sharing type` $longtycon_1$ `=`
	$longtycon'_1$
	 $\cdots$
	 `sharing type` $longtycon_m$ `=`
	$longtycon'_m$

(see note in text concerning $longtycon_1, \ldots, longtycon'_m$)

Signature Expressions *sigexp*

sigexp
`where type` $tyidseq_1$ $longtycon_1$ `=` ty_1
  `and type` $\cdots$
  $\cdots$
  `and type` $tyidseq_n$ $longtycon_n$ `=` ty_n
	sigexp
	 `where type` $tyidseq_1$ $longtycon_1$ `=` ty_1
	 `where type` $\cdots$
	 $\cdots$
	 `where type` $tyidseq_n$ $longtycon_n$ `=` ty_n

Figure 18: Derived forms of Specifications and Signature Expressions