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 1

2 Syntax of the Core 2
   2.1 Reserved Words ................................. 2
   2.2 Special constants ............................... 2
   2.3 Comments ...................................... 3
   2.4 Identifiers .......................... 4
   2.5 Lexical analysis ............................... 5
   2.6 Infixed operators .............................. 5
   2.7 Derived Forms .................................. 7
   2.8 Grammar ...................................... 7
   2.9 Syntactic Restrictions ......................... 11

3 Syntax of Modules 13
   3.1 Reserved Words ................................. 13
   3.2 Identifiers .................................... 13
   3.3 Infixed operators .............................. 13
   3.4 Grammar for Modules ......................... 14
   3.5 Syntactic Restrictions ......................... 14

4 Scope of Explicit Type Identifiers 18

5 Static Semantics for the Core 20
   5.1 Simple Objects ................................. 20
   5.2 Compound Objects .............................. 21
   5.3 Projection, Injection and Modification .......... 21
   5.4 Types, Type Applications and Type functions .... 22
   5.5 Type Schemes .................................. 24
   5.6 Non-expansive Expressions ..................... 24
   5.7 Closure ..................................... 25
   5.8 Existential and Parameterised Objects .......... 26
   5.9 Type Structures and Type Environments .......... 26
   5.10 Inference Rules ................................ 27
   5.11 Further Restrictions ............................ 37
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)
2.3 Comments

\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).
\xxxx The single character with number xxxx (4 hexadecimal digits denoting an integer in the ordinal range of the alphabet).
\" "
\f··f\ This sequence is ignored, where f··f stands for a sequence of one or more formatting characters.

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

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
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:

\[
\text{longx ::= } x \quad \text{identifier} \\
\quad \text{longstrid}.x \quad \text{qualified identifier (n ≥ 1)}
\]

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 ..., 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 \texttt{op}, where permitted, has no effect.\footnote{In Modules, \texttt{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.} Infix
status is cancelled by the \texttt{nonfix} directive. We refer to the three directives collectively as \textit{fixity directives}.

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

\begin{align*}
\text{infix} \langle d \rangle \text{ vid}_1 \cdots \text{ vid}_n \\
\text{infixr} \langle d \rangle \text{ vid}_1 \cdots \text{ vid}_n \\
\text{nonfix} \text{ vid}_1 \cdots \text{ vid}_n
\end{align*}

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. \texttt{infix} and
\texttt{infixr} dictate left and right associativity respectively. In an expression of
the form $\text{exp}_1 \text{ vid}_1 \text{ exp}_2 \text{ vid}_2 \text{ exp}_3$, where $\text{vid}_1$ and $\text{vid}_2$ 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

\begin{align*}
x \ << \ y \ << \ z \ & \text{ parses as } (x \ << \ y) \ << \ z \\
x \ >> \ y \ >> \ z \ & \text{ parses as } x \ >> \ (y \ >> \ z) \\
x \ << \ y \ >> \ z \ & \text{ is illegal} \\
x \ >> \ y \ << \ z \ & \text{ is illegal}
\end{align*}

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

The \textit{scope} of a fixity directive \texttt{dir} is the ensuing program text, except
that if \texttt{dir} occurs in a declaration \texttt{dec} in either of the phrases

\begin{align*}
\text{let } \texttt{dec} \ \text{in } \cdots \ \text{end} \\
\text{local } \texttt{dec} \ \text{in } \cdots \ \text{end}
\end{align*}

then the scope of \texttt{dir} does not extend beyond the phrase. Further scope
limitations are imposed for Modules.

These directives and \texttt{op} are omitted from the Core semantic rules, since
they affect only parsing.
2.7 Derived Forms

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

2.8 Grammar

The phrase classes for the Core are shown in Figure 2. We use the variable $\texttt{atexp}$ to range over $\texttt{AtExp}$, 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 $X_{seq}$ (over which $x_{seq}$ ranges) as follows:

$$x_{seq} ::= x$$

(singleton sequence)

(empty sequence)

$$(x_1, \ldots, x_n)$$

(sequence, $n \geq 1$)

(Note that the “$\ldots$” used here, meaning syntactic iteration, must not be confused with “\ldots” 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. $\text{match}$, $\ldots$) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. “$\text{fn match}$”, if this occurs within a larger match.
\[ \text{atexp} ::= \text{scon} \quad \text{special constant} \\
\langle \text{op} \rangle \text{longvid} \quad \text{value identifier} \\
\{ \langle \text{exprow} \rangle \} \quad \text{record} \\
\text{let dec in exp end} \quad \text{local declaration} \\
[\text{structure modexp as sigexp}] \quad \text{structure package} \\
[\text{functor modexp as sigexp}] \quad \text{functor package} \\
( \text{exp} ) \]

\[ \text{exprow} ::= \text{lab} = \text{exp} ( , \text{exprow} ) \quad \text{expression row} \\
\text{exp} ::= \text{atexp} \quad \text{atomic} \\
\text{exp atexp} \quad \text{application (L)} \\
\text{exp} \text{ vid } \text{exp}_2 \quad \text{infixed application} \\
\text{exp} : \text{ty} \quad \text{typed (L)} \\
\text{exp handle match} \quad \text{handle exception} \\
\text{raise exp} \quad \text{raise exception} \\
\text{fn match} \quad \text{function} \\
\text{match} ::= \text{mrule} ( \mid \text{match} ) \\
\text{mrule} ::= \text{pat } \Rightarrow \text{exp} \]

Figure 3: Grammar: Expressions and Matches
\[
dec ::= \text{val}\ tyidseq\ valbind \quad \text{value declaration}
\text{type}\ tybind \quad \text{type declaration}
\text{datatype}\ datbind \quad \text{datatype declaration}
\text{datatype}\ tycon = \text{datatype}\ tyconpath \quad \text{datatype replication}
\text{abstype}\ datbind\ \text{with}\ dec\ \text{end} \quad \text{abstype declaration}
\text{exception}\ exbind \quad \text{exception declaration}
\text{local}\ dec_1\ \text{in}\ dec_2\ \text{end} \quad \text{local declaration}
\text{open}\ longstrid_1 \cdots longstrid_n \quad \text{open declaration} \quad (n \geq 1)
\text{structure}\ strbind \quad \text{structure declaration}
\text{functor}\ funbind \quad \text{functor declaration}
\text{signature}\ sigbind \quad \text{signature declaration}
\text{empty}\ dec_1\ (;)\ dec_2 \quad \text{sequential declaration}
\text{infix}\langle d\rangle\ vid_1 \cdots vid_n \quad \text{infix (L) directive}
\text{infixr}\langle d\rangle\ vid_1 \cdots vid_n \quad \text{infix (R) directive}
\text{nonfix}\ vid_1 \cdots vid_n \quad \text{nonfix directive}
\\]

\[
\text{valbind ::= pat = exp \langle and\ valbind\rangle}
\text{rec\ valbind}
\text{tybind ::= tyidseq\ tycon = ty \langle and\ tybind\rangle}
\text{datbind ::= tyidseq\ tycon = conbind \langle and\ datbind\rangle}
\text{conbind ::= \langle op\rangle vid \langle of\ ty\rangle \langle 1\ conbind\rangle}
\text{exbind ::= \langle op\rangle vid \langle of\ ty\rangle \langle and\ exbind\rangle}
\text{\langle op\rangle vid = \langle op\rangle longvid \langle and\ exbind\rangle}
\\]

Figure 4: Grammar: Declarations and Bindings
2.9 Syntactic Restrictions

- No expression row, pattern row or type row may bind the same `lab` twice.

- No binding `valbind`, `tybind`, `datbind` or `exbind` may bind the same identifier twice; this applies also to value 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.

\[
\text{eqtype include sharing sig struct :>}
\]

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 \texttt{dir} occurs in a declaration \texttt{dec} in any of the phrases

\[
\begin{align*}
\text{let } & \texttt{dec} \texttt{ in } \cdots \texttt{ end} \\
\text{struct } & \texttt{dec} \texttt{ end} \\
\text{sig } & \texttt{spec} \texttt{ end}
\end{align*}
\]

then the scope of \texttt{dir} does not extend beyond the phrase.
One effect of this limitation is that fixity is local to a basic structure expression – in particular, to such an expression occurring as a functor body. Similarly, fixity is local to a basic signature expression.

Fixity directives (but not \texttt{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 \texttt{atmodexp} to range over \texttt{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 \texttt{strbind}, \texttt{funbind}, or \texttt{sigbind} may bind the same identifier twice.
3.5 Syntactic Restrictions

\[
\text{atmodexp} ::= \text{struct dec end} \quad \text{basic}
\]
\[
\langle \text{op} \rangle \text{longmodid} \quad \text{module identifier}
\]
\[
\text{let dec in modexp end} \quad \text{local declaration}
\]
\[
\text{modexp} ::= \text{atmodexp} \quad \text{atomic}
\]
\[
\text{modexp atmmodexp} \quad \text{functor application}
\]
\[
\text{modexp : sigexp} \quad \text{transparent constraint}
\]
\[
\text{modexp :> sigexp} \quad \text{opaque constraint}
\]
\[
\text{functor (modid: sigexp) => modexp} \quad \text{generative functor}
\]
\[
\text{functor modid: sigexp => modexp} \quad \text{applicative functor}
\]
\[
\text{rec (strid: sigexp) modexp} \quad \text{recursive structure}
\]
\[
\text{strbind} ::= \text{strid = modexp (and strbind)} \quad \text{structure binding}
\]
\[
\text{strid as sigexp = exp (and strbind)} \quad \text{package binding}
\]
\[
\text{funbind} ::= \text{funid = modexp (and funbind)} \quad \text{functor binding}
\]
\[
\text{funid as sigexp = exp (and funbind)} \quad \text{package binding}
\]
\[
\text{sigbind} ::= \text{sigid = sigexp (and sigbind)} \quad \text{basic}
\]
\[
\text{sigexp} ::= \text{sig spec end} \quad \text{signature identifier}
\]
\[
\text{sigid} \quad \text{type realisation}
\]
\[
\text{sigid} \quad \text{tyids seq longtycon = ty} \quad \text{opaque functor signature}
\]
\[
\text{functor (modid : sigexp) \rightarrow sigexp} \quad \text{transparent functor signature}
\]
\[
\text{functor modid : sigexp \rightarrow sigexp} \quad \text{recursive structure signature}
\]

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`. 
3.5 Syntactic Restrictions

\[
\begin{align*}
\text{spec} & ::= \text{val tyidseq valdesc} & \text{value} \\
\text{type} & ::= \text{tyidseq} & \text{type} \\
\text{eqtype} & ::= \text{tyidseq} & \text{eqtype} \\
\text{datatype} & ::= \text{tyidseq datdesc} & \text{datatype} \\
\text{datatype} & ::= \text{tycon} = \text{datatype tyconpath} & \text{replication} \\
\text{exception} & ::= \text{exdesc} & \text{exception} \\
\text{structure} & ::= \text{strdesc} & \text{structure} \\
\text{functor} & ::= \text{fundesc} & \text{functor} \\
\text{signature} & ::= \text{sigbind} & \text{signature} \\
\text{include} & ::= \text{sigexp} & \text{include} \\
\text{empty} & ::= \text{spec} (\_\_\_\_) & \text{empty} \\
\text{sequential} & ::= \text{spec} (\_\_\_) \text{spec} & \text{sequential} \\
\text{sharing} & ::= \text{spec} (\_\_\_\_\_\_) & \text{sharing} \\
\text{spec1} & ::= \text{spec} (\_\_\_\_\_\_) & \text{spec2} \\
\text{spec1} & ::= \text{spec} (\_\_\_\_\_\_\_) & \text{spec2} \\
\text{longtycon} & ::= \text{longtycon1} = \cdots = \text{longtycon_n} & (n \geq 2) \\
\text{infix} & ::= \langle d \rangle \text{ vid1} \cdots \text{ vid_n} & \text{infix (L) directive} \\
\text{infixr} & ::= \langle d \rangle \text{ vid1} \cdots \text{ vid_n} & \text{infix (R) directive} \\
\text{nonfix} & ::= \langle d \rangle \text{ vid1} \cdots \text{ vid_n} & \text{nonfix directive} \\
\text{valdesc} & ::= \langle d \rangle \text{ vid} : \text{ ty} (\_\_\_\_\_) \text{ valdesc} & \text{valdesc} \\
\text{typdesc} & ::= \langle d \rangle \text{ tyidseq tycon} (\_\_\_\_\_\_\_\_) \text{ typdesc} & \text{typdesc} \\
\text{datdesc} & ::= \langle d \rangle \text{ tyidseq tycon} = \text{condesc} (\_\_\_\_\_\_\_\_) \text{ datdesc} & \text{datdesc} \\
\text{condesc} & ::= \langle d \rangle \text{ vid} (\_\_\_\_\_\_\_\_) (\_\_\_\_\_\_\_\_) \text{ condesc} & \text{condesc} \\
\text{exdesc} & ::= \langle d \rangle \text{ vid} (\_\_\_\_\_\_\_\_) (\_\_\_\_\_\_\_\_) \text{ exdesc} & \text{exdesc} \\
\text{strdesc} & ::= \langle d \rangle \text{ strid} : \text{ sigexp} (\_\_\_\_\_\_\_\_) \text{ strdesc} & \text{strdesc} \\
\text{fundesc} & ::= \langle d \rangle \text{ funid} : \text{ sigexp} (\_\_\_\_\_\_\_\_) \text{ fundesc} & \text{fundesc}
\end{align*}
\]

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. Similarly, 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 \( \alpha \) 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 \( \alpha \) occurs unguarded in the phrase.

Then we say that \( \alpha \) is implicitly scoped at a particular value declaration `val tyidseq valbind` or value specification `val tyidseq valdesc` in a program if (1) \( \alpha \) occurs unguarded in this phrase, and (2) \( \alpha \) 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::` 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

\[
\begin{align*}
\text{val } x &= \text{let } \text{id}:'a\to'a = \text{fn } z => z \text{ in id id end} \\
\text{val } x &= (\text{let } \text{id}:'a\to'a = \text{fn } z => z \text{ in id id end}; \text{fn } z => z:'a)
\end{align*}
\]

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

\[
\begin{align*}
\text{val } x &= \text{let } \text{id}:'a\to'a = \text{fn } z => z \text{ in id id end} \\
\text{val } 'a x &= (\text{let } \text{id}:'a\to'a = \text{fn } z => z \text{ in id id end}; \text{fn } z => z:'a)
\end{align*}
\]

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 \texttt{id} in the application \texttt{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.

\[
\begin{align*}
\alpha &\in \text{TyVar} & \text{type variables} \\
t &\in \text{TyName} & \text{type names} \\
is &\in \text{IdStatus}=\{c, e, v\} & \text{identifier status descriptors}
\end{align*}
\]

Figure 9: Simple Semantic Objects

Each \(\alpha\in\text{TyVar}\) possesses a boolean equality attribute, which determines whether or not it admits equality. Each \(t\in\text{TyName}\) has a kind \(K\in\text{Kind}\) (defined in Figure 10). We denote the class of type names with kind \(K\) by \(\text{TyName}^K\), letting \(t^K\) range over elements of \(\text{TyName}^K\). 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 \(\text{type}(scon)\) which is either \text{int, real, word, char} or \text{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 $\text{Fin} A$ denotes the set of finite subsets of $A$, and $A \rightarrow B$ denotes the set of finite maps (partial functions with finite domain) from $A$ to $B$. The domain and range of a finite map, $f$, are denoted $\text{Dom} f$ and $\text{Ran} f$. A finite map will often be written explicitly in the form \{ $a_1 \mapsto b_1, \ldots, a_k \mapsto b_k$, $k \geq 0$; 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 $\phi$, and whose value on this domain is given by $f(x) = e$.

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

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

The compound objects for the static semantics of the Core Language are shown in 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 $\lambda$ and $\forall$ bind type variables. For any semantic object $A$, $\text{tynames} A$ and $\text{tyvars} A$ denote respectively the set of type names and the set of type variables occurring free in $A$.

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

**Injection:** Components may be injected into tuple classes; for example, “$\text{VE in Env}$” means the environment $\{\}, \{\}, \{\}, \{\}, \text{VE}$.
\[ k \in \text{Arity} = \{k; k \geq 0\} \]
\[ K \text{ or } k \text{ or } k^= \text{ or } K \rightarrow K' \in \text{Kind} = \text{Arity} \cup \text{Arity} \cup (\text{Kind} \times \text{Kind}) \]
\[ \tau \in \text{Type} = \text{TyVar} \cup \text{RecType} \cup \text{FunType} \cup \text{ConsType} \cup \text{PackType} \]
\[ (\tau_1, \cdots, \tau_k) \text{ or } \tau^{(k)} \in \text{Type}^{(k)} \]
\[ (\alpha_1, \cdots, \alpha_k) \text{ or } \alpha^{(k)} \in \text{TyVar}^{(k)} \]
\[ \rho \in \text{RecType} = \text{Lab} \rightarrow \text{Type} \]
\[ \tau \rightarrow \tau' \in \text{FunType} = \text{Type} \times \text{Type} \]
\[ \text{ConsType} = \bigcup_{k \in \text{Arity}} \text{ConsType}^k \]
\[ \tau^{(k)} \vartheta^k \in \text{ConsType}^k = \text{Type}^{(k)} \times \text{TypeApp}^k \]
\[ X \in \text{PackType} = \text{ExMod} \]
\[ \theta^k \text{ or } \Lambda \alpha^{(k)} \cdot \tau \text{ or } \vartheta^k \in \text{TypeFcn}^k = \]
\[ (\text{TyVar}^k \times \text{Type}) \cup \text{TypeApp}^k \]
\[ \theta^k^= \in \text{TypeFcn}^{k^=} = \]
\[ \{\theta^k \in \text{TypeFcn}^k; \theta^k \text{ admits equality}\} \]
\[ \theta^K \rightarrow K' \text{ or } \Lambda \alpha^K \cdot \theta^K' \text{ or } \vartheta^K \rightarrow K' \in \text{TypeFcn}^{K \rightarrow K'} \]
\[ = (\text{TyName}^K \times \text{TypeFcn}^{K'}) \cup \text{TypeApp}^{K \rightarrow K'} \]
\[ \vartheta^K \text{ or } t^K \text{ or } \vartheta^{K' \rightarrow K} \theta^K' \in \text{TypeApp}^K = \]
\[ \text{TyName}^K \cup \]
\[ (\bigcup_{K' \in \text{Kind}} (\text{TypeApp}^{K' \rightarrow K} \times \text{TypeFcn}^{K'})) \]
\[ \sigma \text{ or } \forall \alpha^{(k)} \cdot \tau \in \text{TypeScheme} = \bigcup_{k \geq 0} \text{TyVar}^{(k)} \times \text{Type} \]

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 \( \tau \) is an equality type, or admits equality, if it is of one of the forms

- \( \alpha \), where \( \alpha \) admits equality;
5.4 Types, Type Applications and Type functions

\[(\theta^k, VE) \in TyStr = (\bigcup_{k \in \text{Arity}} \text{TypeFcn}^k) \times \text{ValEnv}\]

\[TE \in TyEnv = \text{TyCon}^\text{fin} \rightarrow TyStr\]

\[VE \in \text{ValEnv} = \text{VId}^\text{fin} \rightarrow \text{TypeScheme} \times \text{IdStatus}\]

\[E \text{ or } (GE, FE, SE, TE, VE) \in \text{Env} = \text{SigEnv} \times \text{FunEnv} \times \text{StrEnv} \times \text{TyEnv} \times \text{ValEnv}\]

\[T \in \text{TyNameSet} = \text{Fin}(\text{TyName})\]

\[IE \in \text{IdEnv} = \text{TyId}^\text{fin} \rightarrow \text{Type}\]

\[C \text{ or } (IE, E) \in \text{Context} = \text{IdEnv} \times \text{Env}\]

Figure 11: Compound Semantic Objects (continued)

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

(Note that if \(\tau\) is a package type \([X]\) then it does not admit equality.)

A type function \(\theta\) 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 \(\tau\) also admits equality;
- \(\vartheta\), where \(\vartheta \in \text{TypeApp}^{k=}\).

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 \(\theta = \Lambda t^K . \theta'\) has kind \(K \rightarrow K'\), provided \(\theta'\) 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, \(\Lambda\alpha . \alpha \rightarrow \alpha\) and \(\Lambda\beta . \beta \rightarrow \beta\) are equal type functions even if \(\alpha\) admits equality but \(\beta\) does not.

If the type application \(\vartheta\) has kind \(k\) then we identify the type function \(\theta = \vartheta\) with the type function \(\Lambda\alpha^{(k)} . \alpha^{(k)} \vartheta\) (provided \((\text{tyvars} \alpha^{(k)}) \cap (\text{tyvars} \vartheta) = \))
\( \emptyset \) (eta-conversion). If the type application \( \vartheta \) has kind \( K \to K' \) then we identify the type function \( \theta = \vartheta \) with the type function \( \Lambda t^K.\vartheta \ t \) (provided \( t \not\in \text{tynames } \vartheta \)) (eta-conversion).

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

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

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

(assuming \( t \not\in \text{tynames } \tau^{(k)} \)); and

\[
(t \ \theta)\{\Lambda u.\theta'/t\} = \theta'\{\theta/u\}.
\]

(assuming \( t \not\in \text{tynames } \theta \)).

### 5.5 Type Schemes

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

Two type schemes \( \sigma \) and \( \sigma' \) 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 \( \alpha \) and \( \beta \) admit equality, or neither does. It can be shown that \( \sigma \succ \sigma' \) iff \( \sigma \succ \sigma' \) and \( \sigma' \succ \sigma \).

We consider a type \( \tau \) 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 \( \text{Exp} \) 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 \( nexp \):

\[
\begin{align*}
\text{nexp} & ::= \text{scon} \\
& \quad \langle \text{op}\rangle \text{longvid} \\
& \quad \{\langle \text{nexprow}\rangle\} \\
& \quad (\text{nexp}) \\
& \quad \text{conexp nexp} \\
& \quad \text{nexp ty} \\
& \quad \text{fn match} \\
& \quad [\text{structure nmodexp as sigexp}] \\
& \quad [\text{functor nmodexp as sigexp}] \\
\text{nexprow} & ::= \text{lab} = \text{nexp} (', \text{nexprow}) \\
\text{conexp} & ::= (\text{conexp} (': \text{ty})) \\
& \quad \langle \text{op}\rangle \text{longvid} \\
\text{nmodexp} & ::= \langle \text{op}\rangle \text{longmodid} \\
& \quad (\text{nmodexp}) \\
& \quad \text{nmodexp} : \text{sigexp} \\
& \quad \text{nmodexp} :> \text{sigexp} \\
& \quad \text{functor} \ (\text{modid} : \text{sigexp}) :> \text{modexp} \\
& \quad \text{functor} \ \text{modid} : \text{sigexp} :> \text{modexp} \\
& \quad \text{rec} \ (\text{strid} : \text{sigexp}) \text{nmodexp}
\end{align*}
\]

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

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

### 5.7 Closure

Let \( \tau \) be a type and \( A \) a semantic object. Then \( \text{Clos}_A(\tau) \), the closure of \( \tau \) with respect to \( A \), is the type scheme \( \forall \alpha^{(k)}.\tau \), where \( \alpha^{(k)} = \text{tyvars}(\tau) \setminus \text{tyvars} \ A \). Commonly, \( A \) will be a context \( C \). We abbreviate the total closure \( \text{Clos}_\{\} (\tau) \) to \( \text{Clos}(\tau) \). If the range of a value environment \( \text{VE} \) contains only types
(rather than arbitrary type schemes) we set

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

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

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

### 5.8 Existential and Parameterised Objects

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

\[
\exists T.a \in \text{Ex}(A) = \text{TyNameSet} \times A \\
\Lambda T.a \in \text{Par}(A) = \text{TyNameSet} \times A
\]

(where \(a\) ranges over elements of \(A\)).

The prefixes \(\exists T.\) and \(\Lambda T.\), are binding constructs. Two objects in \(\text{Ex}(A)\) ( \(\text{Par}(A)\)) 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 \((\theta^{(k)}, \text{VE})\) is well-formed if either \(\text{VE} = \{\}\), or \(\theta^{(k)}\) is a type application \(\vartheta^{(k)}\). (The latter case arises, with \(\text{VE} \neq \{\}\), in \textit{datatype} declarations.) All type structures occurring in elaborations are assumed to be well-formed.

A type structure \((\vartheta, \text{VE})\) is said to respect equality if, whenever \(\vartheta \in \text{TypeApp}^{\kappa^\omega}\) (i.e. \(\vartheta\) admits equality), then either \(\vartheta = \text{ref}\) (see Appendix ??) or, for each \(\text{VE}(\text{vid})\) 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 \((\text{vid}, v)\) of type \(\tau^{(k)} \vartheta\) only when it is applicable to the value \(v\) itself, whose type is \(\tau^{(k)} / \alpha^{(k)}\).) A type environment \(\text{TE}\) 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 $VE \neq \emptyset$. 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

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

where no $VE_i$ is the empty map, and for any $E$ we define $\text{Abs}(TE,E)$ to be the environment obtained from $E$ and $TE$ as follows. First, let $\text{Abs}(TE)$ be the type environment $\{ \text{tycon}_i \mapsto (t_i,\emptyset) ; 1 \leq i \leq k \}$ in which all value environments $VE_i$ have been replaced by the empty map. Let $T' = \{ t'_1, \ldots, t'_k \}$ 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 $\text{Abs}(TE,E) = \exists T'.E'$, where $E'$ is the result of simultaneously substituting $t'_i$ for $t_i$, $1 \leq i \leq k$, throughout $\text{Abs}(TE) + E$, i.e. $E' = (\text{Abs}(TE) + E)\{ t'^{(k)}/t^{(k)} \}$ (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

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

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.
5  STATIC SEMANTICS FOR THE CORE

Long Value Identifiers

\[ C \vdash \text{longvid} \Rightarrow (\sigma, \text{is}) \]

\[ \text{vid} \in \text{Dom} C \]
\[ C \vdash \text{vid} \Rightarrow C(\text{vid}) \] (1)

\[ C \vdash \text{longstrid} \Rightarrow \text{RS} \quad S = S \text{ of } \text{RS} \quad \text{vid} \in \text{Dom } S \]
\[ C \vdash \text{longstrid.vid} \Rightarrow S(\text{vid}) \] (2)

Atomic Expressions

\[ C \vdash \text{atexp} \Rightarrow \tau \]

\[ C \vdash \text{scon} \Rightarrow \text{type(scon)} \] (3)

\[ C \vdash \text{longvid} \Rightarrow (\sigma, \text{is}) \quad \sigma \succ \tau \]
\[ C \vdash \text{longvid} \Rightarrow \tau \] (4)

\[ \langle C \vdash \text{exrown} \Rightarrow \varrho \rangle \]
\[ C \vdash \{ \langle \text{exrown} \rangle \} \Rightarrow \{ \langle + \varrho \rangle \} \text{ in Type} \] (5)

\[ C \vdash \text{dec} \Rightarrow \exists \text{TE} \quad T \cap \text{tynames} C = \emptyset \]
\[ C + E \vdash \text{exp} \Rightarrow \tau \quad T \cap \text{tynames} \tau = \emptyset \]
\[ C \vdash \text{let dec in exp end} \Rightarrow \tau \] (6)

\[ C \vdash^a \text{modexp} \Rightarrow \exists \text{T}.\text{RS} \]
\[ C \vdash \text{sigexp} \Rightarrow \Lambda T^\prime.\text{RS}^\prime \]
\[ T \cap \text{tynames}(\Lambda T^\prime.\text{RS}^\prime) = \emptyset \]
\[ \Lambda T^\prime.\text{RS}^\prime \geq \text{RS}'' \prec \text{RS} \]
\[ C \vdash [\text{structure modexp as sigexp}] \Rightarrow [\exists T^\prime.\text{RS}^\prime] \] (7)

\[ C \vdash^t \text{modexp} \Rightarrow \exists T.F \]
\[ C \vdash \text{sigexp} \Rightarrow \Lambda T^\prime.F^\prime \]
\[ T \cap \text{tynames}(\Lambda T^\prime.F^\prime) = \emptyset \]
\[ \Lambda T^\prime.F^\prime \geq F^\prime \prec F \]
\[ C \vdash [\text{functor modexp as sigexp}] \Rightarrow [\exists T^\prime.F^\prime] \] (8)

\[ C \vdash \text{exp} \Rightarrow \tau \]
\[ C \vdash (\text{exp}) \Rightarrow \tau \] (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**

\[
\frac{C \vdash \text{exp} \Rightarrow \tau}{C \vdash \text{lab} = \text{exp} \leftarrow \text{exprow} \Rightarrow \{\text{lab} \mapsto \tau\}(+ \varrho)}
\]  

(10)

**Expressions**

\[
\frac{C \vdash \text{atexp} \Rightarrow \tau}{\frac{C \vdash \text{exp} \Rightarrow \tau}{C \vdash \text{atexp} \Rightarrow \tau}}
\]

(11)

\[
\frac{C \vdash \text{exp} \Rightarrow \tau' \rightarrow \tau \quad C \vdash \text{atexp} \Rightarrow \tau'}{C \vdash \text{exp} \ \text{atexp} \Rightarrow \tau}
\]

(12)

\[
\frac{C \vdash \text{exp} \Rightarrow \tau \quad C \vdash \text{ty} \Rightarrow \tau}{C \vdash \text{exp} \ : \ \text{ty} \Rightarrow \tau}
\]

(13)

\[
\frac{C \vdash \text{exp} \Rightarrow \tau \quad C \vdash \text{match} \Rightarrow \text{exn} \rightarrow \tau}{C \vdash \text{exp} \ \text{handle} \ \text{match} \Rightarrow \tau}
\]

(14)

\[
\frac{C \vdash \text{exp} \Rightarrow \text{exn}}{C \vdash \text{raise} \ \text{exp} \Rightarrow \tau}
\]

(15)

\[
\frac{C \vdash \text{match} \Rightarrow \tau}{C \vdash \text{fn} \ \text{match} \Rightarrow \tau}
\]

(16)

**Comments:**

(11) The relational symbol \( \vdash \) is overloaded for all syntactic classes (here atomic expressions and expressions).
Here $\tau$ is determined by $C$ and $ty$. Notice that type variables in $ty$ cannot be instantiated in obtaining $\tau$; thus the expression $1:\alpha$ will not elaborate successfully, nor will the expression $(\text{fn } x=>x):\alpha->\beta$. The effect of type variables in an explicitly typed expression is to indicate exactly the degree of polymorphism present in the expression.

Note that $\tau$ does not occur in the premise; thus a raise expression has “arbitrary” type.

### Matches

\[
C \vdash \text{match} \Rightarrow \tau
\]

\[
\frac{C \vdash \text{mrule} \Rightarrow \tau}{C \vdash \text{mrule} (\mid \text{match}) \Rightarrow \tau}
\] (17)

### Match Rules

\[
C \vdash \text{mrule} \Rightarrow \tau
\]

\[
\frac{C \vdash \text{pat} \Rightarrow (\text{VE},\tau)}{C + \text{VE} \vdash \text{exp} \Rightarrow \tau'}
\]

\[
\frac{C \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \tau \Rightarrow \tau'}{C \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \tau \Rightarrow \tau'}
\]

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 $\text{pat}$ (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 $\text{exp}$ (see rule 13).

### Declarations

\[
C \vdash \text{dec} \Rightarrow \exists T.E
\]

\[
C \vdash \text{tyidseq} \Rightarrow (\alpha_1,\ldots,\alpha_k),IE \quad C + IE \vdash \text{valbind} \Rightarrow \text{VE}
\]

\[
VE' = \text{Clos}_{C,\text{valbind}}\text{VE}
\]

\[
\{\alpha_1,\ldots,\alpha_k\} \cap \text{tyvars}VE' = \emptyset
\]

\[
C \vdash \text{val} \text{ tyidseq valbind} \Rightarrow \exists \emptyset.\text{VE}' \text{ in Env}
\]

\[
C \vdash \text{typbind} \Rightarrow \text{TE}
\]

\[
C \vdash \text{type} \text{ typbind} \Rightarrow \exists \emptyset.\text{TE} \text{ in Env}
\]

\[
C + \text{TE} \vdash \text{datbind} \Rightarrow \text{VE},\text{TE}
\]

\[
T = \{t;(t,\text{VE'}) \in \text{Ran}\text{TE}\}
\]

\[
T \cap \text{tynames}C = \emptyset
\]

\[
\text{TE} \text{ maximises equality}
\]

\[
C \vdash \text{datatype} \text{ datbind} \Rightarrow \exists T.(\text{VE},\text{TE}) \text{ in Env}
\] (21)
5.10 Inference Rules

\[
C \vdash \text{tyconpath} \Rightarrow (\theta,VE)
\]
\[
TE = \{\text{tycon} \mapsto (\theta,VE)\}
\]
\[
C \vdash \text{datatype tycon = datatype tyconpath} \Rightarrow \exists\emptyset.(VE,TE) \text{ in Env}
\]

\[
C + TE \vdash \text{datbind} \Rightarrow VE,TE
\]
\[
T = \{t; (t,VE) \in \text{RanTE}\}
\]
\[
C + (VE,TE) \vdash \text{dec} \Rightarrow \exists T'.E
\]
\[
T \cap \text{tynames} C = \emptyset
\]
\[
Abs(TE,E) = \exists T''.E'
\]
\[
T'' \cap T' = \emptyset
\]
\[
C \vdash \text{abstype datbind with dec end} \Rightarrow \exists T \cup T'' . \text{Abs}(TE,E)
\]

\[
C \vdash \text{exbind} \Rightarrow VE
\]
\[
C \vdash \text{exception exbind} \Rightarrow \exists\emptyset.VE \text{ in Env}
\]
\[
C \vdash \text{local dec}_1 \text{ in dec}_2 \text{ end} \Rightarrow \exists T_1 \cup T_2 . E_2
\]

\[
C \vdash \text{longstrid}_1 \Rightarrow RS_1
\]
\[
S_1 = \text{S of } RS_1
\]
\[
\vdots
\]
\[
C \vdash \text{longstrid}_n \Rightarrow RS_n
\]
\[
S_n = \text{S of } RS_n
\]
\[
C \vdash \text{open longstrid}_1 \cdots \text{longstrid}_n \Rightarrow \exists\emptyset.(S_1 + \cdots + S_n) \text{ in Env}
\]

\[
C \vdash \text{strbind} \Rightarrow \exists T.SE
\]
\[
C \vdash \text{structure strbind} \Rightarrow \exists T.SE \text{ in Env}
\]

\[
C \vdash \text{funbind} \Rightarrow \exists T.FE
\]
\[
C \vdash \text{functor funbind} \Rightarrow \exists T.FE \text{ in Env}
\]

\[
C \vdash \text{signature sigbind} \Rightarrow GE
\]
\[
C \vdash \text{signature sigbind} \Rightarrow \exists\emptyset.GE \text{ in Env}
\]

\[
C \vdash \Rightarrow \exists\emptyset.\{\} \text{ in Env}
\]

\[
C \vdash \text{dec}_1 \Rightarrow \exists T_1.E_1
\]
\[
T_1 \cap \text{tynames} C = \emptyset
\]
\[
C + E_1 \vdash \text{dec}_2 \Rightarrow \exists T_2.E_2
\]
\[
T_2 \cap (T_1 \cup \text{tynames } E_1) = \emptyset
\]
\[
C \vdash \text{dec}_1 (;) \text{ dec}_2 \Rightarrow \exists T_1 \cup T_2 . E_1 + E_2
\]

Comments:
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 $\{\alpha_1, \ldots, \alpha_k\}$ 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 $\text{val } tyidseq \text{ valbind}$ occurs inside some larger value binding $\text{val } tyidseq_0 \text{ valbind}_0$ then no type variable $\alpha$ bound to a type identifier listed in $tyidseq_0$ will become bound by the $\text{Clos}_{C, \text{valbind}}(VE)$ operation; for $\alpha$ must be in $IE$ of $C$ and hence excluded from closure by the definition of the closure operation (Section 5.7, page 26) since $\text{tyvars}(IE$ of $C) \subseteq \text{tyvars} C$.

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.

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

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

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

Value Bindings

$$
\frac{C \vdash \text{pat } \Rightarrow (VE, \tau) \quad C \vdash \text{exp } \Rightarrow \tau \quad (C \vdash \text{valbind } \Rightarrow VE')}{C \vdash \text{pat } = \text{exp } \langle \text{and } \text{valbind} \rangle \Rightarrow VE \langle + VE' \rangle} \quad (32)
$$

$$
\frac{C \vdash \text{valbind } \Rightarrow VE}{C \vdash \text{rec } \text{valbind } \Rightarrow VE} \quad (33)
$$

Comments:

When the option is present we have $\text{Dom} \text{VE } \cap \text{Dom} \text{VE'} = \emptyset$ 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
\[
\text{datatype } t = f ; \text{ val rec } f = \text{ fn } x \Rightarrow x ; \text{ is legal.}
\]

**Type Bindings**

\[
C \vdash \text{typbind} \Rightarrow \text{TE} \\
C \vdash \text{tyidseq} \Rightarrow (\alpha^{(k)}, IE) \quad C + IE \vdash ty \Rightarrow \tau \quad (C \vdash \text{typbind} \Rightarrow \text{TE}) \\
C \vdash \text{tyidseq tycon} = ty \text{ (and typbind)} \Rightarrow \{ tycon \mapsto (\Lambda \alpha^{(k)}.\tau, \{ \}) \} (\vdash \text{TE})
\]

**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 \( \text{tycon} \notin \text{Dom } \text{TE} \).

**Data Type Bindings**

\[
C \vdash \text{datbind} \Rightarrow \text{VE}, \text{TE} \\
C \vdash \text{tyidseq} \Rightarrow (\alpha^{(k)}, IE) \\
C + IE, \alpha^{(k)} t \vdash \text{conbind} \Rightarrow \text{VE} \\
(C \vdash \text{datbind} \Rightarrow \text{VE}', \text{TE}') \quad \forall (t', \text{VE}''') \in \text{Ran } \text{TE}, t \neq t' \\
C \vdash \text{tyidseq tycon} = \text{conbind (and datbind)} \Rightarrow Clos_C \text{VE}(\vdash \text{VE}') \quad \{ tycon \mapsto (t, Clos_C \text{VE}) \} (\vdash \text{TE}')
\]

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

**Constructor Bindings**

\[
(C, \tau \vdash \text{conbind} \Rightarrow \text{VE}) \\
(C \vdash ty \Rightarrow \tau') \quad \langle C, \tau \vdash \text{conbind} \Rightarrow \text{VE} \rangle \\
C, \tau \vdash \text{vid (of ty)} \langle C, \tau \vdash \text{conbind} \Rightarrow \text{VE} \rangle \\
\{ \text{vid} \mapsto (\tau, c) \} (\vdash \{ \text{vid} \mapsto (\tau' \rightarrow \tau, c) \} (\vdash \text{+ VE}))
\]

**Comment:** By the syntactic restrictions \( \text{vid} \notin \text{Dom } \text{VE} \).
Exception Bindings

\[ C \vdash \text{exbind} \Rightarrow VE \]

\[
\frac{(C \vdash ty \Rightarrow \tau)}{C \vdash \text{exbind} \Rightarrow VE}\]

\[
\frac{C \vdash \text{vid} \langle \text{of ty} \rangle \langle \langle \text{and exbind} \rangle \Rightarrow \{\text{vid} \mapsto (\text{exn}, e)\} \{+ \} \langle + VE \rangle}{C \vdash \text{longvid} \Rightarrow (\tau, e) \langle \langle C \vdash \text{exbind} \Rightarrow VE \rangle \Rightarrow \{\text{vid} \mapsto (\tau, e)\} \{+\} \langle + VE \rangle}
\]

(37)

\[
C \vdash \text{vid} = \text{longvid} \langle \langle \text{and exbind} \rangle \Rightarrow \{\text{vid} \mapsto (\tau, e)\} \{+\} \langle + VE \rangle
\]

(38)

Comments:

(37) Notice that \( \tau \) may contain type variables.

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

Atomic Patterns

\[ C \vdash \text{atpat} \Rightarrow (\text{VE}, \tau) \]

\[
\frac{C \vdash \_ \Rightarrow (\{\}, \tau)}{C \vdash \text{atpat} \Rightarrow (\text{VE}, \tau)}
\]

\[ C \vdash \text{scon} \Rightarrow (\{\}, \text{type(scon)}) \]

\[
\frac{\text{vid} \notin \text{Dom}(C) \text{ or } \text{is of } C(\text{vid}) = \nu}{C \vdash \text{vid} \Rightarrow (\{\text{vid} \mapsto (\tau, \nu), \tau})
\]

\[
\frac{C \vdash \text{longvid} \Rightarrow (\sigma, \text{is}) \text{ is } \neq \nu \text{ and } \tau^{(k)} \vartheta}{C \vdash \text{longvid} \Rightarrow (\{\}, \tau^{(k)} \vartheta)}
\]

\[
\frac{\langle C \vdash \text{patrow} \Rightarrow (\text{VE}, \vartheta) \rangle}{C \vdash \{\langle \text{patrow} \rangle \Rightarrow (\{\} \langle + \text{VE} \rangle, \{\} \langle + \vartheta \rangle \text{ in Type})}
\]

\[
\frac{C \vdash \text{pat} \Rightarrow (\text{VE}, \tau)}{C \vdash (\text{pat}) \Rightarrow (\text{VE}, \tau)}
\]

Comments:

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

\[ C \vdash \text{patrow} \Rightarrow (VE, \varrho) \]

\[ C \vdash \ldots \Rightarrow (\{\}, \varrho) \] (45)

\[ C \vdash \text{pat} \Rightarrow (VE, \tau) \]

\[ \langle C \vdash \text{patrow} \Rightarrow (VE', \varrho) \quad \text{lab} \notin \text{Dom } \varrho \rangle \]

\[ C \vdash \text{lab} = \text{pat} \langle , \text{patrow} \rangle \Rightarrow (VE' + VE'', \{\text{lab} \mapsto \tau\} + \varrho) \] (46)

Patterns

\[ C \vdash \text{pat} \Rightarrow (VE, \tau) \]

\[ C \vdash \text{atpat} \Rightarrow (VE, \tau) \]

\[ C \vdash \text{atpat} \Rightarrow (VE, \tau) \] (47)

\[ C \vdash \text{longvid} \Rightarrow (\sigma, \text{is}) \quad \text{is} \neq v \quad \sigma \succ \tau' \rightarrow \tau \]

\[ C \vdash \text{atpat} \Rightarrow (VE, \tau') \]

\[ C \vdash \text{longvid atpat} \Rightarrow (VE, \tau) \] (48)

\[ C \vdash \text{pat} \Rightarrow (VE, \tau) \quad C \vdash \text{ty} \Rightarrow \tau \]

\[ C \vdash \text{pat : ty} \Rightarrow (VE, \tau) \] (49)

\[ vid \notin \text{Dom}(C) \text{ or } \text{is of } C(vid) = v \]

\[ \langle C \vdash \text{ty} \Rightarrow \tau \rangle \quad C \vdash \text{pat} \Rightarrow (VE, \tau) \quad vid \notin \text{DomVE} \]

\[ C \vdash \text{vid}(: \text{ty}) \text{ as pat} \Rightarrow (\{\text{vid} \mapsto (\tau, v)\} + VE, \tau) \] (50)

Long Type Constructors

\[ C \vdash \text{longtycon} \Rightarrow (\theta^k, VE) \]

\[ \text{tycon} \in \text{Dom } C \]

\[ C \vdash \text{tycon} \Rightarrow C(\text{tycon}) \] (51)

\[ C \vdash \text{longstrid} \Rightarrow RS \quad S = S \text{ of } RS \quad \text{tycon} \in \text{Dom } S \]

\[ C \vdash \text{longstrid.tycon} \Rightarrow S(\text{tycon}) \] (52)
Type Constructor Paths

\[ C \vdash \text{tyconpath} \Rightarrow (\theta^k, \text{VE}) \]

\[ C \vdash \text{longtycon} \Rightarrow (\theta, \text{VE}) \]

\[ C \vdash \text{longtycon} \Rightarrow (\theta, \text{VE}) \]

\[ C \vdash \text{longtycon \ where \ strid} = \text{modexp} \Rightarrow (\theta, \text{VE}) \]

Type Identifier Sequences

\[ C \vdash \text{tyidseq} \Rightarrow (\alpha^k, \text{IE}) \]

\[ \alpha \text{ admits equality iff } \text{tyid} \in \text{ETyId} \quad \alpha \not\in (\text{tyvars} C) \]

\[ C \vdash \text{tyid} \Rightarrow ((\alpha), \{ \text{tyid} \mapsto (\alpha) \}) \]

\[ C \vdash \Rightarrow ((), \{\}) \]

\[ C \vdash (\text{tyid}_1, \cdots, \text{tyid}_k) \Rightarrow ((\alpha_1, \cdots, \alpha_k), \{ \text{tyid}_1 \mapsto \alpha_1 \} + \cdots + \{ \text{tyid}_k \mapsto \alpha_k \}) \]

Type Expressions

\[ C \vdash \text{ty} \Rightarrow \tau \]

\[ \text{tyid} \in \text{Dom } C \]

\[ C \vdash \text{tyid} \Rightarrow C(\text{tyid}) \]

\[ \langle C \vdash \text{tyrow} \Rightarrow \varrho \rangle \]

\[ C \vdash \{ \langle \text{tyrow} \rangle \} \Rightarrow \{\} \langle + \varrho \rangle \text{ in Type} \]

\[ \text{tyseq} = \text{ty}_1 \cdots \text{ty}_k \quad C \vdash \text{ty}_i \Rightarrow \tau_i \ (1 \leq i \leq k) \]

\[ C \vdash \text{tyconpath} \Rightarrow (\theta^k, \text{VE}) \]

\[ C \vdash \text{tyseq} \text{ tyconpath} \Rightarrow \tau^{(k)} \theta^k \]

\[ C \vdash \text{ty} \Rightarrow \tau \quad C \vdash \text{ty}' \Rightarrow \tau' \]

\[ C \vdash \text{ty} \rightarrow \text{ty}' \Rightarrow \tau \rightarrow \tau' \]
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, \ldots, lab_m=pat_m, \ldots\} the program context must determine uniquely the domain \{lab_1, \ldots, lab_n\} of its row type, where \(m \leq n\); thus, the context must determine the labels \{lab_{m+1}, \ldots, lab_n\} of the fields to be matched by the wildcard. For this purpose, an explicit type constraint may be needed.

2. In a match of the form \(pat_1 \Rightarrow exp_1 \mid \cdots \mid pat_n \Rightarrow exp_n\) 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 \texttt{fn match}, the \texttt{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 \texttt{vid atpat}_1 \cdots \texttt{atpat}_n; \texttt{(ty)} = \texttt{exp} (consisting of one clause only), each separate \texttt{atpat}_i should be exhaustive by itself.
3. For each value binding $\text{pat} = \text{exp}$ the compiler must issue a report (but still compile) if $\text{pat}$ is not exhaustive. This will detect a mistaken declaration like $\text{val nil} = \text{exp}$ in which the user expects to declare a new variable $\text{nil}$ (whereas the language dictates that $\text{nil}$ is here a constant pattern, so no variable gets declared). However, this warning should not be given when the binding is a component of a top-level declaration $\text{val tyidseq valbind}$; e.g. $\text{val x::l = exp}_1 \text{ and y = exp}_2$ is not faulted by the compiler at top level, but may of course generate a $\text{Bind}$ 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.

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

Figure 12: Further Compound Semantic Objects

The prefixes \(\Lambda T.\), \(\exists T.\) and \(\forall T. \rightarrow\) 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 \(C + \{\text{modid} \mapsto M\}\) to mean \(C + \{\text{strid} \mapsto RS\}\) if \(M = RS\) and where \(\text{strid} = \text{modid}\); and \(C + \{\text{funid} \mapsto RS\}\) if \(M = F\) and where \(\text{funid} = \text{modid}\). 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 $\varphi : \text{TyName} \rightarrow \text{TypeFcn}$ such that $t$ and $\varphi(t)$ have the same kind; in particular, if $t$ admits equality then so does $\varphi(t)$.

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

Realisations $\varphi$ are extended to apply to all semantic objects; their effect is to replace each name $t$ by $\varphi(t)$ (performing $\beta$-reductions as necessary to preserve the structure of constructed types, type applications and type functions). In applying $\varphi$ 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 ($\Lambda T.\_ \exists T.\_ \text{ and } \forall T.\_ \rightarrow \_)$,

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

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 $\varphi$ such that $\varphi(M') = M$ and $\text{Supp } \varphi \subseteq T$.

6.4 Functor Instantiation

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

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.
6.5 Enrichment

- 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. \( \text{Dom } FE_1 \supseteq \text{Dom } FE_2 \), and \( FE_1(\text{funid}) \succ FE_2(\text{funid}) \) for all \( \text{funid} \in \text{Dom } FE_2 \),
  
  2. \( \text{Dom } SE_1 \supseteq \text{Dom } SE_2 \), and \( SE_1(\text{strid}) \succ SE_2(\text{strid}) \) for all \( \text{strid} \in \text{Dom } SE_2 \),
  
  3. \( \text{Dom } TE_1 \supseteq \text{Dom } TE_2 \), and \( TE_1(\text{tycon}) \succ TE_2(\text{tycon}) \) for all \( \text{tycon} \in \text{Dom } TE_2 \), and
  
  4. \( \text{Dom } VE_1 \supseteq \text{Dom } VE_2 \), and \( VE_1(\text{vid}) \succ VE_2(\text{vid}) \) for all \( \text{vid} \in \text{Dom } VE_2 \).

- 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 \( \varphi \) such that:
  
  1. \( T_2 \cap \text{tynames}(F_1) = \emptyset \),
  
  2. \( M_2 \succ \varphi(M_1) \),
  
  3. \( \varphi(X_1) \succ X_2 \), and
  
  4. \( \text{Supp } \varphi \subseteq T_1 \).

- 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. \( T_1 \cap \text{tynames}(X_2) = \emptyset \) and \( M_1 \succ \varphi_2(M_2) \) for some realisation \( \varphi_2 \) with \( \text{Supp } \varphi_2 \subseteq T_2 \).
Finally, a type structure \((\theta_1, VE_1)\) enriches another type structure \((\theta_2, VE_2)\), written \((\theta_1, VE_1) \succ (\theta_2, VE_2)\), if

1. \(\theta_1\) and \(\theta_2\) 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 \([X]\) \(\in\) 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:
6.8 Applicative Module Expressions

\[ \text{appdec} ::= \text{val tyidseq valbind} \]
\[ \quad \text{type typbind} \]
\[ \quad \text{datatype datbind} \]
\[ \quad \text{datatype tycon = datatype tyconpath} \]
\[ \quad \text{abstype datbind with appdec end} \]
\[ \quad \text{exception exbind} \]
\[ \quad \text{local appdec in appdec end} \]
\[ \quad \text{open longstrid} \_ \cdots \_ \text{longstrid} \_n \]
\[ \quad \text{structure appstrbind} \]
\[ \quad \text{functor appfunbind} \]
\[ \quad \text{signature sigbind} \]

\[ \quad \text{appdec} \_1 (;) \text{appdec} \_2 \]
\[ \quad \text{infix} (d) \text{vid} \_1 \cdots \text{vid} \_n \]
\[ \quad \text{infixr} (d) \text{vid} \_1 \cdots \text{vid} \_n \]
\[ \quad \text{nonfix} \text{vid} \_1 \cdots \text{vid} \_n \]

\[ \text{appstrbind} ::= \text{strid = appmodexp (and appstrbind)} \]
\[ \text{appfunbind} ::= \text{funid = appmodexp (and appfunbind)} \]
\[ \text{appatmodexp} ::= \text{struct appdec end} \]
\[ \quad \langle \text{op}\rangle \text{longmodid} \]
\[ \quad \text{let appdec in modexp end} \]
\[ \quad (\text{appmodexp}) \]

\[ \text{appmodexp} ::= \text{appatmodexp} \]
\[ \quad \text{appmodexp appatmodexp} \]
\[ \quad \text{appmodexp : sigexp} \]
\[ \quad \text{appmodexp :> sigexp} \]
\[ \quad \text{functor (modid:sigexp) => modexp} \]
\[ \quad \text{functor modid:sigexp => modexp} \]
\[ \quad \text{rec (strid:sigexp)appmodexp} \]

Figure 13: Applicative Module Expressions
Type soundness is preserved by the inference rules by ensuring that if \( \text{modexp} \) occurs as the body of a functor, i.e. a phrases of the form

\[
\text{functor modid: sigexp }\Rightarrow \text{modexp}
\]

or

\[
\text{functor } (\text{modid: sigexp}) \Rightarrow \text{modexp},
\]

then \( \text{modexp} \) must be applicative.

This restriction applies only to functor bodies; in particular, it does not preclude declarations of the form \( \text{strid as sigexp }= \text{exp} \) or \( \text{funid as sigexp }= \text{exp} \) from occurring in top-level declarations or structures, nor does it preclude them from occurring within Core expressions of the form \text{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. \textit{A priori}, the module expression \( \langle \text{op} \rangle \text{longmodid} \) 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 \( \langle \text{op} \rangle \text{longmodid} \) occurs as the right hand side of a structure (functor) binding, then \text{longmodid} must be interpreted as an element of \text{longStrId} (\text{longFunId}); if \( \langle \text{op} \rangle \text{longmodid} \) occurs in the functor position of an application, then \text{longmodid} must be interpreted as an element of \text{longFunId}; if \( \langle \text{op} \rangle \text{longmodid} \) is constrained by a signature then the signature forces a unique interpretation on \text{longmodid} (depending on whether the signature specifies a structure or functor). Similarly, if \( \langle \text{op} \rangle \text{longmodid} \) occurs as the argument of a functor application, then the functor’s domain forces a unique interpretation on \text{longmodid}. Indeed, the only ambiguity that remains occurs when \( \langle \text{op} \rangle \text{longmodid} \) is the body of a functor. In this case, the optional prefix \( \langle \text{op} \rangle \) is used to resolve the ambiguity: the \textit{absence} of \text{op} signals that \text{longmodid} refers to structure; the \textit{presence} of \text{op} signals that \text{op} \text{longmodid} refers to a functor. When the interpretation of \( \langle \text{op} \rangle \text{longmodid} \) is already determined by the context, the optional prefix \( \langle \text{op} \rangle \) has no effect.

Since this method of disambiguation relies on type information 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:

\[ \text{ex} \in \text{Expectation} = \{s, f, m\} \]

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 \(\langle \text{op} \rangle \text{longmodid}\) must be resolved by \(\langle \text{op} \rangle\).

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

\[
\begin{align*}
\text{expect}(\_ ) & \in \text{Sig} \to \text{Expectation} \\
\text{expect}(G) & = \begin{cases} 
  s & \text{if } G = \Lambda T.RS \\
  f & \text{if } G = \Lambda T.F 
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{expect}(\_ ) & \in \text{Fun} \to \text{Expectation} \\
\text{expect}(F) & = \begin{cases} 
  s & \text{if } F = \forall T.RS \to X \\
  f & \text{if } F = \forall T.F' \to X 
\end{cases}
\end{align*}
\]

6.10 Inference Rules

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

\(A \vdash \text{phrase} \Rightarrow A'\)

to be inferred, where \(A\) is typically a context, \(\text{phrase}\) 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**

\[
\begin{align*}
C \vdash \text{longstrid} & \Rightarrow RS \\
\text{strid} & \in \text{Dom} C \\
\frac{C \vdash \text{strid} \Rightarrow C(\text{strid})}{C \vdash \text{longstrid}, \text{strid} \Rightarrow SE(\text{strid})}
\end{align*}
\]

\[
\begin{align*}
C \vdash \text{longstrid} & \Rightarrow RS \\
SE & = SE \text{ of } (S \text{ of } RS) \\
\text{strid} & \in \text{Dom} SE \\
\frac{C \vdash \text{longstrid}.\text{strid} \Rightarrow SE(\text{strid})}{C \vdash \text{longstrid}.\text{strid} \Rightarrow SE(\text{strid})}
\end{align*}
\]

\[
\begin{align*}
C \vdash \text{longstrid} & \Rightarrow RS \\
SE & = SE \text{ of } (S \text{ of } RS) \\
\text{strid} & \in \text{Dom} SE
\end{align*}
\]
Long Functor Identifiers

\[
C \vdash \text{funid} \Rightarrow F \quad \text{funid} \in \text{Dom } C
\]

\[
C \vdash \text{funid} \Rightarrow C(\text{funid}) \quad (67)
\]

\[
C \vdash \text{longstrid} \Rightarrow RS \quad FE = FE \text{ of } (S \text{ of } RS) \quad \text{funid} \in \text{Dom } FE
\]

\[
C \vdash \text{longstrid.funid} \Rightarrow FE(\text{funid}) \quad (68)
\]

Long Module Identifiers

\[
C \vdash \text{longmodid} \Rightarrow M
\]

\[
C \vdash \text{longstrid} \Rightarrow RS \quad \text{longstrid} = \text{longmodid}
\]

\[
C \vdash (\text{op}) \text{ longmodid} \Rightarrow RS \quad (69)
\]

\[
C \vdash \text{longfunid} \Rightarrow F \quad \text{longfunid} = \text{longmodid}
\]

\[
C \vdash (\text{op}) \text{ longmodid} \Rightarrow F \quad (70)
\]

\[
C \vdash \text{longstrid} \Rightarrow RS \quad \text{longstrid} = \text{longmodid}
\]

\[
C \vdash (\text{op}) \text{ longmodid} \Rightarrow RS \quad (71)
\]

\[
C \vdash \text{longfunid} \Rightarrow F \quad \text{longfunid} = \text{longmodid}
\]

\[
C \vdash (\text{op}) \text{ longmodid} \Rightarrow F \quad (72)
\]

Atomic Module Expressions

\[
C \vdash \text{atmodexp} \Rightarrow X
\]

\[
C \vdash \text{dec} \Rightarrow \exists T.E
\]

\[
C \vdash \text{struct} \text{ dec end} \Rightarrow \exists T.(F.E \text{ of } E, S.E \text{ of } E, T.E \text{ of } E, V.E \text{ of } E) \quad (73)
\]

\[
C \vdash (\text{op}) \text{ longmodid} \Rightarrow M
\]

\[
C \vdash (\text{op}) \text{ longmodid} \Rightarrow \exists \emptyset.M \quad (74)
\]

\[
C \vdash \text{dec} \Rightarrow \exists T_1.E \quad T_1 \cap \text{tynames } C = \emptyset
\]

\[
C + E \vdash \text{modexp} \Rightarrow \exists T_2.M \quad T_2 \cap T_1 = \emptyset
\]

\[
C \vdash \text{let dec in modexp end} \Rightarrow \exists T_1 \cup T_2.M \quad (75)
\]

\[
C \vdash \text{modexp} \Rightarrow X
\]

\[
C \vdash (\text{modexp}) \Rightarrow X \quad (76)
\]

Comments:
(73) The resulting structure contains the functor, structure, type and value components of $E$. Signatures declared in $\textit{dec}$ are local to $\textit{dec}$ 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**

\[
\begin{align*}
C \vdash^x \textit{modexp} &\Rightarrow X \\
C \vdash^x \textit{atmodexp} &\Rightarrow X \\
C \vdash^x \textit{atmodexp} &\Rightarrow X
\end{align*}
\]

\[
\begin{align*}
C \vdash^x \textit{sigexp} &\Rightarrow \exists T.F \\
T \cap (T' \cup \text{tynames}\ M) &= \emptyset \\
T' \cap \text{tynames}\ F &= \emptyset \\
F \geq M' &\rightarrow \exists T''.M'' \\
M &> M' \\
T'' \cap (T \cup T') &= \emptyset
\end{align*}
\]

\[
\begin{align*}
C \vdash^x \textit{modexp} \textit{atmodexp} &\Rightarrow \exists T \cup T' \cup T''.M''
\end{align*}
\]

\[
\begin{align*}
C \vdash \textit{modexp} &\Rightarrow \exists T.F \\
C \vdash \textit{expect}(F) &\Rightarrow \exists T'.M \\
T \cap \text{tynames}\ G &= \emptyset \\
G &\geq M' > M \\
T'' \cap (T \cup T') &= \emptyset
\end{align*}
\]

\[
\begin{align*}
C \vdash \textit{modexp : sigexp} &\Rightarrow \exists T.M'
\end{align*}
\]

\[
\begin{align*}
C \vdash \textit{sigexp} &\Rightarrow \Lambda T'.M' \\
C \vdash \textit{expect}(\Lambda T'.M') &\Rightarrow \exists T.M \\
T \cap \text{tynames}(\Lambda T'.M') &= \emptyset \\
\Lambda T'.M' &\geq M'' > M
\end{align*}
\]

\[
\begin{align*}
C \vdash \textit{modexp : > sigexp} &\Rightarrow \exists T'.M'
\end{align*}
\]

\[
\begin{align*}
\textit{modexp} \text{ is applicative} \\
C \vdash \textit{sigexp} &\Rightarrow \Lambda T.M \\
T \cap \text{tynames}\ C &= \emptyset \\
C + \{\textit{modid} \mapsto M\} \vdash^x \textit{modexp} &\Rightarrow X
\end{align*}
\]

\[
\begin{align*}
C \vdash^x \textit{functor}(\textit{modid:sigexp}) &\Rightarrow \exists x.(\forall T.M \rightarrow X)
\end{align*}
\]
modexp is applicative
C ⊢ sigexp ⇒ \Lambda T.M
T \cap \text{tynames} C = \emptyset \quad T = \{t_1^{K_1}, \ldots, t_n^{K_n}\}
C + \{\text{modid} \mapsto M\} \vdash^* \text{modexp} ⇒ \exists T'.M'
T'' \cap (T \cup \text{tynames} M \cup \text{tynames}(\exists T'.M')) = \emptyset
\varphi = \{u^{K_1} \mapsto u^{K_1 \cdots K_n \rightarrow K_2 t_1 \cdots t_n; u^K \in T'}\}
T'' = \{u^{K_1 \cdots K_n \rightarrow K_2 t_1 \cdots t_n; u^K \in T'}\}

C \vdash_{\text{ex}} \text{functor} \text{modid}: \text{sigexp} \Rightarrow \text{modexp} ⇒ \exists T''.(\forall T.M \rightarrow \exists \emptyset.\varphi(M')) \tag{82}

C \vdash \text{sigexp} ⇒ \Lambda T.RS
T \cap \text{tynames} C = \emptyset
C + \{\text{strid} \mapsto RS\} \vdash^* \text{modexp} ⇒ \exists T'.RS
T' \cap (T \cup \text{tynames} RS) = \emptyset
RS' > RS

C \vdash_{\text{ex}} \text{rec} (\text{strid}: \text{sigexp}) \text{modexp} ⇒ \exists T \cup T'.(RS, RS') \tag{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 ∪ 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 \rightarrow X_F. \text{Let } \varphi \text{ be a realisation such that } \varphi(M_F \rightarrow X_F) = M' \rightarrow \exists T''.M'' \text{ (with Supp } \varphi \subseteq T_F). \text{ Sharing between the formal domain and the formal range of the functor is represented by occurrences of the same type name of } T_F \text{ in both } M_F \text{ and } X_F. \text{ These shared occurrences are preserved by } \varphi, \text{ yielding sharing between the actual domain } M \text{ and the actual range type } \exists T''.M''. \text{of this functor application.}

(81),(82) In both rules, the functor body modexp is elaborated in the extended context C + \{modid \mapsto M\}. The side condition T \cap \text{tynames} C =
∅, 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 \( \forall T.M \to X = \forall T.M \to \exists T'.M' \). 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 \( \text{modexp} \) is elaborated in the extended context \( C + \{ \text{modid} \mapsto M \} \), names in \( T' \) may have hidden functional dependencies on the type parameters \( T \) of the formal argument \( \text{modid} \). These dependencies are made explicit by applying the realisation \( \varphi \) to \( M' \). This effectively skolemises each occurrence in \( M' \) of a name \( u \in T' \) 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 \( \exists T''.(\forall T.M \to \exists \emptyset.\varphi(M')) \).

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

\[ C \vdash \text{strbind} \Rightarrow \exists T.\text{SE} \]

- \( C \vdash^* \text{modexp} \Rightarrow \exists T.RS \)
- \( (C \vdash \text{strbind} \Rightarrow \exists T'.\text{SE}) \)
- \( (T \cap (T' \cup \text{tynames SE}) = \emptyset) \)
- \( (T' \cap (\text{tynames RS}) = \emptyset) \)

\[ C \vdash \text{strid} = \text{modexp} \langle \text{and strbind} \rangle \Rightarrow \exists T \langle \cup T' \rangle \cdot \{ \text{strid} \mapsto \rightarrow \text{RS} \} \langle + \text{SE} \rangle \]  \( (84) \)

Functor Bindings

\[ C \vdash \text{funbind} \Rightarrow \exists T.FE \]

- \( C \vdash^* \text{modexp} \Rightarrow \exists T.F \)
- \( (C \vdash \text{funbind} \Rightarrow \exists T'.FE) \)
- \( (T \cap (T' \cup \text{tynames FE}) = \emptyset) \)
- \( (T' \cap (\text{tynames F}) = \emptyset) \)

\[ C \vdash \text{funid} = \text{modexp} \langle \text{and funbind} \rangle \Rightarrow \exists T \langle \cup T' \rangle \cdot \{ \text{funid} \mapsto \rightarrow F \} \langle + \text{FE} \rangle \]  \( (86) \)

Signature Bindings

\[ C \vdash \text{sigbind} \Rightarrow \text{GE} \]

- \( C \vdash \text{sigexp} \Rightarrow G \langle C \vdash \text{sigbind} \Rightarrow \text{GE} \rangle \)

\[ C \vdash \text{sigid} = \text{sigexp} \langle \text{and sigbind} \rangle \Rightarrow \{ \text{sigid} \mapsto G \} \langle + \text{GE} \rangle \]  \( (88) \)
6.10  Inference Rules

Signature Expressions

\[ C \vdash \text{spec} \Rightarrow \Lambda T.(GE, FE, SE, TE, VE) \]
\[ C \vdash \text{SIG spec end} \Rightarrow \Lambda T.(FE, SE, TE, VE) \] (89)

\[ \text{sigid} \in \text{Dom} \ C \]
\[ C \vdash \text{sigid} \Rightarrow C(\text{sigid}) \] (90)

\[ C \vdash \text{sig} \Rightarrow \Lambda T.RS \quad S = S \text{ of } RS \]
\[ C \vdash \text{tyidseq} \Rightarrow (\alpha^{(k)}, IE) \quad C + IE \vdash ty \Rightarrow \tau \]
\[ (\{\},\{\}, FE \text{ of } S, SE \text{ of } S, TE \text{ of } S, VE \text{ of } S) \vdash \text{longtycon} \Rightarrow (t, VE) \]
\[ t \in T \quad t \text{ has arity } k \]
\[ \Lambda \alpha^{(k)}, \tau = \emptyset \]
\[ \varphi = \{ t \mapsto \Lambda \alpha^{(k)}, \tau \} \quad \Lambda \alpha^{(k)}, \tau \text{ admits equality, if } t \text{ does} \]
\[ \varphi(S) \text{ well-formed} \]
\[ C \vdash \text{sig where type tyidseq longtycon = ty} \Rightarrow \Lambda T \setminus \{t\}.\varphi(RS) \] (91)

\[ C \vdash \text{functor (modid : sigexp} _1 \rightarrow \text{sigexp} _2 \Rightarrow \Lambda T.M \]
\[ T \cap \text{tynames} C = \emptyset \]
\[ C + \{ \text{modid } \mapsto M \} \vdash \text{sigexp} _2 \Rightarrow \Lambda T'.M' \]
\[ C \vdash \text{functor (modid : sigexp} _1 \rightarrow \text{sigexp} _2 \Rightarrow \Lambda T'.M' \] (92)

\[ C \vdash \text{sigexp} _1 \Rightarrow \Lambda T.M \]
\[ T \cap \text{tynames} C = \emptyset \quad T = \{ t_1^{K_1}, \ldots, t_n^{K_n} \} \]
\[ C + \{ \text{modid } \mapsto M \} \vdash \text{sigexp} _2 \Rightarrow \Lambda T'.M' \]
\[ T'' \cap (T \cup \text{tynames} \ M \cup \text{tynames}(\Lambda T'.M')) = \emptyset \]
\[ \varphi = \{ u^K \mapsto u_1^{K_1} \cdots u_n^{K_n} \mapsto t_1 \cdots t_n; u^K \in T' \} \]
\[ T'' = \{ u_1^{K_1} \cdots u_n^{K_n}; u^K \in T' \} \]
\[ C \vdash \text{functor modid : sigexp} _1 \rightarrow \text{sigexp} _2 \Rightarrow \Lambda T''.(\forall T.M \rightarrow \exists \varphi(M')) \] (93)
\[ C \vdash \text{sigexp}_1 \Rightarrow \Lambda T, RS \]
\[ T \cap \text{tynames} C = \emptyset \]
\[ C + \{\text{strid} \mapsto RS\} \vdash \text{sigexp}_2 \Rightarrow \Lambda T', RS' \]
\[ T' \cap (T \cup \text{tynames} RS) = \emptyset \]
\[ \varphi(\text{RS}') \gg \varphi(\text{RS}) \]
\[ \text{Supp} \varphi = T \]
\[ T \cap \text{Yield}(\varphi) = \emptyset \]
\[ C \vdash \textbf{rec (strid: sigexp)} \text{sigexp} \Rightarrow \Lambda T', \varphi(\text{RS}, \text{RS}') \]

Comments:

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

(94) An opaque functor signature specifies a set of functors. A functor that “matches” the signature \textbf{functor} (modid : \text{sigexp}_1) \rightarrow \text{sigexp}_2 must be applicable to any actual argument whose type “matches” \text{sigexp}_1. Thus \text{sigexp}_1 specifies the type parameters \( T \) and domain \( M \) of any matching functor. The signature expression \text{sigexp}_2, which is elaborated in the extended context \( C + \{\text{modid} \mapsto M\} \), specifies the range of the matching functor, up to some opaque realisation of \( T' \).

In this way, the type parameters arising \text{sigexp}_1 determine the polymorphism of the specified functors, while the type parameters arising from \text{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 \textbf{functor modid : \text{sigexp}_1} \rightarrow \text{sigexp}_2 must be applicable to any actual argument whose type “matches” \text{sigexp}_1. Thus \text{sigexp}_1 specifies the type parameters \( T \) and domain \( M \) of any matching functor. The range signature expression \text{sigexp}_2, which is elaborated in the extended context \( C + \{\text{modid} \mapsto M\} \), specifies the result of applying such a functor.

If \text{sigexp}_2 elaborates to a signature \( \Lambda T'.M' \), then any type names in \( T' \) represent types that have an unspecified realisation in \text{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 \( \varphi \) to \( M' \) caters for these dependencies. The realisation parameterises each occurrence in \( M' \) of a name \( u \in T' \) 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 \( \Lambda T''.(\forall T.M \rightarrow \exists \emptyset.\varphi(M')) \).

In this way, the type parameters arising \( \text{sigexp}_1 \) determine the polymorphism of the specified functors, while the type parameters arising from \( \text{sigexp}_2 \) index variations in the range of the specified functors. These parameters represent unspecified argument-result type dependencies.

### Specifications

\[ C \vdash \text{spec} \Rightarrow \Lambda T.E \]

\[ C \vdash \text{tyidseq} \Rightarrow (\alpha_1, \ldots, \alpha_k), IE \quad C + IE \vdash \text{valdesc} \Rightarrow VE \]

\[ C \vdash \text{val tyidseq valdesc} \Rightarrow \Lambda \emptyset.\text{Clos}_C VE \text{ in Env} \quad (95) \]

\[ C \vdash \text{typdesc} \Rightarrow \Lambda T.TE \quad \forall (t, VE) \in T, \ t \text{ does not admit equality} \]

\[ C \vdash \text{type typdesc} \Rightarrow \Lambda T.TE \text{ in Env} \quad (96) \]

\[ C \vdash \text{typdesc} \Rightarrow \Lambda T.TE \quad \forall (t, VE) \in T, \ t \text{ admits equality} \]

\[ C \vdash \text{eqtype typdesc} \Rightarrow \Lambda T.TE \text{ in Env} \quad (97) \]

\[ C + TE \vdash \text{datdesc} \Rightarrow VE, TE \]

\[ T = \{ t; (t, VE') \in \text{Ran} TE \} \]

\[ T \cap \text{tynames} C = \emptyset \]

\[ \text{TE maximises equality} \]

\[ C \vdash \text{datatype datdesc} \Rightarrow \Lambda T.(VE, TE) \text{ in Env} \quad (98) \]

\[ C \vdash \text{tyconpath} \Rightarrow (\theta, VE) \]

\[ TE = \{ \text{tycon} \mapsto (\theta, VE) \} \]

\[ C \vdash \text{datatype tycon = datatype tyconpath} \Rightarrow \Lambda \emptyset.(VE, TE) \text{ in Env} \quad (99) \]

\[ C \vdash \text{exdesc} \Rightarrow VE \]

\[ C \vdash \text{exception exdesc} \Rightarrow \Lambda \emptyset.VE \text{ in Env} \quad (100) \]
\[ \begin{align*}
C \vdash \text{strdesc} & \Rightarrow \Lambda T.\text{SE} \\
C \vdash \text{structure strdesc} & \Rightarrow \Lambda T.\text{SE} \text{ in Env} 
\end{align*} \] (101)
\[ \begin{align*}
C \vdash \text{fundesc} & \Rightarrow \Lambda T.\text{FE} \\
C \vdash \text{functor fundesc} & \Rightarrow \Lambda T.\text{FE} \text{ in Env} 
\end{align*} \] (102)
\[ \begin{align*}
C \vdash \text{sigbind} & \Rightarrow \Lambda \emptyset \text{GE} \\
C \vdash \text{signature sigbind} & \Rightarrow \Lambda \emptyset \text{GE} \text{ in Env} 
\end{align*} \] (103)
\[ \begin{align*}
C \vdash \text{sigexp} & \Rightarrow \Lambda (\text{FE, SE, TE, VE}) \\
C \vdash \text{include sigexp} & \Rightarrow \Lambda (\emptyset, \text{FE, SE, TE, VE}) 
\end{align*} \] (104)
\[ \begin{align*}
C \vdash \Rightarrow \Lambda \emptyset (\emptyset, \emptyset, \emptyset, \emptyset, \emptyset) 
\end{align*} \] (105)
\[ \begin{align*}
C \vdash \text{spec}_1 & \Rightarrow \Lambda T_1.E_1 \\
C + E_1 \vdash \text{spec}_2 & \Rightarrow \Lambda T_2.E_2 \\
T_2 \cap (T_1 \cup \text{tynames } E_1) & = \emptyset \\
C \vdash \text{spec}_1 \langle ; \rangle \text{ spec}_2 & \Rightarrow \Lambda T_1 \cup T_2. E_1 + E_2 
\end{align*} \] (106)
\[ \begin{align*}
C \vdash \text{spec} & \Rightarrow \Lambda T. E \\
(\emptyset, E) \vdash \text{longtycon}_i & \Rightarrow (t_i, \text{VE}_i), \ i = 1..n \\
t_1, \ldots, t_n & \text{ have arity } k \\
\{t_1, \ldots, t_n\} \subseteq T & \Rightarrow \varphi = \{t_1 \mapsto t, \ldots, t_n \mapsto t\} \\
C \vdash \text{spec sharing type longtycon}_1 = \cdots = \text{longtycon}_n & \Rightarrow \Lambda T \setminus (\{t_1, \ldots, t_n\} \setminus \{t\}). \varphi (E) 
\end{align*} \] (107)

Comments:

(95) \(\text{VE}\) is determined by \(C\) and \(\text{valdesc}\).

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

(100) \(\text{VE}\) is determined by \(C\) and \(\text{exdesc}\) and contains monotypes only.

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

Value Descriptions

\[ C \vdash \text{valdesc} \Rightarrow VE \]

\[ C \vdash ty \Rightarrow \tau \quad \langle C \vdash \text{valdesc} \Rightarrow VE \rangle \]

\[ C \vdash \text{vid} : ty \text{ (and valdesc)} \Rightarrow \{ \text{vid} \mapsto (\tau, v) \} \langle + VE \rangle \] (108)

Type Descriptions

\[ C \vdash \text{typdesc} \Rightarrow \Lambda.T.TE \]

\[ C \vdash \text{tyidseq} \Rightarrow (\alpha^k, VE) \]

\[ t \text{ has arity } k \]

\[ \langle C \vdash \text{typdesc} \Rightarrow \Lambda.T.TE \quad t \notin T \rangle \]

\[ C \vdash \text{tyidseq tycon (and typdesc)} \Rightarrow \Lambda\{t\}\langle\cup T\rangle.\{\text{tycon} \mapsto (t, \{\})\}\langle + TE \rangle \] (109)

Comment: Note that the value environment in the resulting type structure
must be empty. For example, \texttt{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

\[ C \vdash \text{datdesc} \Rightarrow VE, TE \]

\[ C \vdash \text{tyidseq} \Rightarrow (\alpha^k, IE) \]

\[ C + IE, \alpha^k t \vdash \text{condesc} \Rightarrow VE \]

\[ \langle C \vdash \text{datdesc} \Rightarrow VE', TE' \rangle \quad \forall (t', VE'') \in \text{Ran } TE, t \neq t' \]

\[ C \vdash \text{tyidseq tycon = condesc (and datdesc)} \Rightarrow \]

\[ \text{Clos}_{C,VE}\langle + VE' \rangle, \{\text{tycon} \mapsto (t, \text{Clos}_{C,VE})\} \langle + TE' \rangle \] (110)

Constructor Descriptions

\[ C, \tau \vdash \text{condesc} \Rightarrow VE \]

\[ \langle C \vdash ty \Rightarrow \tau' \rangle \quad \langle \langle C, \tau \vdash \text{condesc} \Rightarrow VE \rangle \rangle \]

\[ \quad \langle C, \tau \vdash \text{vid of ty} \rangle \langle \langle \text{ | condesc} \rangle \Rightarrow \}

\[ \quad \{\text{vid} \mapsto (\tau, c)\} \langle + \{\text{vid} \mapsto (\tau' \rightarrow \tau, c)\} \} \rangle \langle + VE \} \rangle \] (111)

Exception Descriptions

\[ C \vdash \text{exdesc} \Rightarrow VE \]

\[ \langle C \vdash ty \Rightarrow \tau \rangle \quad \langle \langle C \vdash \text{exdesc} \Rightarrow VE \rangle \rangle \]

\[ \quad \langle C \vdash \text{vid of ty} \rangle \langle \langle \text{and exdesc} \rangle \Rightarrow \}

\[ \quad \{\text{vid} \mapsto (\text{exn}, e)\} \langle + \{\text{vid} \mapsto (\tau \rightarrow \text{exn}, e)\} \rangle \rangle \langle + VE \} \rangle \] (112)
Structure Descriptions

\[ C \vdash \text{strdesc} \Rightarrow \Lambda.T.SE \]
\[
\begin{align*}
C \vdash \text{sigexp} & \Rightarrow \Lambda.T.RS \\
(C \vdash \text{strdesc} & \Rightarrow \Lambda.T'.SE) \\
(T \cap (T' \cup \text{tynames } SE) & = \emptyset) \\
(T' \cap (\text{tynames } RS) & = \emptyset) \\
\hline
C \vdash \text{strid} : \text{sigexp \ (and \ strdesc)} & \Rightarrow \\
\Lambda.T(\cup T').\{\text{strid} \mapsto RS\} \langle + SE\rangle
\end{align*}
\]

Functor Descriptions

\[ C \vdash \text{fundesc} \Rightarrow \Lambda.T.FE \]
\[
\begin{align*}
C \vdash \text{sigexp} & \Rightarrow \Lambda.T.F \\
(C \vdash \text{fundesc} & \Rightarrow \Lambda.T'.FE) \\
(T \cap (T' \cup \text{tynames } FE) & = \emptyset) \\
(T' \cap (\text{tynames } F) & = \emptyset) \\
\hline
C \vdash \text{funid} : \text{sigexp \ (and \ fundesc)} & \Rightarrow \\
\Lambda.T(\cup T').\{\text{funid} \mapsto F\} \langle + FE\rangle
\end{align*}
\]
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 \( \pi \) 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 \( \text{fun tyidseq fvalbind} \). The mixed forms \( \text{val tyidseq rec fvalbind} \), \( \text{val tyidseq fvalbind} \) and \( \text{fun tyidseq valbind} \) 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 \texttt{withtype} , the identifiers bound by \texttt{datbind} and by \texttt{typbind} must be distinct. Then the transformed binding \texttt{datbind'} in the equivalent form is obtained from \texttt{datbind} by expanding out all the definitions made by \texttt{typbind}. More precisely, if \texttt{typbind} is

\[
\text{tyidseq}_1 \\text{tycon}_1 = t_1 \quad \text{and} \quad \cdots \quad \text{and} \quad \text{tyidseq}_n \\text{tycon}_n = t_n
\]

then \texttt{datbind'} is the result of simultaneous replacement (in \texttt{datbind}) of every type expression \( \text{tyseq}_i \text{tycon}_i \) \( (1 \leq i \leq n) \) by the corresponding defining expression

\[
t_i \{\text{tyseq}_i/\text{tyidseq}_i\}
\]

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

\[ spec \; \text{sharing} \; \text{longstrid}_1 = \cdots = \text{longstrid}_k \]

is a derived form whose equivalent form is

\[
\begin{align*}
\text{spec} \\
\text{sharing type} \; \text{longtycon}_1 &= \text{longtycon'}_1 \\
\cdots
\text{sharing type} \; \text{longtycon}_m &= \text{longtycon'}_m
\end{align*}
\]

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 \textbf{include} specifications. Then the equivalent form contains all type-sharing constraints of the form

\[ \text{sharing type} \; \text{longstrid}_i, \text{longtycon} = \text{longstrid}_j, \text{longtycon} \]

(\( 1 \leq i < j \leq k \)), 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

#### Expressions $exp$

<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>{}</td>
</tr>
<tr>
<td>$(exp_1, \ldots, exp_n)$</td>
<td>${1=exp_1, \ldots, n=exp_n}$ $(n \geq 2)$</td>
</tr>
<tr>
<td># lab</td>
<td>fn ${lab=vid, \ldots} =&gt; vid$ $(vid\ new)$</td>
</tr>
<tr>
<td>case $exp$ of match</td>
<td>$(fn\ match)(exp)$</td>
</tr>
</tbody>
</table>
| 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 $(n \geq 1)$ |
| $(exp_1; \ldots; exp_n; exp)$ | case $exp_1$ of (_) $=>$
|                        | ... |
|                        | case $exp_n$ of (_) $=>$ $exp$ |
| let dec in $exp_1; \ldots; exp_n$ end | let dec in $(exp_1; \ldots; exp_n)$ 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 |
| $[exp_1, \ldots, exp_n]$ | $exp_1 :: \ldots :: exp_n :: nil$ $(n \geq 0)$ |

#### Patterns $pat$

<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>{}</td>
</tr>
<tr>
<td>$(pat_1, \ldots, pat_n)$</td>
<td>${1=pat_1, \ldots, n=pat_n}$ $(n \geq 2)$</td>
</tr>
<tr>
<td>$[pat_1, \ldots, pat_n]$</td>
<td>$pat_1 :: \ldots :: pat_n :: nil$ $(n \geq 0)$</td>
</tr>
</tbody>
</table>

#### Pattern Rows $patrow$

<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$vid(:ty)\ as\ pat\ \langle,\ patrow\rangle$</td>
<td>$vid = vid(:ty)\ as\ pat\ \langle,\ patrow\rangle$</td>
</tr>
</tbody>
</table>

#### Type Expressions $ty$

<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ty_1 * \ldots * ty_n$</td>
<td>${1:ty_1, \ldots, n:ty_n}$ $(n \geq 2)$</td>
</tr>
</tbody>
</table>

Figure 14: Derived forms of Expressions

Figure 15: Derived forms of Patterns and Type Expressions
### Derived Forms: Function-value Bindings and Declarations

<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Function-value Bindings fvalbind</strong></td>
<td></td>
</tr>
<tr>
<td>\langle \text{op} \rangle vid atpat_{11} \cdots atpat_{1n} \langle \colon ty \rangle = \text{exp}_1</td>
<td>\langle \text{op} \rangle \text{vid} = \text{fn} \text{vid}_1 \Rightarrow \cdots \Rightarrow \text{fn} \text{vid}_n \Rightarrow \text{case (vid}<em>1, \cdots, \text{vid}<em>n \rangle \text{ of (atpat}</em>{11}, \cdots, \text{atpat}</em>{1n} \rangle \Rightarrow \text{exp}_1 \langle \colon ty \rangle}</td>
</tr>
<tr>
<td>\langle \text{op} \rangle vid atpat_{21} \cdots atpat_{2n} \langle \colon ty \rangle = \text{exp}_2</td>
<td>\langle \text{op} \rangle \text{vid} atpat_{21} \cdots atpat_{2n} \langle \colon ty \rangle = \text{exp}_2 \langle \colon ty \rangle}</td>
</tr>
</tbody>
</table>
| \langle \text{op} \rangle vid atpat_{m1} \cdots atpat_{mn} \langle \colon ty \rangle = \text{exp}_m \begin{array}{l}
\langle \text{and fvalbind} \rangle \\
\text{with deg end}
\end{array} | \langle \text{op} \rangle \text{vid} atpat_{m1} \cdots atpat_{mn} \langle \colon ty \rangle = \text{exp}_m \langle \colon ty \rangle} |

\begin{array}{l}
\text{\(m, n \geq 1; \text{vid}_1, \cdots, \text{vid}_n \text{ distinct and new) }
\end{array}

<table>
<thead>
<tr>
<th>Declarations dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{fun tyidseq fvalbind}</td>
</tr>
<tr>
<td>\text{datatype datbind withtype typbind}</td>
</tr>
<tr>
<td>\text{abstype datbind withtype typbind}</td>
</tr>
<tr>
<td>\text{with deg end}</td>
</tr>
</tbody>
</table>

(see note in text concerning datbind$'$)

Figure 16: Derived forms of Function-value Bindings and Declarations
<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
</table>
| **Structure Bindings** `strbind` | **Equivalent Form**  
| `strid : sigexp = modexp (and strbind)` | `strid = modexp : sigexp (and strbind)` |
| `strid :> sigexp = modexp (and strbind)` | `strid = modexp :> sigexp (and strbind)` |

<table>
<thead>
<tr>
<th><strong>Module Expressions</strong> <code>modexp</code></th>
<th><strong>Functor Bindings</strong> <code>funbind</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(</code> <code>dec</code> <code>)</code></td>
<td><code>funid (</code> <code>spec</code> <code>)</code> <code>=&gt; sigexp</code>)`</td>
</tr>
<tr>
<td><code>(</code> <code>dec</code> <code>)</code></td>
<td><code>funid (</code> <code>spec</code> <code>)</code> <code>=&gt; sigexp</code>)`</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Functor Bindings</strong> <code>funbind</code></th>
<th><strong>Functor Bindings</strong> <code>funbind</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>funid (⟨ ⟨modid1 : sigexp1⟩⟩1 : sigexp1)1 =&gt;</code></td>
<td><code>funid = functor (⟨ ⟨modid1 : sigexp1⟩⟩1 =&gt;</code></td>
</tr>
<tr>
<td><code>:: modexp</code></td>
<td><code>:: modexp</code></td>
</tr>
<tr>
<td><code>funcmod =&gt;</code></td>
<td><code>funcmod =&gt;</code></td>
</tr>
<tr>
<td><code>⟨ ⟨funbind⟩⟩</code></td>
<td><code>⟨ ⟨funbind⟩⟩</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Functor Bindings</strong> <code>funbind</code></th>
<th><strong>Functor Bindings</strong> <code>funbind</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>funid (⟨ ⟨modid1 : sigexp1⟩⟩1 : sigexp1)1 =&gt;</code></td>
<td><code>funid = functor (⟨ ⟨modid1 : sigexp1⟩⟩1 =&gt;</code></td>
</tr>
<tr>
<td><code>:: modexp</code></td>
<td><code>:: modexp</code></td>
</tr>
<tr>
<td><code>funcmod =&gt;</code></td>
<td><code>funcmod =&gt;</code></td>
</tr>
<tr>
<td><code>⟨ ⟨funbind⟩⟩</code></td>
<td><code>⟨ ⟨funbind⟩⟩</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Programs</strong> <code>program</code></th>
<th><strong>Programs</strong> <code>program</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>exp ; ⟨program⟩</code></td>
<td><code>val it = exp ; ⟨program⟩</code></td>
</tr>
</tbody>
</table>

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

**Specifications** \( \textit{spec} \)

- \( \text{type tyidseq tycon = ty} \)
- \( \text{type tyidseq}_1 tycon_1 = ty_1 \)
  - and \( \cdots \)
  - and \( tyidseq_n tycon_n = ty_n \)
- \( \text{include sigid}_1 \cdots \text{sigid}_n \)
- \( \text{spec sharing longstrid}_1 = \cdots = \text{longstrid}_k \)

**Equivalent Form**

- \( \text{include sig type tyidseq tycon} \)
- \( \text{end where type tyidseq tycon = ty} \)
- \( \text{type tyidseq}_1 tycon_1 = ty_1 \)
- \( \text{type } \cdots \)
- \( \text{type tyidseq}_n tycon_n = ty_n \)
- \( \text{include sigid}_1; \cdots; \text{include sigid}_n \)
- \( \text{spec sharing longtycon}_1 = \cdots = \text{longtycon}_1' \)
- \( \text{sharing type longtycon}_m = \text{longtycon}_m' \)

(see note in text concerning \( \text{longtycon}_1, \ldots, \text{longtycon}_m' \))

### Signature Expressions \( \textit{sigexp} \)

- \( \text{where type tyidseq}_1 \text{longtycon}_1 = ty_1 \)
  - and \( \cdots \)
  - and \( tyidseq_n \text{longtycon}_n = ty_n \)
- \( \text{sigexp} \)
- \( \text{where type tyidseq}_1 \text{longtycon}_1 = ty_1 \)
- \( \text{where type } \cdots \)
- \( \text{where type tyidseq}_n \text{longtycon}_n = ty_n \)

Figure 18: Derived forms of Specifications and Signature Expressions