The Semantics of Standard ML
Version 1

by

Robert Harper    Robin Milner
Mads Tofte

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

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

August 1987

Copyright © 1987, LFCS
Preface to Version 1

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

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

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

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

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

Any future Version of this document will indicate precisely how it differs from its predecessor.
The Semantics of Standard ML
Version 1

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

August 13, 1987

Contents

1 Introduction 4

2 Syntax of the Core 6
2.1 Reserved Words 6
2.2 Special constants 6
2.3 Comments 7
2.4 Identifiers 7
2.5 Lexical analysis 8
2.6 Infixed operators 8
2.7 Derived Forms 9
2.8 Grammar 9
2.9 Syntactic Restrictions 13

3 Syntax of Modules 14
3.1 Reserved Words 14
3.2 Identifiers 14
3.3 Infixed operators 14
3.4 Grammar for Modules 15
3.5 Syntactic Restrictions 16
3.6 Pure Functor Forms 18
3.7 Closure Restrictions 19
A Appendix: Derived Forms 63
B Appendix: Full Grammar of the Core 66
C Appendix: The Initial Static Basis 70
D Appendix: The Initial Dynamic Basis 73
1 Introduction

This document provides a complete formal description of Standard ML.

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

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

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

A further factoring of the formal description is possible, because of the structure of the language. ML consists of a lower level, called the Core language (or Core, for short) and an upper level concerned with programming-in-the-large, called Modules. The Core is a complete language in its own right, and its embedding in the full language is simple; therefore each of the three parts of the formal description is further divided into two — one for the Core, and one for Modules.

The Core provides many phrase classes, for programming convenience. But about half of these classes are derived forms, whose meaning can be given by translation into the other half which we call the Bare language. (There are no derived forms for Modules). Thus each of the three parts for the Core treats only the bare language; the derived forms are treated in Appendix A. A full grammar for the Core including derived forms is presented in Appendix B.
In Appendix C and D the initial basis is detailed. This basis, divided into its static and dynamic parts, contains the static and dynamic meanings of all predefined identifiers.

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

\[ A \vdash \text{phrase} \Rightarrow A' \]

to be inferred, where \( A \) is often a basis (static or dynamic) and \( A' \) a semantic object – often a type in the static semantics and a value in the dynamic semantics. One should read such a sentence as follows: "in the basis \( A \), the phrase \text{phrase} evaluates – or elaborates – to the object \( A' \)". Although the rules themselves are formal the semantic objects, particularly the static ones, are the subject of a mathematical theory which is presented in a succinct form in the relevant sections. This theory, particularly the theory of types and signatures, will benefit from a more pedagogic treatment in other publications; the treatment here is probably the minimum required to understand the meaning of the rules.

The robustness of the semantics depends upon theorems. Some of these are stated but not proved; others are presented as "claims" rather than theorems – often they have been proved for a skeletal language, and although we are confident of their truth their proofs in the context of the full language will present an interesting challenge to a computer-assisted proof methodology, to attain complete certainty.
2 Syntax of the Core

2.1 Reserved Words

The following are the reserved words used in the Core. They may not (except =) be used as identifiers. In this document the alphabetic reserved words are always shown in typewriter font.

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

2.2 Special constants

An integer constant is any non-empty sequence of digits, possibly preceded by a negation symbol (\`).

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

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

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

The formatting characters are a subset of the non-printable characters including at least space, tab, newline, formfeed. The last form allows long strings to be written on more than one line, by writing \ at the end of one line and at the start of the next.
Var (value variables) long
Con (value constructors) long
Exn (exception names) long
TyVar (type variables) long
TyCon (type constructors) long
Lab (record labels) long
StrId (structure identifiers) long

Figure 1: Identifiers

2.3 Comments

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

2.4 Identifiers

The classes of identifiers for the Core are shown in Figure 1. We use var, tyvar to range over Var, TyVar etc. For each class $X$ marked "long" there is also a class

$$\text{Long}X = \text{StrId}^* \times X$$

If $x$ ranges over $X$, then $\text{long}x$, or $\text{strid}_1, \ldots, \text{strid}_k.x$, $k \geq 0$, ranges over Long$X$. These long identifiers constitute the only link between the Core and the language constructs for Modules; by omitting them, and the open declaration, we obtain the Core as a complete programming language in its own right. (The corresponding adjustment to the Core static and dynamic semantics is simply to omit structure environments $SE$.)

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

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

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

A type variable tyvar may be any alphanumeric identifier starting with a prime; the subclass EtyVar of TyVar, the equality type variables, consists of
those which start with two or more primes. The other six classes (Var, Con, Exn, TyCon, Lab and SStrId) are represented by identifiers not starting with a prime; the class Lab is also extended to include the numeric labels 1 2 3 \ldots.

TyVar is therefore disjoint from the other six classes. Otherwise, the syntax class of an occurrence of identifier \textit{id} in a Core phrase is determined thus:

1. Immediately before "." – i.e. in a long identifier – or in an open declaration, \textit{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, \textit{id} is a record label.

3. Elsewhere in types \textit{id} is a type constructor, and must be within the scope of the type binding or datatype binding which introduced it.

4. Elsewhere \textit{id} is an exception name if it occurs immediately after \texttt{raise}, at the start of a handler rule \texttt{hrule}, or within an exception declaration or specification.

5. Elsewhere, \textit{id} is a value constructor if it occurs in the scope of a datatype binding which introduced it as such, otherwise it is a value variable.

It follows from the last rule that no value declaration can make a "hole" in the scope of a value constructor by introducing the same identifier as a variable; this is because, in the scope of the declaration which introduces \textit{id} as a value constructor, any occurrence of \textit{id} in a pattern is interpreted as the constructor and not as the binding occurrence of a new variable.

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

2.5 Lexical analysis

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

2.6 Infixed operators

An identifier may be given \texttt{infix status} by the \texttt{infix} or \texttt{infixr} directive, which may occur as a declaration; this status only pertains to its use as a \texttt{var} or
a \textit{con} within the scope (see below) of the directive. If \textit{id} has infix status, then \textit{"exp\_1 id exp\_2"} (resp. \textit{"pat\_1 id pat\_2"}) may occur – in parentheses if necessary – wherever the application \textit{"id\{1=exp\_1,2=exp\_2\}"} or its derived form \textit{"id(exp\_1, exp\_2)"} (resp \textit{"id(pat\_1, pat\_2)"}) would otherwise occur. On the other hand, non-infixed occurrences of \textit{id} must be prefixed by the keyword \textit{op}. Infix status is cancelled by the \textit{nonfix} directive. We refer to the three directives collectively as \textit{fixity directives}.

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

\[
infix\ (d)\ id_1 \ldots id_n
\]

\[
infixr\ (d)\ id_1 \ldots id_n
\]

\[
nonfix\ id_1 \ldots id_n
\]

where \((d)\) is an optional decimal digit \(d\) indicating binding precedence. A higher value of \(d\) indicates tighter binding; the default is 0. \textit{infix} and \textit{infixr} dictate left and right associativity respectively; association is always to the left for different operators of the same precedence. The precedence of infix operators relative to other expression and pattern constructions is given in Appendix B.

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

\[
let\ dec\ in\ \ldots\ end
\]

\[
local\ dec\ in\ \ldots\ end
\]

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

These directives and \textit{op} are omitted from the semantic rules, since they effect only parsing.

\section{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 \textit{bare language}. These derived forms, and their equivalent forms in the bare language, are given in Appendix A.

\section{2.8 Grammar}

The phrase classes for the Core are shown in Figure 2. We use the variable \textit{atexp} to range over \textit{AtExp}, etc.
AtExp  atomic expressions
ExpRow  expression rows
Exp     expressions
Match   matches
Mrule   match rules
Handler handlers
Hrule   handler rules

Dec     declarations
ValBind value bindings
TypBind type bindings
DatBind datatype bindings
Constrs datatype constructions
ExnBind exception bindings

AtPat   atomic patterns
PatRow  pattern rows
Pats    patterns

Ty      type expressions
TyRow   type expression rows

Figure 2: Core Phrase Classes

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

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

\[
xseq ::= x \quad \text{(singleton sequence)}
\]
\[
\quad (x) \quad \text{(empty sequence)}
\]
\[
\quad (x_1, \ldots, x_n) \quad \text{(sequence, } n \geq 1)\]

(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.
- L (resp. R) means left (resp. right) association.
• The syntax of types binds more tightly that that of expressions.

• Each iterated construct (e.g. `match`, `handler`, ...) extends as far right as possible; thus, parentheses may be needed around an expression which terminates with a match, e.g. “fn match”, if this occurs within a larger match.

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

\[
\begin{align*}
atexp & ::= (op) longvar & \text{value variable} \\
      & | (op) longcon & \text{value constructor} \\
      & | \{ \text{exprow} \} & \text{record} \\
      & | \text{let dec in exp end} & \text{local declaration} \\
      & | (exp) & \\
\text{exprow} & ::= lab = exp ( , exprow) & \text{expression row} \\
\text{exp} & ::= atexp & \text{atomic} \\
      & | exp atexp & \text{application (L)} \\
      & | exp_1 \text{id} exp_2 & \text{infixed application} \\
      & | exp : ty & \text{typed (L)} \\
      & | exp handle handler & \text{handle exception} \\
      & | \text{raise longzn with exp} & \text{raise exception} \\
      & | fn match & \text{function} \\
\text{match} & ::= mrule ( \mid match) \\
\text{mrule} & ::= pat => \text{exp} \\
\text{handler} & ::= hrule ( \mid | \text{handler}) \\
\text{hrule} & ::= \text{longzn with match} \\
      & | ? => \text{exp}
\end{align*}
\]

Figure 3: Grammar: Expressions, Matches and Handlers
\[\text{def} := \text{val valbind} \quad \text{value declaration}
\text{type typbind} \quad \text{type declaration}
\text{datatype datbind} \quad \text{datatype declaration}
\text{abstype datbind with dec end} \quad \text{abstype declaration}
\text{exception exnbind} \quad \text{exception declaration}
\text{local dec}_1 \text{ in } \text{dec}_2 \text{ end} \quad \text{local declaration}
\text{open longdist}_1 \ldots \text{longdist}_n \quad \text{open declaration (n } \geq 1)\text{ empty declaration}
\text{sequential declaration}
\text{infix (L) directive}
\text{infix (R) directive}
\text{nonfix directive}
\text{valbind} := \text{pat} = \text{exp} \langle \text{and valbind} \rangle
\text{rec valbind}
\text{typbind} := \text{tyvarseq tycon } = \text{ty} \langle \text{and typbind} \rangle
\text{datbind} := \text{tyvarseq tycon } = \text{constrs} \langle \text{and datbind} \rangle
\text{constrs} := \langle \text{op}\rangle \text{con} \langle \text{of } ty \rangle \langle | \text{constrs} \rangle
\text{exnbind} := \text{exn } \langle : ty \rangle = \text{longexn} \langle \text{and exnbind} \rangle
\]

Figure 4: Grammar: Declarations and Bindings

\[\text{atpat} := \_ \quad \text{wildcard}
\langle \text{op}\rangle \text{var} \quad \text{variable}
\text{longcon} \quad \text{constant}
\{ \langle \text{patrow} \rangle \} \quad \text{record}
( \text{pat} ) \quad \text{(patrow)} := \ldots \quad \text{wildcard}
\text{lab} = \text{pat} (, \text{patrow}) \quad \text{pattern row}
\text{pat} \quad \text{atpat} \quad \text{atomic}
\langle \text{op}\rangle \text{longcon atpat} \quad \text{construction}
\text{pat}_1 \text{ con pat}_2 \quad \text{infixed construction}
\text{pat} : \text{ty} \quad \text{typed}
\langle \text{op}\rangle \text{var} (, \text{ty}) \text{as pat} \quad \text{layered}
\]

Figure 5: Grammar: Patterns

12
\[ ty ::= tyvar \quad \text{type variable} \]
\[ \{ \langle tyrow \rangle \} \quad \text{record type expression} \]
\[ tyseq longtycon \quad \text{type construction} \]
\[ ty \rightarrow ty' \quad \text{function type expression (R)} \]
\[ (ty) \quad \text{type-expression row} \]
\[ tyrow ::= \text{lab}: ty \langle , tyrow \rangle \quad \text{type-expression row} \]

Figure 6: Grammar: Type expressions

2.9 Syntactic Restrictions

- No pattern may contain the same \textit{var} twice. No expression row, pattern row or type row may bind the same \textit{lab} twice.

- No binding \textit{valbind}, \textit{typbind}, \textit{datbind} or \textit{exnbind} may bind the same identifier twice; this applies also to value constructors within a \textit{datbind}.

- In the left side \textit{tyvarseq tycon} of any \textit{typbind} or \textit{datbind}, \textit{tyvarseq} must not contain the same \textit{tyvar} twice. Any \textit{tyvar} occurring within the right side must occur in \textit{tyvarseq}.

- Every non-local exception binding – that is, not localised by \textit{let} or \textit{local} – must be explicitly constrained by a type containing no type variables.
3 Syntax of Modules

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

3.1 Reserved Words

The following are the additional reserved words used in Modules.

eqtype functor include open sharing
sig signature struct structure

3.2 Identifiers

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

3.3 Infixed operators

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

let strdec in ... end
local strdec in ... end
struct strdec end

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

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

The phrase classes for Modules are shown in Figure 7. We use the variable \textit{strezp} to range over StrExp, etc. The conventions adopted in presenting the grammatical

\begin{align*}
\text{StrExp} & \quad \text{structure expressions} \\
\text{StrDec} & \quad \text{structure-level declarations} \\
\text{StrBind} & \quad \text{structure bindings} \\
\text{SigExp} & \quad \text{signature expressions} \\
\text{SigDec} & \quad \text{signature declarations} \\
\text{SigBind} & \quad \text{signature bindings} \\
\text{Spec} & \quad \text{specifications} \\
\text{ValDesc} & \quad \text{value descriptions} \\
\text{TypDesc} & \quad \text{type descriptions} \\
\text{DatDesc} & \quad \text{datatype descriptions} \\
\text{ExnDesc} & \quad \text{exception descriptions} \\
\text{StrDesc} & \quad \text{structure descriptions} \\
\text{SharEq} & \quad \text{sharing equations} \\
\text{FunDec} & \quad \text{functor declarations} \\
\text{FunBind} & \quad \text{functor bindings} \\
\text{FunSigExp} & \quad \text{functor signature expressions} \\
\text{FunSpec} & \quad \text{functor specifications} \\
\text{FunDesc} & \quad \text{functor descriptions} \\
\text{Program} & \quad \text{programs}
\end{align*}

\textbf{Figure 7: Modules Phrase Classes}

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

It should be noted that functor specifications (FunSpec) cannot occur in programs; neither can the associated functor descriptions (FunDesc) and functor signature expressions (FunSigExp). The purpose of a \textit{funspec} is to specify the static attributes (i.e. functor signature) of one or more functors. This will be useful, in fact essential, for separate compilation of functors. If, for example, a functor \( g \) refers to another functor \( f \) then — in order to compile \( g \) in the absence of the declaration of \( f \) — at least the specification of \( f \) (i.e. its functor signature) must be available. At present there is no special grammatical form for a separately compilable “chunk” of text — which we may like to call call a \textit{module} —
containing a fundec together with a funspec specifying its global references. However, below in the semantics for Modules it is defined when a declared functor matches a functor signature specified for it. This determines exactly those functor environments (containing declared functors such as \( f \)) into which the separately compiled "chunk" containing the declaration of \( g \) may be loaded.

\[
\begin{align*}
\text{strexp} \ ::= & \quad \text{struct strdec end} \\
& \quad \text{longstrid} \\
& \quad \text{funid ( strdec )} \\
& \quad \text{let strdec in strexp end} \\
\text{strdec} \ ::= & \quad \text{dec} \\
& \quad \text{structure strbind} \\
& \quad \text{local strdec}_1 \text{ in strdec}_2 \text{ end} \\
& \quad \text{strdec}_1 ( ; ) \text{ strdec}_2 \\
\text{strbind} \ ::= & \quad \text{strid ( ; sigexp ) = strexp (and strbind)} \\
\text{sigexp} \ ::= & \quad \text{sig spec end} \\
& \quad \text{sigid} \\
\text{sigdec} \ ::= & \quad \text{signature sigbind} \\
& \quad \text{sigdec}_1 ( ; ) \text{ sigdec}_2 \\
\text{sigbind} \ ::= & \quad \text{sigid = sigexp (and sigbind)}
\end{align*}
\]

Figure 8: Grammar: Structures and Signatures

3.5 Syntactic Restrictions

- No binding \text{strbind}, \text{sigbind}, or \text{funbind} may bind the same identifier twice.

- No description \text{valdesc}, \text{typdesc}, \text{datdesc}, \text{exndesc}, \text{strdesc} or \text{fundesc} may describe the same identifier twice.
\[ \begin{align*}
\text{spec} & \quad ::= \quad \text{val valdesc} \\
& \quad \quad \quad \quad \text{type typdesc} \\
& \quad \quad \quad \quad \text{eqtype typdesc} \\
& \quad \quad \quad \quad \text{datatype datdesc} \\
& \quad \quad \quad \quad \text{exception ezndesc} \\
& \quad \quad \quad \quad \text{structure strdesc} \\
& \quad \quad \quad \quad \text{sharing shareq} \\
& \quad \quad \quad \quad \text{local spec}_1 \text{ in spec}_2 \text{ end} \\
& \quad \quad \quad \quad \text{open longstrid}_1 \cdots \text{ longstrid}_n \\
& \quad \quad \quad \quad \text{include sigid}_1 \cdots \text{ sigid}_n \\
\text{spec}_1 \ (:) \ \text{spec}_2
\end{align*} \]

\[ \begin{align*}
\text{valdesc} & \quad ::= \quad \text{var : ty} \ (\text{and valdesc}) \\
\text{typdesc} & \quad ::= \quad \text{tyvareq tycon} \ (\text{and typdesc}) \\
\text{datdesc} & \quad ::= \quad \text{tyvareq tycon} = \text{constrs} \ (\text{and datdesc}) \\
\text{ezndesc} & \quad ::= \quad \text{exn : ty} \ (\text{and ezndesc}) \\
\text{strdesc} & \quad ::= \quad \text{strid : sigezp} \ (\text{and strdesc}) \\
\text{shareq} & \quad ::= \quad \text{longstrid}_1 = \cdots = \text{longstrid}_n \\
& \quad \quad \quad \quad \text{structure sharing} \quad (n \geq 2) \\
& \quad \quad \quad \quad \text{type longtycon}_1 = \cdots = \text{longtycon}_n \\
& \quad \quad \quad \quad \text{type sharing} \quad (n \geq 2) \\
& \quad \quad \quad \quad \text{shareq}_1 \text{ and shareq}_2 \\
& \quad \quad \quad \quad \text{multiple}
\end{align*} \]

Figure 9: Grammar: Specifications
fundec ::= functor funbind

fundec₁ (;) fundec₂

funbind ::= funid ( spec ) ( : sigexp ) = strexp
          ( and funbind )

funsigexp ::= ( spec ) : sigexp

funspec ::= functor fundesc

funspec₁ (;) funspec₂

fundesc ::= funid funsigexp ( and fundesc )

program ::= strdec
         sigdec
         fundec
         program₁ (;) program₂

Figure 10: Grammar: Functors and Programs

3.6 Pure Functor Forms

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

strexp ::= funid ( strexp )

funbind ::= funid ( strid : sigexp ) ( : sigexp' ) = strexp
          ( and funbind )

funsigexp ::= ( strid : sigexp ) : sigexp'

Figure 11: Functor forms defined in the Semantics

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

\[ \textit{funid} \ (\ \textit{strdec} \ ) \]

is translated to the pure form

\[ \textit{funid} \ (\ \textit{struct} \ \textit{strdec} \ \textit{end} \ ) \]

which “wraps up” the \textit{strdec} as a structure. On the other hand the applied form of functor binding,

\[ \textit{funid} \ (\ \textit{spec} \ ) \ (\ : \ \textit{sigexp} \ ) = \textit{strexp} \]

is translated to the pure form

\[ \textit{funid} \ (\ X : \ \textit{sig} \ \textit{spec} \ \textit{end} \ ) \ (\ : \ \textit{sigexp}' \ ) = \textit{let} \ \textit{open} \ X \ \textit{in} \ \textit{strexp} \ \textit{end} \]

(where \( X \) is a structure identifier not previously occurring in the functor binding) which “wraps up” the \textit{spec} as a signature, but compensates in the body to allow direct reference to members of \textit{spec}. The form of \textit{sigexp}' depends on the form of \textit{sigexp}. If \textit{sigexp} is simply a signature identifier \textit{sigid}, then \textit{sigexp}' is also \textit{sigid}; otherwise \textit{sigexp} must take the form \textit{sig spec, end}, and then \textit{sigexp}' is

\[ \textit{sig} \ \textit{local} \ \textit{open} \ X \ \textit{in} \ \textit{spec, end end} \]

Finally, the applied form of functor signature expression,

\[ (\ \textit{spec} \ ) : \ \textit{sigexp} \]

is translated to the pure form

\[ (\ X : \ \textit{sig} \ \textit{spec} \ \textit{end} \ ) : \ \textit{sigexp}' \]

where \textit{sigexp}' is obtained from \textit{sigexp} exactly as for functor binding above.

### 3.7 Closure Restrictions

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

However, in the present version of the language, apart from references to identifiers bound in the initial basis \( B_0 \) (which may occur anywhere), it is required that signatures only refer non-locally to signature identifiers and that functors only refer non-locally to functor and signature identifiers. These restrictions ease separate compilation; however, they may be relaxed in a future version of the language.

More precisely, the restrictions are as follows (ignoring reference to identifiers bound in \( B_0 \)):
• In any signature binding \( \text{sigid} = \text{sigexp} \), the only non-local references in \( \text{sigexp} \) are to signature identifiers.

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

• In any functor binding \( \text{funid} \ (\text{spec}) \ (\text{: sigexp}) = \text{stexp} \), the only non-local references in \( \text{spec} \), \( \text{sigexp} \) and \( \text{stexp} \) are to functor and signature identifiers, except that both \( \text{sigexp} \) and \( \text{stexp} \) may refer to identifiers specified in \( \text{spec} \).

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

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

4.1 Simple Objects

All semantic objects in the static semantics of the entire language are built from identifiers and two further kinds of simple objects: type constructor names and structure names. Type constructor names are the values taken by type constructors; we shall usually refer to them briefly as type names, but they are to be clearly distinguished from type variables and type constructors. Structure names play an active role only in the Modules semantics; they enter the Core semantics only because they appear in structure environments, which (in turn) are needed in the Core semantics only to determine the values of long identifiers. The simple object classes, and the variables ranging over them, are shown in Figure 12. We have included TyVar in the table to make visible the use of \( \alpha \) in the semantics to range over TyVar.

\[
\begin{align*}
\alpha & \text{ or } \texttt{tyvar} \in \text{TyVar} & \text{type variables} \\
t & \in \text{TyName} & \text{type names} \\
m & \in \text{StrName} & \text{structure names}
\end{align*}
\]

Figure 12: Simple Semantic Objects

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

4.2 Compound Objects

When \( A \) and \( B \) are sets FinA denotes the set of finite subsets of \( A \), and \( A \stackrel{\text{fin}}{\rightarrow} B \) 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 Figure 13.

\[
\begin{align*}
\tau & \in \text{Type} = \text{TyVar} \cup \text{RecType} \cup \text{FunType} \cup \text{ConsType} \\
(r_1, \ldots, r_k) & \text{ or } r^{(k)} \in \text{Type}^k \\
(\alpha_1, \ldots, \alpha_k) & \text{ or } \alpha^{(k)} \in \text{TyVar}^k \\
\theta & \in \text{RecType} = \text{Lab}^\text{fn} \text{ Type} \\
\tau \to \tau' & \in \text{FunType} = \text{Type} \times \text{Type} \\
\text{ConsType} & = \bigcup_{k \geq 0} \text{ConsType}^{(k)} \\
\tau^{(k)}t & \in \text{ConsType}^{(k)} = \text{Type}^k \times \text{Name}^{(k)} \\
\theta & \text{ or } \Lambda \alpha^{(k)}\tau \in \text{TypeFcn} = \bigcup_{k \geq 0} \text{TyVar}^k \times \text{Type} \\
\sigma & \text{ or } \forall \alpha^{(k)}\tau \in \text{TypeScheme} = \bigcup_{k \geq 0} \text{TyVar}^k \times \text{Type} \\
S & \text{ or } (m, E) \in \text{Str} = \text{Name} \times \text{Env} \\
(\theta, CE) & \in \text{TyStr} = \text{TypeFcn} \times \text{ConEnv} \\
SE & \in \text{StrEnv} = \text{StrId}^\text{fn} \text{ Str} \\
TE & \in \text{TyEnv} = \text{TyCon}^\text{fn} \text{ TyStr} \\
CE & \in \text{ConEnv} = \text{Con}^\text{fn} \text{ TypeScheme} \\
VE & \in \text{VarEnv} = (\text{Var} \cup \text{Con})^\text{fn} \text{ TypeScheme} \\
EE & \in \text{ExnEnv} = \text{Exn}^\text{fn} \text{ Type} \\
E & \text{ or } (SE, TE, VE, EE) \in \text{Env} = \text{StrEnv} \times \text{TyEnv} \times \text{VarEnv} \times \text{ExnEnv} \\
T & \in \text{TyNameSet} = \text{Fin}(\text{TyName}) \\
C & \text{ or } T, E \in \text{Context} = \text{TyNameSet} \times \text{Env}
\end{align*}
\]

Figure 13: Compound Semantic Objects

Note that \( A \) and \( V \) bind type variables. For any semantic object \( A \), tynames \( A \) and tyvars \( A \) denote respectively the set of type names and the set of type variables occurring free in \( A \).
4.3 Projection, Injection and Modification

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

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

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

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

$$(VE of (SE of \ldots(SE of (SE of E)strid_1)strid_2) \ldots)strid_k)con.$$ 

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

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

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

Finally, we frequently need to modify a context $C$ by an environment $E$ (or a type environment $TE$ say), at the same time extending $T$ of $C$ to include the type names of $E$ (or of $TE$ say). We therefore define $C \oplus TE$, for example, to mean $C + (\text{tynames}TE,TE)$.

4.4 Types and Type functions

A type $\tau$ is an equality type, or admits equality, if it is of one of the forms

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

A type function \(\theta = \Lambda \alpha^{(k)} . \tau\) has arity \(k\); it must be closed – i.e. \(\text{tyvars}(\tau) \subseteq \alpha^{(k)}\) – and the bound variables must be distinct. Two type functions are considered equal if they only differ in their choice of bound variables (alpha-conversion). If \(t\) has arity \(k\), then we write \(t\) to mean \(\Lambda \alpha^{(k)} . \alpha^{(k)} t\) (eta-conversion); thus \(\text{TyName} \subseteq \text{TypeFcn}\). \(\theta = \Lambda \alpha^{(k)} . \tau\) is an equality type function, or admits equality, if when the type variables \(\alpha^{(k)}\) are chosen to admit equality then \(\tau\) also admits equality.

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

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

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

### 4.5 Type Schemes

A type scheme \(\sigma = \forall \alpha^{(k)} . \tau\) generalises a type \(\tau'\), written \(\sigma \succ \tau'\), if \(\tau' = \tau / \alpha^{(k)}\) 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. It can be shown that \(\sigma = \sigma'\) iff \(\sigma \succ \sigma'\) and \(\sigma' \succ \sigma\).

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

### 4.6 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 \(\text{Clos}_{1}(\tau)\) to \(\text{Clos}(\tau)\). If the range of a variable environment \(\text{VE}\) contains only types (rather than arbitrary type schemes) we set

\[
\text{Clos}_{A} \text{VE} = \{\text{var} \mapsto \text{Clos}_{A}(\tau) ; \text{VE}(\text{var}) = \tau\}
\]

with a similar definition for \(\text{Clos}_{A} \text{CE}\).
4.7 Type Environments and Well-formedness

A type environment takes the form
\[ TE = \{\text{tycon}_i \mapsto (\theta_i, CE_i) : 1 \leq i \leq k\} \]

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

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

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

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

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

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

For any \( TE \) as above, \( \text{Abs} \ TE \) is the type environment \( \{\text{tycon}_1 \mapsto (\theta_1, \emptyset), \ldots\} \) in which all constructor environments \( CE_i \) have been replaced by the empty map. The effect is to convert each \( \text{tycon}_i \) into an ordinary type constructor.
4.8 Inference Rules

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

\[ A \vdash \text{phrase} \Rightarrow A' \]

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

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

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

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

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

Atomic Expressions

\[
\begin{align*}
C(\text{longvar}) &\gg r & \frac{C \vdash \text{longvar} \Rightarrow r}{(1)} \\
C(\text{longcon}) &\gg r & \frac{C \vdash \text{longcon} \Rightarrow r}{(2)} \\
\langle C \vdash \text{exprob} \Rightarrow q \rangle & & \frac{\langle C \vdash \{ \text{exprob} \} \rangle \Rightarrow \{ \} (+ q) \text{ in Type}}{(3)}
\end{align*}
\]
\[
\frac{C \vdash \text{dec} \Rightarrow E \quad C \oplus E \vdash \text{exp} \Rightarrow \tau}{C \vdash \text{let dec in } \text{exp end} \Rightarrow \tau}
\]

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

Comments:

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

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

Expression Rows

\[
\frac{C \vdash \text{exp} \Rightarrow \tau \quad \langle C \vdash \text{exprow} \Rightarrow \emptyset \rangle}{C \vdash \text{lab} = \text{exp} (\_, \text{exprow}) \Rightarrow \{ \text{lab} \mapsto \tau \}(\emptyset \emptyset)}
\]

Expressions

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

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

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

\[
\frac{C \vdash \text{exp} \Rightarrow \tau \quad C \vdash \text{handler} \Rightarrow \tau}{C \vdash \text{exp handle handler} \Rightarrow \tau}
\]

\[
\frac{C(\text{longzn}) = \tau \quad C \vdash \text{exp} \Rightarrow \tau}{C \vdash \text{raise longzn with exp} \Rightarrow \tau'}
\]

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

Comments:

(7) The relational symbol \vdash is overloaded for all syntactic classes (here atomic expressions and expressions).
(9) Here $\tau$ is determined by $C$ and $ty$.

(11) Note that $\tau'$ does not occur in the premises; thus a `raise` expression has "arbitrary" type.

**Matches**

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

\[
C \vdash \text{mrule} \Rightarrow \tau \quad (C \vdash \text{match} \Rightarrow \tau) \\
C \vdash \text{mrule} (\mid \text{match}) \Rightarrow \tau
\]  \hspace{1cm} (13)

**Match Rules**

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

\[
C \vdash \text{pat} \Rightarrow \text{exp} \Rightarrow \tau \rightarrow \tau'
\]  \hspace{1cm} (14)

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

**Handlers**

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

\[
C \vdash \text{hrule} \Rightarrow \tau \quad (C \vdash \text{handler} \Rightarrow \tau) \\
C \vdash \text{hrule} (\mid \mid \text{handler}) \Rightarrow \tau
\]  \hspace{1cm} (15)

**Handle Rules**

\[
C(\text{longax}) = \tau' \quad C \vdash \text{match} \Rightarrow \tau' \rightarrow \tau
\]

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

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

\[
C \vdash ? \Rightarrow \text{exp} \Rightarrow \tau
\]  \hspace{1cm} (16) \hspace{1cm} (17)

**Declarations**

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

\[
C \vdash \text{val} \text{valbind} \Rightarrow \text{Clos}CVE \text{ in Env}
\]  \hspace{1cm} (18)
\[
\frac{C \vdash \text{typbind} \Rightarrow TE}{C \vdash \text{type typbind} \Rightarrow TE \text{ in Env}} \quad (19)
\]
\[
\frac{C \otimes TE \vdash \text{datbind} \Rightarrow VE, TE \quad \forall (t, CE) \in \text{Ran} TE, \; t \notin (T \text{ of } C)}{C \vdash \text{datatype datbind} \Rightarrow (VE, TE) \text{ in Env}} \quad (20)
\]
\[
\frac{C \otimes TE \vdash \text{datbind} \Rightarrow VE, TE \quad \forall (t, CE) \in \text{Ran} TE, \; t \notin (T \text{ of } C)}{C \otimes (VE, TE) \vdash \text{dec} \Rightarrow E}
\]
\[
C \vdash \text{abstype datbind with dec end} \Rightarrow E + \text{Abs } TE \quad (21)
\]
\[
C \vdash \text{exnbind} \Rightarrow EE \quad (22)
\]
\[
\frac{C \vdash \text{exception exnbind} \Rightarrow EE \text{ in Env}}{C \vdash \text{local dec}_1 \text{ in dec}_2 \text{ end} \Rightarrow E_2} \quad (23)
\]
\[
C(\text{longstrid}_1) = (m_1, E_1) \quad \cdots \quad C(\text{longstrid}_n) = (m_n, E_n)
\]
\[
C \vdash \text{open longstrid}_1 \cdots \text{longstrid}_n \Rightarrow E_1 + \cdots + E_n \quad (24)
\]
\[
\frac{C \vdash \{\} \Rightarrow \{\} \text{ in Env}}{C \vdash \text{dec}_1 \Rightarrow E_1 \quad C \otimes E_1 \vdash \text{dec}_2 \Rightarrow E_2}
\]
\[
C \vdash \text{dec}_1 \langle \_ \rangle \text{ dec}_2 \Rightarrow E_1 + E_2 \quad (26)
\]

Comments:

(18) Here \(VE\) will contain types rather than general type schemes. The closure of \(VE\) is exactly what allows variables to be used polymorphically, via rule 1.

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

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

\[
C \vdash valbind \Rightarrow VE
\]

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

(27)

\[
C + VE \vdash valbind \Rightarrow VE
\]

(28)

Comments:

(27) When the option is present we have DomVE \cap DomVE' = \emptyset by the syntactic restrictions.

(28) Modifying C by VE on the left captures the recursive nature of the binding. From rule 27 we see that any type scheme occurring in VE will have to be a type. Thus each use of a recursive function in its own body must be ascribed the same type.

Type Bindings

\[
C \vdash typbind \Rightarrow TE
\]

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

(29)

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

Data Type Bindings

\[
C \vdash datbind \Rightarrow VE, TE
\]

\[
\frac{\text{tyvarseq} = \alpha^{(k)} \quad C, \alpha^{(k)}t \vdash constrs \Rightarrow CE \quad (C \vdash datbind \Rightarrow VE, TE)} {C \vdash tyvarseq \; tycon = constrs \langle \text{and datbind} \rangle \Rightarrow \text{ClosCE} \langle + \; VE \rangle, \{tycon \mapsto (t, \text{ClosCE})\} \langle + \; TE \rangle}
\]

(30)

Comment: The syntactic restrictions ensure DomVE \land DomCE = \emptyset and \(tycon \notin \text{Dom}TE\).

Constructor Bindings

\[
C, \tau \vdash constrs \Rightarrow CE
\]

\[
\frac{\langle C \vdash ty \Rightarrow \tau' \rangle \quad \langle (C, \tau \vdash constrs \Rightarrow CE) \rangle} {\langle C, \tau \vdash \text{con} \langle \text{of } ty \rangle \langle \{ \{\text{con} \mapsto \tau\} \rangle \langle + \; CE \rangle \rangle}
\]

(31)

Comment: By the syntactic restrictions con \(\notin\) DomCE.
Exception Bindings

\[ C \vdash \text{exnbind} \Rightarrow EE \]

\[
\frac{(C \vdash ty \Rightarrow \tau) \quad \langle \langle C \vdash \text{exnbind} \Rightarrow EE \rangle \rangle}{\langle C \vdash \text{exn} \langle : ty \rangle \langle \langle \langle C \vdash \text{exnbind} \Rightarrow EE \rangle \rangle \Rightarrow \{\text{exn} \mapsto \text{unit}\} \langle + \{\text{exn} \mapsto \tau\} \rangle \langle \langle + EE \rangle \rangle}
\]

\[
\frac{C(\text{longexn}) = \tau \quad (C \vdash ty \Rightarrow \tau) \quad \langle \langle C \vdash \text{exnbind} \Rightarrow EE \rangle \rangle}{\langle C \vdash \text{exn} \langle : ty \rangle = \text{longexn} \langle \langle \langle \text{and} \text{exnbind} \rangle \rangle \Rightarrow \{\text{exn} \mapsto \tau\} \langle \langle + EE \rangle \rangle}
\]

Comments:

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

Atomic Patterns

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

\[
\frac{C \vdash - \Rightarrow (\{\}, \tau)}{(34)}
\]

\[
\frac{C \vdash \text{var} \Rightarrow (\{\text{var} \mapsto \tau\}, \tau)}{(35)}
\]

\[
\frac{C(\text{longcon}) = \tau \quad (C \vdash \text{longcon} \Rightarrow (\{\}, \tau))}{(C \vdash \text{patrow} \Rightarrow (VE, \varnothing))}
\]

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

Comments:

(35) Note that \( \text{var} \) can assume a type, not a general type scheme.

Pattern Rows

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

\[
\frac{C \vdash \ldots \Rightarrow (\{\}, \varnothing)}{(39)}
\]

\[
\frac{C \vdash \text{pat} \Rightarrow (VE, \tau) \quad \langle C \vdash \text{patrow} \Rightarrow (VE', \varnothing) \quad \text{lab} \notin \text{Dom} \varnothing \rangle}{C \vdash \text{lab} = \text{pat} \langle , \text{patrow}\rangle \Rightarrow (VE \langle + VE'\rangle, \{\text{lab} \mapsto \tau\} \langle + \varnothing\rangle)}
\]

Comment:

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

31
Patterns

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

\[ C \vdash \text{atpat} \Rightarrow (VE, \tau) \]
\[ C \vdash \text{atpat} \Rightarrow (VE, \tau') \]
\[ C(\text{longcon}) \Rightarrow \tau' \Rightarrow \tau \quad C \vdash \text{atpat} \Rightarrow (VE, \tau') \]
\[ C \vdash \text{longcon atpat} \Rightarrow (VE, \tau) \]
\[ C \vdash \text{ty} \Rightarrow \tau \]
\[ C \vdash \text{var} \Rightarrow (VE, \tau) \quad (C \vdash \text{ty} \Rightarrow \tau) \]
\[ C \vdash \text{ty} \Rightarrow (VE', \tau) \]
\[ C \vdash \text{var}(\text{ty}) \text{ as pat} \Rightarrow (VE + VE', \tau) \]

Comments:

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

Type Expressions

\[ C \vdash \text{tyvar} \Rightarrow \alpha \]
\[ C \vdash \{\text{tyvar} \Rightarrow \alpha\} \]
\[ \langle C \vdash \text{ty} \Rightarrow \theta \rangle \]
\[ \{\text{ty}\} \Rightarrow \{\theta\} \text{ in Type} \]
\[ \text{tyseq} = \text{ty}_{1} \cdots \text{ty}_{k} \quad C \vdash \text{ty}_{i} \Rightarrow \tau_{i} \ (1 \leq i \leq k) \]
\[ C(\text{longtycon}) = (\theta, CE) \]
\[ C \vdash \text{tyseq longtycon} \Rightarrow \tau^{(k)}\theta \]
\[ C \vdash \text{ty} \Rightarrow \tau \quad C \vdash \text{ty}' \Rightarrow \tau' \]
\[ C \vdash \text{ty} \Rightarrow \tau \quad C \vdash \text{ty}' \Rightarrow \tau' \]
\[ C \vdash (\text{ty}) \Rightarrow \tau \]

Comments:

(47) Recall that for \( \tau^{(k)}\theta \) to be defined, \( \theta \) must have arity \( k \).
Type-expression Rows

\[ C \vdash ty \Rightarrow \tau \quad \langle C \vdash tyrow \Rightarrow \rho \rangle \]

\[ C \vdash \text{lab : ty} \langle , tyrow \rangle \Rightarrow \{ \text{lab} \mapsto \tau \} \langle + \rho \rangle \]

(50)

Comment: The syntactic constraints ensure \( \text{lab} \notin \text{Dom} \rho \).

4.9 Further Restrictions

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

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

2. For each occurrence of a record pattern containing a record wildcard, i.e. of the form \( \{ \text{lab}_1=\text{pat}_1, \cdots, \text{lab}_m=\text{pat}_m, \cdots \} \), the program context must determine uniquely the domain \( \{ \text{lab}_1, \cdots, \text{lab}_n \} \) of its record type, where \( m \leq n \); thus, the context must determine the labels \( \{ \text{lab}_{m+1}, \cdots, \text{lab}_n \} \) of the fields to be matched by the wildcard. For this purpose, an explicit type constraint may be needed. This restriction is necessary to ensure the existence of principal type schemes.

3. In a match, the sequence of patterns \( \text{pat}_1, \cdots, \text{pat}_n \) must be irredundant and exhaustive. That is, each \( \text{pat}_j \) must match some value (of the right type) which is not matched by \( \text{pat}_i \) for any \( i < j \), and every value (of the right type) must be matched by some \( \text{pat}_i \). The compiler must give a warning on violation of this restriction, but should still compile the match.

4.10 Principal Environments

Let \( C \) be a context, and suppose that \( C \vdash \text{dec} \Rightarrow E \) according to the preceding Inference Rules. Then \( E \) is principal (for \( \text{dec} \) in the context \( C \)) if, for all \( E' \) for which \( C \vdash \text{dec} \Rightarrow E' \), we have \( E \gg E' \). We claim that if \( \text{dec} \) elaborates to any environment in \( C \) then it elaborates to a principal environment in \( C \). Strictly, we must allow for the possibility that type names which do not occur in \( C \), are chosen differently for \( E \) and \( E' \); the stated claim is therefore made up to such variation.
5 Static Semantics for Modules

5.1 Semantic Objects

The simple objects for Modules static semantics are exactly as for the Core. The compound objects are those for the Core, augmented by those in Figure 14.

\[
\begin{align*}
M & \in \text{StrNameSet} = \text{Fin(StrName)} \\
N \text{ or } (M,T) & \in \text{NameSet} = \text{StrNameSet} \times \text{TyNameSet} \\
\Sigma \text{ or } (N)S & \in \text{Sig} = \text{NameSet} \times \text{Str} \\
\Phi \text{ or } (N)(S, (N')S') & \in \text{FunSig} = \text{NameSet} \times (\text{Str} \times \text{Sig}) \\
G & \in \text{SigEnv} = \text{SigId} \overset{\text{fn}}{\rightarrow} \text{Sig} \\
F & \in \text{FunEnv} = \text{FunId} \overset{\text{fn}}{\rightarrow} \text{FunSig} \\
B \text{ or } N,F,G,E & \in \text{Basis} = \text{NameSet} \times \text{FunEnv} \times \text{SigEnv} \times \text{Env}
\end{align*}
\]

Figure 14: Further Compound Semantic Objects

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

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

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

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

The operations of projection, injection and modification are as for the Core. Also, we frequently need to modify a basis \( B \) by an environment \( E \) (or a structure environment \( SE \) say), at the same time extending \( N \) of \( B \) to include the type names and structure names of \( E \) (or of \( SE \) say). We therefore define \( B \oplus SE \), for example, to mean \( B + (\text{names } SE, \text{SE}) \).
5.2 Consistency

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

\[ CE_1 = \{\} \text{ or } CE_2 = \{\} \text{ or } CE_1 = CE_2 \]

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

1. If \(m\) of \(S_1 = m\) of \(S_2\), and both \(S_1(longstrid)\) and \(S_2(longstrid)\) exist, then
   \[ m of S_1(longstrid) = m of S_2(longstrid) \]

2. If \(m\) of \(S_1 = m\) of \(S_2\), and both \(S_1(longtycon)\) and \(S_2(longtycon)\) exist, then
   \[ \theta of S_1(longtycon) = \theta of S_2(longtycon) \]

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

As an example, a functor signature \((N)(S,(N')(S'))\) is consistent if, assuming first that \(N \cap N' = \emptyset\), \(A = \{S_1, S_2\}\) is consistent.

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

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

5.3 Well-formedness

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

A signature \((N)S\) is well-formed if, whenever \((m,E)\) is a substructure of \(S\) and \(m \notin N\), then \(N \cap \text{names}(E) = \emptyset\). A functor signature \((N)(S,(N')(S'))\) is well-formed if \((N)S\), \((N')(S')\) and \((N \cup N')(S')\) are well-formed, and \(\text{names}(N')S' \cap N \subseteq \text{names} S\) (the latter condition is satisfied automatically for user-defined functors).

An object or assembly \(A\) is well-formed if every type environment, signature and functor signature occurring in \(A\) is well-formed.
5.4 Cycle-freedom

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

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

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

5.5 Admissibility

An object or assembly $A$ is admissible if it is consistent, well-formed and cycle-free. Henceforth it is assumed that all objects mentioned are admissible; in particular, the admissibility of each semantic object mentioned is taken as a condition throughout the semantic rules which follow. (In our semantic description we have not undertaken to indicate how admissibility should be checked in an implementation.)

5.6 Type Realisation

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

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

5.7 Realisation

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

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

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

5.8 Signature Instantiation

A structure $S_2$ is an instance of a signature $\Sigma_1 = (N_1)S_1$, written $\Sigma_1 \geq S_2$, if there exists a realisation $\varphi$ such that $\varphi(S_1) = S_2$ and $\text{Supp} \varphi \subseteq N_1$. A signature
\( \Sigma_2 = (N_2)S_2 \) is an instance of \( \Sigma_1 = (N_1)S_1 \), written \( \Sigma_1 \geq \Sigma_2 \), if \( \Sigma_1 \geq S_2 \) and \( N_2 \cap \text{name}(\Sigma_1) = \emptyset \). We claim that \( \Sigma_1 \geq \Sigma_2 \) iff, for all \( S \), whenever \( \Sigma_2 \geq S \) then \( \Sigma_1 \geq S \).

5.9 Functor Signature Instantiation

A pair \( (S, (N')S') \) is called a functor instance. Given \( \Phi = (N_1)(S_1, (N'_1)S'_1) \), a functor instance \( (S_2, (N'_2)S'_2) \) is an instance of \( \Phi \), written \( \Phi \geq (S_2, (N'_2)S'_2) \), if there exists a realisation \( \varphi \) such that \( \varphi(S_1, (N'_1)S'_1) = (S_2, (N'_2)S'_2) \) and \( \text{Supp} \varphi \subseteq N_1 \).

5.10 Enrichment

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

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

1. \( m_1 = m_2 \)
2. \( E_1 \triangleright E_2 \)

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

1. \( \text{Dom} SE_1 \supseteq \text{Dom} SE_2 \), and \( SE_1(\text{strid}) \triangleright SE_2(\text{strid}) \) for all \( \text{strid} \in \text{Dom} SE_2 \)
2. \( \text{Dom} TE_1 \supseteq \text{Dom} TE_2 \), and \( TE_1(\text{tycon}) \triangleright TE_2(\text{tycon}) \) for all \( \text{tycon} \in \text{Dom} TE_2 \)
3. \( \text{Dom} VE_1 \supseteq \text{Dom} VE_2 \), and \( VE_1(\text{var}) \triangleright VE_2(\text{var}) \) for all \( \text{var} \in \text{Dom} VE_2 \)
4. \( \text{Dom} EE_1 \supseteq \text{Dom} EE_2 \), and \( EE_1(\text{exn}) = EE_2(\text{exn}) \) for all \( \text{exn} \in \text{Dom} EE_2 \)

Finally, a type structure \( (\theta_1, CE_1) \) enriches another type structure \( (\theta_2, CE_2) \), written \( (\theta_1, CE_1) \triangleright (\theta_2, CE_2) \), if

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

5.11 Principal Signatures

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

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

\[ A \vdash \text{phrase} \Rightarrow A' \]

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

Although not assumed in our definitions, it is intended that every basis \( B = N, F, G, E \) in which a program is elaborated has the property that names \( F \cup \) names \( G \cup \text{names} E \subseteq N \). This is not the case for bases in which signature expressions and specifications are elaborated, but the following Theorem can be proved:

Let \( S \) be an inferred sentence \( B \vdash \text{program} \Rightarrow B' \) in which \( B \) satisfies the above condition. Then \( B' \) also satisfies the condition.

Moreover, if \( S' \) is a sentence of the form \( B'' \vdash \text{phrase} \Rightarrow A \) occurring in a proof of \( S \), where \( \text{phrase} \) is either a structure expression or a structure declaration, then \( B'' \) also satisfies the condition.

Finally, if \( T, E \vdash \text{phrase} \Rightarrow A \) occurs in a proof of \( S \), where \( \text{phrase} \) is a phrase of the Core, then tnames \( E \subseteq T \).

Structure Expressions

\[
\begin{align*}
B \vdash \text{strdec} & \Rightarrow E \quad m \notin (N \text{ of } B) \cup \text{names } E \\
B & \vdash \text{struct strdec end} \Rightarrow (m, E) \\
B(\text{longstrid}) & = S \\
B & \vdash \text{longstrid} \Rightarrow S \\
B & \vdash \text{strexp} \Rightarrow S \\
B(\text{funid}) & \supseteq (S'', (N'')S') , S \supsetneq S'' \\
(N \text{ of } B) \cap N' &= \emptyset \\
B & \vdash \text{funid ( strexp )} \Rightarrow S' \\
B \vdash \text{strdec} & \Rightarrow E \quad B \oplus E \vdash \text{strexp} \Rightarrow S \\
B & \vdash \text{let strdec in strexp end} \Rightarrow S
\end{align*}
\]

Comments:

(51) The side condition ensures that each generative structure expression receives a new name. If the expression occurs in a functor body the structure name
will be bound by \((N')\) in rule 95; this will ensure that for each application of the functor, by rule 53, a new distinct name will be chosen for the structure generated.

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

The realisation \(\varphi\) for which \(\varphi(B(funid)) = (S'', (N')S')\) is uniquely determined by \(B(funid)\) and \(S\), since \(S > S''\) can only hold if the type names and structure names in \(S\) and \(S''\) agree. Recall that enrichment \(>\) allows more components and more polymorphism, while instantiation \(\geq\) does not.

Sharing specified in the declaration of the functor \(funid\) is represented by the occurrence of the same name in both components of \(B(funid)\), and this repeated occurrence is preserved by \(\varphi\), yielding sharing between the argument structure \(S\) and the result structure \(S'\) of this functor application.

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

**Structure-level Declarations**

\[
\begin{align*}
B \vdash strdec \Rightarrow E & \\
\text{C of } B \vdash dec \Rightarrow E & \quad \text{E principal in (C of B)} \\
B \vdash dec \Rightarrow E & \\
\text{B \vdash strbind \Rightarrow SE} & \\
B \vdash \text{structure strbind} \Rightarrow SE \text{ in Env} & \\
B \vdash strdec_1 \Rightarrow E_1 & \quad B \oplus E_1 \vdash strdec_2 \Rightarrow E_2 \\
B \vdash \text{local strdec}_1 \text{ in strdec}_2 \text{ end} \Rightarrow E_2 & \\
\text{B \vdash } & \\
\text{\{\} in Env} & \\
B \vdash strdec_1 \Rightarrow E_1 & \quad B \oplus E_1 \vdash strdec_2 \Rightarrow E_2 \\
B \vdash strdec_1 (:) \strdec_2 \Rightarrow E_1 + E_2 & \\
\end{align*}
\]

**Comments:**

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

\[ B \vdash \text{strbind} \Rightarrow SE \]

\[
\frac{B \vdash \text{strexp} \Rightarrow S \quad \langle B \vdash \text{sigexp} \Rightarrow S', \ S \triangleright S' \rangle}{\langle B + \text{names} S \vdash \text{strbind} \Rightarrow SE \rangle}
\]

\[
\frac{B \vdash \text{strid} (\cdot : \text{sigexp}) = \text{strexp} (\langle \text{and} \ \text{strbind} \rangle) \Rightarrow \{ \text{strid} \mapsto S' \langle \cdot \rangle \} \langle + SE \rangle}{(60)}
\]

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

Signature Expressions

\[ B \vdash \text{sigexp} \Rightarrow S \]

\[
\frac{B \vdash \text{spec} \Rightarrow E}{B \vdash \text{sig spec end} \Rightarrow (m, E)}
\]

\[
\frac{B(m) \triangleright S}{B \vdash \text{sigid} \Rightarrow S}
\]

Comments:

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

(62) The instance \text{S} of \text{B}\langle\text{sigid}\rangle\text{ is not determined by this rule, but as in rule 61--the instance may be chosen to achieve sharing properties or enrichment conditions.}

Signature Declarations

\[ B \vdash \text{sigdec} \Rightarrow G \]

\[
\frac{B \vdash \text{sigbind} \Rightarrow G}{B \vdash \text{signature} \ \text{sigbind} \Rightarrow G}
\]

\[
\frac{B \vdash \{}{\}
\]

\[
\frac{B \vdash \text{sigdec}_1 \Rightarrow G_1 \quad B + G_1 \vdash \text{sigdec}_2 \Rightarrow G_2}{B \vdash \text{sigdec}_1 \langle ; \rangle \ \text{sigdec}_2 \Rightarrow G_1 + G_2}
\]

Comments:

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

\[ B \vdash sigbind \Rightarrow G \]

(66)

\[ B \vdash sigexp \Rightarrow S \quad (N)S \text{ principal in } B \quad (B \vdash sigbind \Rightarrow G) \quad B \vdash sigid = sigexp \text{ (and } sigbind) \Rightarrow \{sigid \mapsto (N)S\} \quad (\vdash G) \]

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

Specifications

\[ B \vdash spec \Rightarrow E \]

(67)

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

\[ B \vdash val valdesc \Rightarrow \text{ ClosVE in Env} \]

(68)

\[ C \text{ of } B \vdash typdesc \Rightarrow TE \]

\[ B \vdash type typdesc \Rightarrow TE \text{ in Env} \]

\[ C \text{ of } B \vdash typdesc \Rightarrow TE \quad \forall (\theta, CE) \in \text{ RanTE}, \theta \text{ admits equality} \]

\[ B \vdash eqtype typdesc \Rightarrow TE \text{ in Env} \]

(69)

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

\[ B \vdash datatype datdesc \Rightarrow (VE, TE) \text{ in Env} \]

(70)

\[ C \text{ of } B \vdash exndesc \Rightarrow EE \]

\[ B \vdash exception exndesc \Rightarrow EE \text{ in Env} \]

(71)

\[ B \vdash strdesc \Rightarrow SE \]

\[ B \vdash structure strdesc \Rightarrow SE \text{ in Env} \]

(72)

\[ B \vdash shareq \Rightarrow \{\} \]

\[ B \vdash sharing shareq \Rightarrow \{\} \text{ in Env} \]

(73)

\[ B \vdash spec_1 \Rightarrow E_1 \quad B + E_1 \vdash spec_1 \Rightarrow E_2 \]

\[ B \vdash local spec_1 \text{ in spec}_2 \text{ end } \Rightarrow E_2 \]

(74)

\[ B(longstrid_1) = (m_1, E_1) \quad \ldots \quad B(longstrid_n) = (m_n, E_n) \]

\[ B \vdash open longstrid_1 \cdots longstrid_n \Rightarrow E_1 + \cdots + E_n \]

(75)

\[ B(sigid_1) \geq (m_1, E_1) \quad \ldots \quad B(sigid_n) \geq (m_n, E_n) \]

\[ B \vdash include sigid_1 \cdots sigid_n \Rightarrow E_1 + \cdots + E_n \]

(76)

\[ B \vdash \Rightarrow \{\} \text{ in Env} \]

(77)
\[
\frac{B \vdash \text{spec}_1 \Rightarrow E_1 \quad B \vdash \text{spec}_1 \Rightarrow E_2}{B \vdash \text{spec}_1 \langle ; \rangle \text{spec}_2 \Rightarrow E_1 + E_2}
\] (78)

Comments:

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

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

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

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

Value Descriptions

\[
\frac{C \vdash valdesc \Rightarrow VE}{C \vdash \text{ty} \Rightarrow \tau \quad (C \vdash valdesc \Rightarrow VE)}
\frac{C \vdash \text{var : ty (and valdesc) } \Rightarrow \{ \text{var } \mapsto \tau \} \langle + VE \rangle}{C \vdash \text{var : ty (and valdesc) } \Rightarrow \{ \text{var } \mapsto \tau \} \langle + VE \rangle}
\] (79)

Type Descriptions

\[
\frac{\text{tyvarseq } = \alpha^{(k)} \quad (C \vdash \text{typdesc } \Rightarrow TE)}{C \vdash \text{tyvarseq tycon (and typdesc) } \Rightarrow \{\text{tycon } \mapsto (\theta, \{\})\} \langle + TE \rangle}
\] (80)

Comment: Note that any \(\theta\) of arity \(k\) may be chosen.

Datatype Descriptions

\[
\frac{\text{tyvarseq } = \alpha^{(k)} \quad C, \alpha^{(k)}t \vdash \text{constrs } \Rightarrow CE \quad (C \vdash \text{datdesc } \Rightarrow VE, TE)}{C \vdash \text{tyvarseq tycon } = \text{constrs (and datdesc) } \Rightarrow ClosCE \langle + VE \rangle, \{\text{tycon } \mapsto (\text{t, ClosCE})\} \langle + TE \rangle}
\] (81)

Exception Descriptions

\[
\frac{C \vdash \text{ty} \Rightarrow \tau \quad \text{tyvars}(\tau) = \emptyset \quad (C \vdash \text{exndesc } \Rightarrow EE)}{C \vdash \text{exn : ty (and exndesc) } \Rightarrow \{\text{exn } \mapsto \tau\} \langle + EE \rangle}
\] (82)
Structure Descriptions

\[ B ⊢ \text{strdesc} \Rightarrow SE \]

\[ B ⊢ \text{sigexp} \Rightarrow S \quad (B ⊢ \text{strdesc} \Rightarrow SE) \]
\[
\begin{align*}
B ⊢ \text{strid} : \text{sigexp} \text{ (and strdesc)} & \Rightarrow \{ \text{strid} \mapsto S \} \quad (+ SE)
\end{align*}
\] (83)

Sharing Equations

\[ B ⊢ \text{shareq} \Rightarrow \{\} \]

\[ m \text{ of } (B(\text{longstrid}_1)) = \cdots = m \text{ of } (B(\text{longstrid}_n)) \quad (84) \]
\[
\begin{align*}
B ⊢ \text{longstrid}_1 = \cdots = \text{longstrid}_n & \Rightarrow \{\} \\
\end{align*}
\]

\[ B ⊢ \text{longtycon}_1 = \cdots = B(\text{longtycon}_n) \quad (85) \]
\[
\begin{align*}
B ⊢ \text{type longtycon}_1 = \cdots = \text{longtycon}_n & \Rightarrow \{\} \\
\end{align*}
\]

\[ B ⊢ \text{shareq}_1 \Rightarrow \{\} \quad B ⊢ \text{shareq}_2 \Rightarrow \{\} \]
\[
\begin{align*}
B ⊢ \text{shareq}_1 \text{ and shareq}_2 & \Rightarrow \{\}
\end{align*}
\] (86)

Comments:

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

\[ B(\text{longstrid}_1) = \cdots = B(\text{longstrid}_n) \]

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

Functor Specifications

\[ B ⊢ \text{funspec} \Rightarrow F \]

\[ B ⊢ \text{fundesc} \Rightarrow F \quad (87) \]
\[
\begin{align*}
B ⊢ \text{functor fundesc} & \Rightarrow F \\
\end{align*}
\]

\[ B ⊢ \Rightarrow \{\} \quad (88) \]

\[ B ⊢ \text{funspec}_1 \Rightarrow F_1 \quad B + F_1 ⊢ \text{funspec}_2 \Rightarrow F_2 
\]
\[
\begin{align*}
B ⊢ \text{funspec}_1 \text{ (;}\text{funspec}_2 \Rightarrow F_1 + F_2 \\
\end{align*}
\] (89)

Functor Descriptions

\[ B ⊢ \text{fundesc} \Rightarrow F \]

\[ B ⊢ \text{funsigexp} \Rightarrow \Phi \quad (B ⊢ \text{fundesc} \Rightarrow F) \]
\[
\begin{align*}
B ⊢ \text{funid funsigexp} \text{ (and fundesc)} & \Rightarrow \{ \text{funid} \mapsto \Phi \}(+ F)
\end{align*}
\] (90)
Functor Signature Expressions

\[ B \vdash \text{funsigexp} \Rightarrow \Phi \]

\[ B \vdash \text{sigexp} \Rightarrow S \quad (N)S \text{ principal in } B \]
\[ B \oplus \{ \text{strid} \mapsto S\} \vdash \text{sigexp'} \Rightarrow S' \]
\[ N' = \text{names } S' \setminus ((N \text{ of } B) \cup N) \]
\[ B \vdash (\text{strid} : \text{sigexp}) : \text{sigexp'} \Rightarrow (N)(S,(N')S') \quad (91) \]

Functor Declarations

\[ B \vdash \text{fundec} \Rightarrow F \]

\[ B \vdash \text{funbind} \Rightarrow F \]
\[ B \vdash \text{functor } \text{funbind} \Rightarrow F \quad (92) \]

\[ B \vdash - \Rightarrow \{\} \]

\[ B \vdash \text{fundec}_1 \Rightarrow F_1 \quad B + F_1 \vdash \text{fundec}_2 \Rightarrow F_2 \]
\[ B \vdash \text{fundec}_1 (;) \text{fundec}_2 \Rightarrow F_1 + F_2 \quad (94) \]

Functor Bindings

\[ B \vdash \text{sigexp} \Rightarrow S \quad (N)S \text{ principal in } B \]
\[ B \oplus \{ \text{strid} \mapsto S\} \vdash \text{strexp} \Rightarrow S' \]
\[ \langle B \oplus \{ \text{strid} \mapsto S\} \vdash \text{sigexp'} \Rightarrow S'', S' \succ S'' \rangle \]
\[ N' = \text{names } S' \setminus ((N \text{ of } B) \cup N) \]
\[ \langle \{ B \vdash \text{funbind} \Rightarrow F\} \rangle \]
\[ B \vdash \text{funid}(\text{strid} : \text{sigexp}) \langle : \text{sigexp'} \rangle = \text{strexp}(\langle \text{and funbind} \rangle) \Rightarrow \{\text{funid} \mapsto (N)(S,(N')S'')\} \langle + F\} \quad (95) \]

Comment: Here \((N)S\) is required to be principal so as to be as general as possible given the sharing constraints in \text{sigexp}. Since \(\oplus\) is used, any structure name \(m\) and type name \(t\) in \(S\) acts like a constant in the functor body; in particular, it ensures that further names generated during elaboration of the body are distinct from \(m\) and \(t\). The set \(N'\) is chosen such that every name free in \((N)S\) or \((N)(S,(N')S')\) is free in \(B\).

Programs

\[ B \vdash \text{program} \Rightarrow B' \]

\[ B \vdash \text{strdec} \Rightarrow E \]
\[ B \vdash \text{strdec} \Rightarrow (\text{names }E,E) \text{ in Basis} \quad (96) \]
\[
B \vdash \text{sigdec} \Rightarrow G \\
B \vdash \text{sigdec} \Rightarrow (\text{names } G, G) \text{ in Basis}
\]

\[
B \vdash \text{fundec} \Rightarrow F \\
B \vdash \text{fundec} \Rightarrow (\text{names } F, F) \text{ in Basis}
\]

\[
B \vdash \text{program}_1 \Rightarrow B_1 \\
B + B_1 \vdash \text{program}_2 \Rightarrow B_2 \\
B \vdash \text{program}_1 \langle ; \rangle \text{ program}_2 \Rightarrow B_1 + B_2
\]

5.13 Functor Matching

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

\[
\Phi_1 = (N_1)(S_1, (N'_1)S'_1)
\]

and that later \( f \) is declared with functor signature

\[
\Phi_2 = (N_2)(S_2, (N'_2)S'_2)
\]

the following matching rule will be employed:

A functor signature \( \Phi_2 = (N_2)(S_2, (N'_2)S'_2) \) matches another functor signature, \( \Phi_1 = (N_1)(S_1, (N'_1)S'_1) \), if

1. There is a realisation \( \varphi \), \( \text{Supp} \varphi \subseteq N_2 \), such that \( \varphi S_2 \prec S_1 \)

2. Assuming that \( \text{Yield} \varphi \cap N'_2 = \emptyset \), there is a realisation \( \varphi' \), \( \text{Supp} \varphi' \subseteq N'_1 \), such that \( \varphi' S'_1 \prec \varphi S'_2 \)

The first condition ensures that the real functor signature \( \Phi_2 \) for \( f \) requires the argument \text{strexp} of any application \( f(\text{strexp}) \) to have no more sharing, and no more richness, than was predicted by the specified signature \( \Phi_1 \). The second condition ensures that the real functor signature \( \Phi_2 \), instantiated to \( (\varphi S_2, (N'_2)\varphi S'_2) \), provides in the result of the application \( f(\text{strexp}) \) no less sharing, and no less richness, than was predicted by the specified signature \( \Phi_1 \).
6 Dynamic Semantics for the Core

6.1 Reduced Syntax

Since types are fully dealt with in the static semantics, the dynamic semantics ignores them. The Core syntax is therefore reduced by the following transformations, for the purpose of the dynamic semantics:

- All explicit type ascriptions ":: ty" are omitted.
- Any declaration of the form "type typbind" or "datatype datbind" is replaced by the empty declaration.
- A declaration of the form "abstype datbind with dec end" is replaced by "dec".
- The Core phrase classes typbind, datbind, constrs, ty and tyrow are omitted.

6.2 Simple Objects

All objects in the dynamic semantics are built from identifier classes together with the simple object classes shown (with the variables which range over them) in Figure 15.

\[
\begin{align*}
a & \in \text{Addr} \quad \text{addresses} \\
e & \in \text{Exc} \quad \text{exceptions} \\
b & \in \text{BasVal} \quad \text{basic values} \\
\{\text{FAIL}\} & \text{failure}
\end{align*}
\]

Figure 15: Simple Semantic Objects

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

6.3 Compound Objects

The compound objects for the dynamic semantics are shown in Figure 16. Many conventions and notations are adopted as in the static semantics; in particular
\[ v \in \text{Val} = \{:=\} \cup \text{BasVal} \cup \text{Con} \cup (\text{Con} \times \text{Val}) \cup \\
\text{Record} \cup \text{Addr} \cup \text{Closure} \]

\[ r \in \text{Record} = \text{Lab} \Rightarrow \text{Val} \]

\[ |e,v| \text{ or } p \in \text{Pack} = \text{Exc} \times \text{Val} \]

\[ \text{(match, } E, VE) \in \text{Closure} = \text{Match} \times \text{Env} \times \text{VarEnv} \]

\[ \text{mem} \in \text{Mem} = \text{Addr} \Rightarrow \text{Val} \]

\[ \text{excs} \in \text{ExcSet} = \text{Fin(Exc)} \]

\[ (\text{mem}, \text{excs}) \text{ or } s \in \text{State} = \text{Mem} \times \text{ExcSet} \]

\[ \text{(SE, } VE, EE) \text{ or } E \in \text{Env} = \text{StrEnv} \times \text{VarEnv} \times \text{ExnEnv} \]

\[ SE \in \text{StrEnv} = \text{StrId} \Rightarrow \text{Env} \]

\[ VE \in \text{VarEnv} = \text{Var} \Rightarrow \text{Val} \]

\[ EE \in \text{ExnEnv} = \text{Exn} \Rightarrow \text{Exc} \]

Figure 16: Compound Semantic Objects

projection, injection and modification all retain their meaning. We generally omit
the injection functions taking Con, Con × Val etc into Val. For records \( r \in \text{Record} \)
however, we write this injection explicitly as "in Val"; this accords with the fact
that there is a separate phrase class ExpRow, whose members evaluate to records.
We take \( \cup \) to mean disjoint union over semantic object classes.

Although the same names, e.g. \( E \) for an environment, are used as in the
static semantics, the objects denoted are different. This need cause no confusion
since the static and dynamic semantics are presented completely separately. An
important point is that structure names \( m \) have no significance at all in the
dynamic semantics; this explains why the object class Str = StrName × Env is
absent here – for the dynamic semantics the concepts structure and environment
coincide.

6.4 Basic Values

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

abs floor real sqrt sin cos arctan exp ln
size chr ord explode implode div mod
- \ \\ * + - = <> < > <= >=

std_in std_out open_in open_out close_in close_out

47
The meaning of basic values (almost all of which are functions) is represented by the function

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

which is detailed in Appendix D.

### 6.5 Basic Exceptions

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

\[
\text{ord chr div mod} / \ast \ast \text{floor} \ \text{sqrt exp ln}
\text{iocfailure match bind interrupt}
\]

The exceptions on the first line are raised by basic functions of the same name, and \text{iofailure} by certain of the basic input/output functions, as detailed in Appendix D. The exceptions \text{match} and \text{bind} are raised upon failure of pattern-matching in evaluating a \text{match} or a \text{valbind}, as detailed in the rules to follow. Finally, \text{interrupt} is raised by external intervention.

In a match of the form \( \text{pat}_1 \Rightarrow \text{exp}_1 \mid \ldots \mid \text{pat}_n \Rightarrow \text{exp}_n \) the pattern sequence \( \text{pat}_1, \ldots, \text{pat}_n \) should be \text{irredundant} and \text{exhaustive}. That is, each \( \text{pat}_j \) must match some value (of the right type) which is not matched by \( \text{pat}_i \) for any \( i < j \), and every value (of the right type) must be matched by some \( \text{pat}_i \). The compiler must give warning on violation of this restriction, but should still compile the match. Thus the \text{match} exception will only be raised for a match which has been flagged by the compiler. The restriction is inherited by derived forms; in particular, this means that in the function binding \( \text{var} \ \text{atpat}_1 \ldots \ \text{atpat}_n (: ty) \) (consisting of one clause only), each separate \( \text{atpat}_i \) should be exhaustive by itself.

For each value binding \( \text{pat} = \text{exp} \) the compiler must issue a report (but still compile) if \text{either} \( \text{pat} \) is not exhaustive or \( \text{pat} \) contains no variable. This will (on both counts) detect a mistaken declaration like \( \text{val} \ \text{nil} = \text{exp} \) 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, these warnings should not be given when the binding is a component of a top-level declaration \( \text{val} \ \text{valbind} \); e.g. \( \text{val} \ \text{x :: l = exp}_1 \) and \( y = \text{exp}_2 \) is not faulted by the compiler at top level, but may of course generate a \text{bind} exception.
6.6 Closures

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

\[
\text{Rec} : \text{VarEnv} \to \text{VarEnv}
\]

is defined as follows:

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

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

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

6.7 Inference Rules

The semantic rules allow sentences of the form

\[s, A \vdash \text{phrase} \Rightarrow A', s'\]

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

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

\[
A_1 \vdash \text{phrase}_1 \Rightarrow A'_1 \quad A_2 \vdash \text{phrase}_2 \Rightarrow A'_2 \quad \cdots \\
\cdots A_n \vdash \text{phrase}_n \Rightarrow A'_n
\]

\[
A \vdash \text{phrase} \Rightarrow A'
\]

49
then the full form is intended to be

\[
\begin{array}{c}
\begin{array}{c}
s_0, A_1 \vdash \text{phrase}_1 \Rightarrow A'_1, s_1 \\
s_1, A_2 \vdash \text{phrase}_2 \Rightarrow A'_2, s_2 \\
\vdots \\
s_{n-1}, A_n \vdash \text{phrase}_n \Rightarrow A'_n, s_n \\
\hline
s_0, A \vdash \text{phrase} \Rightarrow A', s_n
\end{array}
\end{array}
\]

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

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

\[
\begin{array}{c}
\begin{array}{c}
s_1, A_1 \vdash \text{phrase}_1 \Rightarrow A'_1, s'_1 \\
\vdots \\
s_n, A_n \vdash \text{phrase}_n \Rightarrow A'_n, s'_n \\
\hline
s, A \vdash \text{phrase} \Rightarrow A', s'
\end{array}
\end{array}
\]

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

\[
\begin{array}{c}
\begin{array}{c}
s_1, A_1 \vdash \text{phrase}_1 \Rightarrow A'_1, s'_1 \\
\vdots \\
s_k, A_k \vdash \text{phrase}_k \Rightarrow p', s'
\end{array}
\end{array}
\]

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

A third convention is that we allow compound variables (variables built from the variables in Figure 16 and the symbol “\(/") to range over unions of semantic objects. For instance the compound variable \( v/p \) ranges over \( \text{Val} \cup \text{Pack} \). We also allow \( x/\text{FAIL} \) to range over \( x/\text{FAIL} \) where \( x \) ranges over \( X \); furthermore, we extend environment modification to allow for failure as follows:

\[
\text{VE + FAIL = FAIL.}
\]

**Atomic Expressions**

\[
E \vdash \text{atexp} \Rightarrow v/p
\]

\[
\begin{align*}
E(\text{longvar}) = v \\
E \vdash \text{longvar} \Rightarrow v
\end{align*}
\]  

\( \text{(100)} \)

\(^1\) There is one exception to the exception convention; no extra rule is added for rule 112 which deals with handlers, since a handler is the only means by which propagation of an exception can be arrested.
\[ \text{longcon} = \text{strid}_1 \ldots \text{strid}_k \cdot \text{con} \]
\[ E \vdash \text{longcon} \Rightarrow \text{con} \quad (101) \]
\[ \langle E \vdash \text{exprow} \Rightarrow r \rangle \]
\[ E \vdash \{ \langle \text{exprow} \rangle \} \Rightarrow \{ \langle + r \rangle \} \text{ in Val} \quad (102) \]
\[ E \vdash \text{dec} \Rightarrow E' \quad E + E' \vdash \text{exp} \Rightarrow v \]
\[ E \vdash \text{let dec in exp end} \Rightarrow v \quad (103) \]
\[ E \vdash \text{exp} \Rightarrow v \]
\[ E \vdash \langle \text{exp} \rangle \Rightarrow v \quad (104) \]

Comments:

(101) Constructors denote themselves

Expression Rows

\[ E \vdash \text{exprow} \Rightarrow r/p \]
\[ E \vdash \text{exp} \Rightarrow v \quad \langle E \vdash \text{exprow} \Rightarrow r \rangle \]
\[ E \vdash \text{lab} = \text{exp} \langle \cdot, \text{exprow} \rangle \Rightarrow \{ \text{lab} \mapsto v \} \langle + r \rangle \quad (105) \]

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

Expressions

\[ E \vdash \text{atexp} \Rightarrow v \]
\[ E \vdash \text{atexp} \Rightarrow v \quad (106) \]
\[ E \vdash \text{exp} \Rightarrow \text{con} \quad \text{con} \neq \text{ref} \quad E \vdash \text{atexp} \Rightarrow v \]
\[ E \vdash \text{exp atexp} \Rightarrow (\text{con}, v) \quad (107) \]
\[ s, E \vdash \text{exp} \Rightarrow \text{ref}, s', s', E \vdash \text{atexp} \Rightarrow v, s'' \quad a \notin \text{Dom(mem of } s'') \]
\[ s, E \vdash \text{exp atexp} \Rightarrow a, s'' + \{ a \mapsto v \} \quad (108) \]
\[ s, E \vdash \text{exp} \Rightarrow :, s', s', E \vdash \text{atexp} \Rightarrow \{ 1 \mapsto a, 2 \mapsto v \}, s'' \]
\[ s, E \vdash \text{exp atexp} \Rightarrow \{ \} \text{ in Val}, s'' + \{ a \mapsto v \} \quad (109) \]
\[ E \vdash \text{exp} \Rightarrow b \quad E \vdash \text{atexp} \Rightarrow v \quad \text{APPLY}(b, v) = v' \]
\[ E \vdash \text{exp atexp} \Rightarrow v' \quad (110) \]
\[
E \vdash \text{exp} \Rightarrow (\text{match}, E', \text{VE}) \quad E \vdash \text{atexp} \Rightarrow v
\]
\[
E' + \text{RecVE}, \ v \vdash \text{match} \Rightarrow v'
\]
\[
E \vdash \text{exp atexp} \Rightarrow v'
\]
\[
E \vdash \text{exp} \Rightarrow v
\]
\[
E \vdash \text{exp handle} \ handler \Rightarrow v
\]
\[
E \vdash \text{exp} \Rightarrow p \quad E, p \vdash \text{handler} \Rightarrow v
\]
\[
E \vdash \text{exp handle} \ handler \Rightarrow v
\]
\[
E(\text{longezn}) = e \quad E \vdash \text{exp} \Rightarrow v
\]
\[
E \vdash \text{raise longezn with} \ \text{exp} \Rightarrow [e, v]
\]
\[
E \vdash \text{fn} \ \text{match} \Rightarrow (\text{match}, E, \{\})
\]

Comments:

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

(107)-(111) Note that none of the rules for function application has a premise in which the operator evaluates to a constructed value, a record or an address. This is because we are interested in the evaluation of well-typed programs only, and in such programs exp will always have a functional type, so v will be either a closure, a constructor, a basic value or :=.

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

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

Matches

\[
E, v \vdash \text{match} \Rightarrow v'/p
\]

\[
E, v \vdash \text{mrule} \Rightarrow v'
\]
\[
E, v \vdash \text{mrule} \ F A I L
\]
\[
E, v \vdash \text{mrule} \Rightarrow [\text{match}, \{\}]
\]
\[
E, v \vdash \text{mrule} \ F A I L \quad E, v \vdash \text{match} \Rightarrow v'
\]

52
Comment: A value $v$ occurs on the left of the turnstile, in evaluating a match. We may think of a match as being evaluated against a value; similarly, we may think of a pattern as being evaluated against a value. Alternative match rules are tried from left to right.

**Match Rules**

\[
E, v \vdash mrule \Rightarrow v'/p/\text{FAIL}
\]

\[
v \vdash \text{pat} \Rightarrow VE \quad E + VE \vdash \text{exp} \Rightarrow v'
\]

\[
E, v \vdash \text{pat} => \text{exp} \Rightarrow v'
\] \hspace{1cm} (119)

\[
v \vdash \text{pat} \Rightarrow \text{FAIL}
\]

\[
E, v \vdash \text{pat} => \text{exp} \Rightarrow \text{FAIL}
\] \hspace{1cm} (120)

**Handlers**

\[
E, p \vdash \text{handler} \Rightarrow v/p
\]

\[
E, p \vdash hrule \Rightarrow v
\] \hspace{1cm} (121)

\[
E, p \vdash hrule (\mid | \mid \text{handler}) \Rightarrow v
\]

\[
E, p \vdash hrule \Rightarrow \text{FAIL}
\]

\[
E, p \vdash hrule \Rightarrow p
\] \hspace{1cm} (122)

\[
E, p \vdash hrule \Rightarrow \text{FAIL} \quad E, p \vdash \text{handler} \Rightarrow v
\]

\[
E, p \vdash hrule \mid | \mid \text{handler} \Rightarrow v
\] \hspace{1cm} (123)

**Handle Rules**

\[
E, p \vdash hrule \Rightarrow v/p/\text{FAIL}
\]

\[
E(\text{longezn}) \neq e
\]

\[
E, [e, v] \vdash \text{longezn with match} \Rightarrow \text{FAIL}
\] \hspace{1cm} (124)

\[
E(\text{longezn}) = e \quad E, v \vdash \text{match} \Rightarrow v'
\]

\[
E, [e, v] \vdash \text{longezn with match} \Rightarrow v'
\] \hspace{1cm} (125)

\[
E \vdash \text{exp} \Rightarrow v
\]

\[
E, p \vdash ? => \text{exp} \Rightarrow v
\] \hspace{1cm} (126)

Comments:

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

\[ E \vdash \text{valbind} \Rightarrow VE \]
\[ E \vdash \text{val valbind} \Rightarrow VE \text{ in Env} \]  
\[ E \vdash \text{exnbind} \Rightarrow EE \]
\[ E \vdash \text{exception exnbind} \Rightarrow EE \text{ in Env} \]
\[ E \vdash \text{dec}_1 \Rightarrow E_1 \quad E + E_1 \vdash \text{dec}_2 \Rightarrow E_2 \]
\[ E \vdash \text{local \ dec}_1 \text{ in \ dec}_2 \text{ end} \Rightarrow E_2 \]
\[ E(\text{longstrid}_1) = E_1 \quad \ldots \quad E(\text{longstrid}_k) = E_k \]
\[ E \vdash \text{open longstrid}_1 \ldots \text{longstrid}_n \Rightarrow E_1 + \ldots + E_k \]
\[ E \vdash \Rightarrow \{\} \text{ in Env} \]
\[ E \vdash \text{dec}_1 \Rightarrow E_1 \quad E + E_1 \vdash \text{dec}_2 \Rightarrow E_2 \]
\[ E \vdash \text{dec}_1 (;) \text{ dec}_2 \Rightarrow E_1 + E_2 \]

Value Bindings

\[ E \vdash \text{exp} \Rightarrow v \quad v \vdash \text{pat} \Rightarrow VE \quad (E \vdash \text{valbind} \Rightarrow VE') \]
\[ E \vdash \text{pat} = \text{exp} \text{ (and valbind) } \Rightarrow VE \text{ (} + \text{ VE')} \]
\[ E \vdash \text{exp} \Rightarrow v \quad v \vdash \text{pat} \Rightarrow \text{FAIL} \]
\[ E \vdash \text{pat} = \text{exp} \text{ (and valbind) } \Rightarrow [\text{bind}, \{\}] \]
\[ E \vdash \text{valbind} \Rightarrow VE \]
\[ E \vdash \text{rec valbind} \Rightarrow \text{RecVE} \]

Exception Bindings

\[ e \notin \text{excs of } s \quad s' = s + \{e\} \quad (s', E \vdash \text{exnbind} \Rightarrow EE, s'') \]
\[ s, E \vdash \text{exn (and exnbind) } \Rightarrow \{\text{exn} \leftrightarrow e\}{+ EE}, s'' \]
\[ E(\text{longexn}) = e \quad (E \vdash \text{exnbind} \Rightarrow EE) \]
\[ E \vdash \text{exn = longexn (and exnbind) } \Rightarrow \{\text{exn} \leftrightarrow e\}{+ EE} \]

Comments:

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

\[ v \vdash \text{atpat} \Rightarrow \text{VE/FAIL} \]

\[
\frac{
\text{\(v \vdash _\Rightarrow \{\}\)}
}{
\text{(138)}
}
\]

\[
\frac{
\text{\(v \vdash \text{var} \Rightarrow \{\text{var} \mapsto v\}\)}
}{
\text{(139)}
}
\]

\[
\frac{
\begin{align*}
\text{longcon} &= \text{strid}_1 \cdots \text{strid}_k \cdot \text{con} \\
\text{\(v \vdash \text{longcon} \Rightarrow \{\}\)}
\end{align*}
}{
\text{(140)}
}
\]

\[
\frac{
\begin{align*}
\text{longcon} &= \text{strid}_1 \cdots \text{strid}_k \cdot \text{con} \\
\text{\(v \vdash \text{longcon} \Rightarrow \text{FAIL}\)}
\end{align*}
}{
\text{(141)}
}
\]

\[
\frac{
\begin{align*}
\text{\(v = \{\}_{(+r)} \text{ in Val}\)} \\
\text{\(\langle r \vdash \text{patrow} \Rightarrow \text{VE/FAIL}\rangle\)}
\end{align*}
}{
\text{\(v \vdash \{\langle \text{patrow} \rangle\} \Rightarrow \{\}_{(+\text{VE/FAIL}}}\)
}
\]

\[
\frac{
\text{\(v \vdash \text{pat} \Rightarrow \text{VE}\)}
}{
\text{\(v \vdash \langle \text{pat} \rangle \Rightarrow \text{VE}\)}
}
\]

\[
\text{(143)}
\]

Comments:

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

Labelled Patterns

\[ r \vdash \text{patrow} \Rightarrow \text{VE/FAIL} \]

\[
\frac{
\text{\(r \vdash \ldots \Rightarrow \{\}\)}
}{
\text{(144)}
}
\]

\[
\frac{
\text{\(r(\text{lab}) \vdash \text{pat} \Rightarrow \text{FAIL}\)}
}{
\text{(145)}
}
\]

\[
\frac{
\text{\(r \vdash \text{lab} = \text{pat} \langle , \text{patrow} \rangle \Rightarrow \text{FAIL}\)}
}{
\text{(146)}
}
\]

Comments:

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

\[ v \vdash \text{atpat} \Rightarrow \text{VE/FAIL} \]

\[ v \vdash \text{atpat} \Rightarrow \text{VE/FAIL} \]

\[ \text{longcon} = \text{strid}_1, \ldots, \text{strid}_k, \text{con} \neq \text{ref} \quad v = (\text{con}, v') \]

\[ v' \vdash \text{atpat} \Rightarrow \text{VE/FAIL} \]

\[ v \vdash \text{longcon atpat} \Rightarrow \text{VE/FAIL} \]

(147)

\[ \text{longcon} = \text{strid}_1, \ldots, \text{strid}_k, \text{con} \neq \text{ref} \quad v \neq (\text{con}, v') \]

\[ v \vdash \text{longcon atpat} \Rightarrow \text{FAIL} \]

(148)

\[ s(a) = v \quad s, v \vdash \text{atpat} \Rightarrow \text{VE/FAIL}, s \]

\[ s, a \vdash \text{ref atpat} \Rightarrow \text{VE/FAIL}, s \]

(149)

\[ v \vdash \text{pat} \Rightarrow \text{VE/FAIL} \]

\[ v \vdash \text{var}(:: \text{ty}) \text{ as pat} \Rightarrow \{\text{var} \mapsto v\} + \text{VE/FAIL} \]

(150)

(151)

Comments:

(149) Any evaluation resulting in FAIL must do so because rule 141 or rule 149 has been applied.
7 Dynamic Semantics for Modules

7.1 Reduced Syntax

Since signatures are mostly dealt with in the static semantics, the dynamic semantics need only take limited account of them. Unlike types, it cannot ignore them completely; the reason is that an explicit signature ascription plays the role of restricting the "view" of a structure - that is, restricting the domains of its component environments. However, the types and the sharing properties of structures and signatures are irrelevant to dynamic evaluation; the syntax is therefore reduced by the following transformations (in addition to those for the Core), for the purpose of the dynamic semantics of Modules:

- Any specification of the form "type typdesc", "eqtype typdesc", "datatype datdesc" or "sharing shareq" is replaced by the empty specification.
- The Modules phrase classes typdesc, datdesc and shareq are omitted.

7.2 Compound Objects

The compound objects for the Modules dynamic semantics, extra to those for the Core dynamic semantics, are shown in Figure 17. An interface $I \in \text{Int}$ represents

$$
(\text{strid} : I, \text{strexp}(I'), B) \in \text{FunctorClosure} \\
= (\text{StrId} \times \text{Int}) \times (\text{StrExp} \times \text{Int}) \times \text{Basis} \\
(IE, \text{vars}, \text{exns}) \text{ or } I \in \text{Int} = \text{IntEnv} \times \text{Fin} (\text{Var}) \times \text{Fin} (\text{Exn}) \\
IE \in \text{IntEnv} = \text{StrId} \circ \text{Int} \\
G \in \text{SigEnv} = \text{SigId} \circ \text{Int} \\
F \in \text{FunEnv} = \text{FunId} \circ \text{FunctorClosure} \\
(F, G, E) \text{ or } B \in \text{Basis} = \text{FunEnv} \times \text{SigEnv} \times \text{Env} \\
(G, IE) \text{ or } IB \in \text{IntBasis} = \text{SigEnv} \times \text{IntEnv}
$$

Figure 17: Compound Semantic Objects

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

$$
\text{Inter} : \text{Env} \rightarrow \text{Int}
$$
as follows:

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

where

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

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

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

A further operation

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

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

$$(SE, VE, EE) \downarrow (IE, \text{vars}, \text{exns}) = (SE', VE', EE')$$

where

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

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

$$VE' = VE \downarrow \text{vars}, \ EE' = EE \downarrow \text{exns}.$$ 

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

### 7.3 Inference Rules

The semantic rules allow sentences of the form

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

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

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

Structure Expressions

\[ B \vdash \text{strexp} \Rightarrow E \]

\[ B \vdash \text{strdec} \Rightarrow E \]

\[ B \vdash \text{struct strdec end} \Rightarrow E \]

\[ B(\text{longstrid}) = E \]

\[ B \vdash \text{longstrid} \Rightarrow E \]

\[ B(\text{funid}) = (\text{strid} : I, \text{strexp'}(: I'), B') \]

\[ B \vdash \text{strexp} \Rightarrow E \quad B' + \{ \text{strid} \mapsto E \downarrow I \} \vdash \text{strexp'} \Rightarrow E' \]

\[ B \vdash \text{funid} \ (\text{strexp} ) \Rightarrow E'(\downarrow I') \]

\[ B \vdash \text{strdec} \Rightarrow E \quad B + E \vdash \text{strexp} \Rightarrow E' \]

\[ B \vdash \text{let strdec in strexp end} \Rightarrow E' \]

Comments:

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

Structure-level Declarations

\[ B \vdash \text{strdec} \Rightarrow E \]

\[ E \text{ of } B \vdash \text{dec} \Rightarrow E \]

\[ B \vdash \text{dec} \Rightarrow E \]

\[ B \vdash \text{strbind} \Rightarrow SE \]

\[ B \vdash \text{structure strbind} \Rightarrow SE \text{ in Env} \]

\[ B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B + E_1 \vdash \text{strdec}_2 \Rightarrow E_2 \]

\[ B \vdash \text{local strdec}_1 \text{ in strdec}_2 \text{ end} \Rightarrow E_2 \]

\[ B \vdash \Rightarrow \{\} \text{ in Env} \]
\[
\frac{B \vdash \text{strdec}_1 \Rightarrow E_1 \quad B + E_1 \vdash \text{strdec}_2 \Rightarrow E_2}{B \vdash \text{strdec}_1 \langle ; \rangle \text{ strdec}_2 \Rightarrow E_1 + E_2}
\]

(160)

**Structure Bindings**

\[
\frac{B \vdash \text{strexp} \Rightarrow E \quad \langle \text{Inter} \, B \vdash \text{sigexp} \Rightarrow I \rangle \quad \langle (B \vdash \text{strbind} \Rightarrow SE) \rangle}{B \vdash \text{strid} \langle : \text{sigexp} \rangle = \text{strexp} \langle \langle \text{and} \, \text{strbind} \rangle \rangle \Rightarrow \{ \text{strid} \rightarrow E(\downarrow I) \} \langle \langle + SE \rangle \rangle}
\]

(161)

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

**Signature Expressions**

\[
IB \vdash \text{sigexp} \Rightarrow I
\]

(162)

\[
\frac{IB \vdash \text{spec} \Rightarrow I}{IB \vdash \text{sig spec end} \Rightarrow I}
\]

(163)

**Signature Declarations**

\[
IB \vdash \text{sigdec} \Rightarrow G
\]

(164)

\[
\frac{IB \vdash \text{sigbind} \Rightarrow G}{IB \vdash \text{signature sigbind} \Rightarrow G}
\]

(165)

\[
\frac{IB \vdash \text{sigdec}_1 \Rightarrow G_1 \quad IB + G_1 \vdash \text{sigdec}_2 \Rightarrow G_2}{IB \vdash \text{sigdec}_1 \langle ; \rangle \text{ sigdec}_2 \Rightarrow G_1 + G_2}
\]

(166)

**Signature Bindings**

\[
\frac{IB \vdash \text{sigexp} \Rightarrow I \quad (IB \vdash \text{sigbind} \Rightarrow G)}{IB \vdash \text{sigid} = \text{sigexp} \langle \text{and} \, \text{sigbind} \rangle \Rightarrow \{ \text{sigid} \rightarrow I \} \langle \langle + G \rangle \rangle}
\]

(167)
Specifications

\[ \vdash \text{valdesc} \Rightarrow \text{vars} \]
\[ \vdash \text{exndesc} \Rightarrow \text{exns} \]
\[ \vdash \text{strdesc} \Rightarrow \text{IE} \]

(168)
(169)
(170)

\[ \vdash \text{spec}_1 \Rightarrow I_1 \quad \text{IB} + \text{IE} \text{ if } \text{IB} \vdash \text{spec}_1 \Rightarrow I_2 \]
\[ \vdash \text{local spec}_1 \text{ in spec}_2 \text{ end} \Rightarrow I_2 \]

(171)

\[ \text{IB}(\text{longstrid}_1) = I_1 \quad \ldots \quad \text{IB}(\text{longstrid}_n) = I_n \]
\[ \text{IB} \vdash \text{open longstrid}_1 \ldots \text{longstrid}_n \Rightarrow I_1 + \ldots + I_n \]

(172)

\[ \text{IB}(\text{sigid}_1) = I_1 \quad \ldots \quad \text{IB}(\text{sigid}_n) = I_n \]
\[ \text{IB} \vdash \text{include sigid}_1 \ldots \text{sigid}_n \Rightarrow I_1 + \ldots + I_n \]

(173)

\[ \text{IB} \vdash \text{spec} \Rightarrow \{\} \text{ in Int} \]
\[ \text{IB} \vdash \text{spec}_1 \Rightarrow I_1 \quad \text{IB} + \text{IE} \text{ if } \text{IB} \vdash \text{spec}_2 \Rightarrow I_2 \]
\[ \vdash \text{spec}_1 \langle ; \rangle \text{ spec}_2 \Rightarrow I_1 + I_2 \]

(174)
(175)

Comments:

(171),(175) Note that \text{vars} of \text{I}_1 \text{ and } \text{exns} \text{ of } \text{I}_1 \text{ are not needed for the evaluation of spec}_2.

Value Descriptions

\[ \vdash \text{valdesc} \Rightarrow \text{vars} \]

(176)

Exception Descriptions

\[ \vdash \text{exndesc} \Rightarrow \text{exns} \]

(177)
Structure Descriptions

\[ \text{IB} \vdash \text{strdesc} \Rightarrow \text{IE} \]

\[ \frac{\text{IB} \vdash \text{sigexp} \Rightarrow I \quad (\text{IB} \vdash \text{strdesc} \Rightarrow \text{IE})}{\text{IB} \vdash \text{strid} : \text{sigexp} \text{ (and strdesc)} \Rightarrow \{\text{strid} \mapsto I\} \langle + \text{IE} \rangle} \]  \hspace{1cm} (178)

Functor Bindings

\[ \text{B} \vdash \text{funbind} \Rightarrow \text{F} \]

\[ \frac{\text{Inter} \ B \vdash \text{sigexp} \Rightarrow I \quad \langle \text{Inter} \ B + \{\text{strid} \mapsto I\} \vdash \text{sigexp}' \Rightarrow I' \rangle \quad \langle (\text{B} \vdash \text{funbind} \Rightarrow \text{F}) \rangle}{\text{B} \vdash \text{funid} (\text{strid} : \text{sigexp}) (\text{strid} : \text{sigexp}') = \text{strexp} \langle \text{(and funbind)} \rangle \Rightarrow \{\text{funid} \mapsto (\text{strid} : I, \text{strexp}(I'), B)\} \langle + \text{F} \rangle} \]  \hspace{1cm} (179)

Functor Declarations

\[ \text{B} \vdash \text{funbind} \Rightarrow \text{F} \]

\[ \frac{\text{B} \vdash \text{funbind} \Rightarrow \text{F}}{\text{B} \vdash \text{functor funbind} \Rightarrow \text{F}} \]  \hspace{1cm} (180)

\[ \frac{\text{B} \vdash \text{funbind} \Rightarrow \text{F}}{\text{B} \vdash \text{functor funbind} \Rightarrow \text{F}} \]  \hspace{1cm} (181)

\[ \frac{\text{B} \vdash \text{fundec}_1 \Rightarrow F_1 \quad \text{B} + F_1 \vdash \text{fundec}_2 \Rightarrow F_2}{\text{B} \vdash \text{fundec}_1 (;) \text{fundec}_2 \Rightarrow F_1 + F_2} \]  \hspace{1cm} (182)

Programs

\[ \text{B} \vdash \text{program} \Rightarrow B' \]

\[ \frac{\text{B} \vdash \text{strdec} \Rightarrow E}{\text{B} \vdash \text{strdec} \Rightarrow E \text{ in Basis}} \]  \hspace{1cm} (183)

\[ \frac{\text{Inter} \ B \vdash \text{sigdec} \Rightarrow G}{\text{B} \vdash \text{sigdec} \Rightarrow G \text{ in Basis}} \]  \hspace{1cm} (184)

\[ \frac{\text{B} \vdash \text{fundec} \Rightarrow F}{\text{B} \vdash \text{fundec} \Rightarrow F \text{ in Basis}} \]  \hspace{1cm} (185)

\[ \frac{\text{B} \vdash \text{program}_1 \Rightarrow B_1 \quad \text{B} + B_1 \vdash \text{program}_2 \Rightarrow B_2}{\text{B} \vdash \text{program}_1 (;) \text{program}_2 \Rightarrow B_1 + B_2} \]  \hspace{1cm} (186)

62
A Appendix: Derived Forms

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

\[ \text{Derived form } \Rightarrow \text{ Equivalent form} \]

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

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

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

The following notes refer to Figure 20:

- In the equivalent form for a function-value binding, the variables \( \text{var}_1, \ldots, \text{var}_n \) must be chosen not to occur in the derived form. The condition \( m, n \geq 1 \) applies.

- In the two forms involving \( \text{twithtype} \), the identifiers bound by \( \text{datbind} \) and by \( \text{typbind} \) must be distinct. Then the transformed binding \( \text{datbind}' \) in the equivalent form is obtained from \( \text{datbind} \) by expanding out all the definitions made by \( \text{typbind} \). More precisely, if \( \text{typbind} \) is

\[
\text{tyvarseq}_1 \ \text{tycon}_1 = \text{ty}_1 \ \text{and} \ \ldots \ \text{and} \ \text{tyvarseq}_n \ \text{tycon}_n = \text{ty}_n
\]

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

\[
\text{ty}_i\{\text{tyseq}_i/\text{tyvarseq}_i\}
\]

- The abbreviation of \( \text{val it} = \text{exp} \) to \( \text{exp} \) is only permitted at top-level, i.e. as a \( \text{program} \).
<table>
<thead>
<tr>
<th>Derived Form</th>
<th>Equivalent Form</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EXPRESSIONS exp</strong></td>
<td></td>
</tr>
<tr>
<td>()</td>
<td><code>{ }</code></td>
</tr>
<tr>
<td>((exp_1, \ldots, exp_n))</td>
<td>{1=exp_1, \ldots, \overline{n}=exp_n}</td>
</tr>
<tr>
<td>#lab \text{ fn } {lab=var, \ldots} \Rightarrow var</td>
<td>raise longzn \text{ raise longzn with ()}</td>
</tr>
<tr>
<td>raise longzn</td>
<td>raise longzn with ()</td>
</tr>
<tr>
<td>case \text{ exp of match}</td>
<td>(fn match)(exp)</td>
</tr>
<tr>
<td>if \text{ exp then exp}_1 \text{ else exp}_2</td>
<td>case \text{ exp of true} \Rightarrow \text{ exp}_1 \text{ }</td>
</tr>
<tr>
<td>\text{ exp}_1 \text{ orelse } \text{ exp}_2</td>
<td>if \text{ exp}_1 \text{ then true else exp}_2</td>
</tr>
<tr>
<td>\text{ exp}_1 \text{ andalso } \text{ exp}_2</td>
<td>if \text{ exp}_1 \text{ then exp}_2 \text{ else false}</td>
</tr>
<tr>
<td>\text{ (exp}_1; \ldots; \text{ exp}_n; \text{ exp})</td>
<td>case \text{ exp}_1 \text{ of (_)} \Rightarrow \text{ ...} \text{ case exp}_n \text{ of (_)} \Rightarrow \text{ exp}</td>
</tr>
<tr>
<td>let \text{ dec in}</td>
<td>let \text{ dec in}</td>
</tr>
<tr>
<td>\text{ exp}_1; \ldots; \text{ exp}_n \text{ end}</td>
<td>(\text{ exp}_1; \ldots; \text{ exp}_n) \text{ end}</td>
</tr>
<tr>
<td>while \text{ exp}_1 \text{ do exp}_2</td>
<td>let \text{ val rec } var = \text{ fn } () \Rightarrow \text{ if } \text{ exp}_1 \text{ then } (\text{ exp}_2; \text{ var }()) \text{ else } () \text{ in var }() \text{ end}</td>
</tr>
<tr>
<td>\text{ [exp}_1, \ldots, \text{ exp}_n\text{]}</td>
<td>\text{ exp}_1::\ldots::\text{ exp}_n::\text{ nil}</td>
</tr>
</tbody>
</table>

Figure 18: Derived forms of Expressions
Derived Form | Equivalent Form
--- | ---
**HANDLING RULES** hrule
\[ longezn \Rightarrow \text{exp} \quad \text{longezn with } (\_ ) \Rightarrow \text{exp} \]

**PATTERNS** pat

\[
\begin{align*}
() & \quad \{ \} \\
(pat_1, \ldots, pat_n) & \quad \{1=pat_1, \ldots, n=pat_n\} \\
[pat_1, \ldots, pat_n] & \quad \text{pat}_1 :: \ldots :: \text{pat}_n :: \text{nil} \\
\end{align*}
\]
\( (n \geq 2) \)
\( (n \geq 0) \)

**LABELLED PATTERNS** patrow

\[
\begin{align*}
\text{id}(\text{:}ty) \; \text{(as pat)} \; \langle, \; \text{patrow} \rangle & \quad \text{id} = \text{id}(\text{:}ty) \; \text{(as pat)} \; \langle, \; \text{patrow} \rangle \\
\end{align*}
\]

**TYPES** ty

\[
\begin{align*}
ty_1 \ast \ldots \ast ty_n & \quad \{1:ty_1, \ldots, n:ty_n\} \\
\end{align*}
\]
\( (n \geq 2) \)

**FUNCTION-VALUE BINDINGS** fvalbind

\[
\begin{align*}
\langle \text{op} \rangle \text{var } \text{atpat}_{11} \cdots \text{atpat}_{1n}(\text{:ty}) = \text{exp}_1 \\
| \langle \text{op} \rangle \text{var } \text{atpat}_{21} \cdots \text{atpat}_{2n}(\text{:ty}) = \text{exp}_2 \\
| \cdots \cdots \\
| \langle \text{op} \rangle \text{var } \text{atpat}_{m1} \cdots \text{atpat}_{mn}(\text{:ty}) = \text{exp}_m \\
& \quad \text{(and } \text{fvalbind} \text{)} \\
\end{align*}
\]

**DECLARATIONS** dec

<table>
<thead>
<tr>
<th>fun fvalbind</th>
<th>val rec fvalbind</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype datbind withtype typbind</td>
<td>datatype datbind' ; type typbind</td>
</tr>
<tr>
<td>abstype datbind withtype typbind with dec end</td>
<td>abstype datbind' with type typbind ; dec end</td>
</tr>
<tr>
<td>exp</td>
<td>val it = exp</td>
</tr>
</tbody>
</table>

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

Figure 20: Derived forms of Function-value Bindings and Declarations
Appendix: Full Grammar of the Core

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

\[
\text{AtExp} \subset \text{AppExp} \subset \text{InfExp} \subset \text{Exp}
\]

The effect is that certain phrases, such as "2 + while \ldots do \ldots", are now disallowed.

The grammatical conventions are exactly as in Section 2, namely:

- The brackets \( (\ ) \) enclose optional phrases.

- For any syntax class \( X \) (over which \( x \) ranges) we define the syntax class \( X\text{seq} \) (over which \( x\text{seq} \) ranges) as follows:

\[
x\text{seq} ::= x \quad \text{(singleton sequence)}
\]

\[
x_1, \ldots, x_n \quad \text{(empty sequence)}
\]

\[
x_1, \ldots, x_n \quad \text{(sequence, \( n \geq 1 \))}
\]

(Note that the "\ldots" used here, a meta-symbol indicating syntactic repetition, 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.

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

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

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

The grammatical rules are displayed in Figures 21, 22, 23 and 24.
\[
\begin{align*}
\text{atexp} & \ ::= \ (\text{op})\text{longvar} & \text{value variable} \\
& \quad | (\text{op})\text{longcon} & \text{value constructor} \\
& \quad | \{\text{exprow}\} & \text{record} \\
& \quad | \#\text{lab} & \text{record selector} \\
& \quad | () & \text{0-tuple} \\
& \quad | (\text{exp}_1, \ldots, \text{exp}_n) & n\text{-tuple, } n \geq 2 \\
& \quad | [\text{exp}_1, \ldots, \text{exp}_n] & \text{list, } n \geq 0 \\
& \quad | (\text{exp}_1; \ldots; \text{exp}_n) & \text{sequence, } n \geq 1 \\
& \text{let } \text{dec} \text{ in } \text{exp}_1; \ldots; \text{exp}_n \text{ end} & \text{local declaration, } n \geq 1 \\
& \quad | (\text{exp}) & \\
\end{align*}
\]

\[
\begin{align*}
\text{exprow} & \ ::= \ \text{lab} = \text{exp} \langle , \text{exprow}\rangle & \text{expression row} \\
\text{appexp} & \ ::= \ \text{atexp} & \text{application expression} \\
& \quad | \text{appexp atexp} & \\
\text{infixp} & \ ::= \ \text{appexp} & \text{infix expression} \\
& \quad | \text{infixp id infixp} & \\
\text{exp} & \ ::= \ \text{infixp} & \text{typed (L)} \\
& \quad | \text{exp : ty} & \text{conjunction} \\
& \quad | \text{exp andalso exp} & \text{disjunction} \\
& \quad | \text{exp orelse exp} & \text{handle exception} \\
& \quad | \text{exp handle handler} & \text{raise exception} \\
& \quad | \text{raise longzn (with exp)} & \text{conditional} \\
& \quad | \text{if exp then exp}_1 \text{ else exp}_2 & \text{iteration} \\
& \quad | \text{while exp}_1 \text{ do exp}_2 & \text{case analysis} \\
& \quad | \text{case exp of match} & \text{function} \\
& \quad | \text{fn match} & \\
\end{align*}
\]

\[
\begin{align*}
\text{match} & \ ::= \ \text{mrule} \langle | \text{match}\rangle \\
\text{mrule} & \ ::= \ \text{pat} \Rightarrow \text{exp} \\
\text{handler} & \ ::= \ \text{hrule} \langle | | \text{handler}\rangle \\
\text{hrule} & \ ::= \ \text{longzn with match} & \\
& \quad | \text{longzn => exp} & \\
& \quad | \ ? \Rightarrow \text{exp} \\
\end{align*}
\]

Figure 21: Grammar: Expressions, Matches and Handlers
\[ \begin{align*}
dec &::= \text{val valbind} & \text{value declaration} \\
&| \text{fun fvalbind} & \text{function declaration} \\
&| \text{type typbind} & \text{type declaration} \\
&| \text{datatype datbind (withtype typbind)} & \text{datatype declaration} \\
&| \text{abstype datbind (withtype typbind)} & \text{abstype declaration} \\
&| \text{with dec end} & \\
&| \text{exception exnbind} & \text{exception declaration} \\
&| \text{local dec\textsubscript{1} in dec\textsubscript{2} end} & \text{local declaration} \\
&| \text{open longstrid\textsubscript{1} ... longstrid\textsubscript{n}} & \text{open declaration, } n \geq 1 \\
\end{align*} \]

\[ \begin{align*}
dec\textsubscript{1} (\cdot) \text{ dec\textsubscript{2}} & \\
\text{infix } (d) \ id\textsubscript{1} \ldots \id\textsubscript{n} & \text{infix (L) directive, } n \geq 1 \\
\text{infixr } (d) \ id\textsubscript{1} \ldots \id\textsubscript{n} & \text{infix (R) directive, } n \geq 1 \\
\text{nonfix } id\textsubscript{1} \ldots \id\textsubscript{n} & \text{nonfix directive, } n \geq 1 \\
\text{exp} & \text{expression (top-level only)} \\
\end{align*} \]

\[ \begin{align*}
\text{valbind} &::= \text{pat } = \text{ exp } \text{(and valbind)} \\
&| \text{rec valbind} \\
\text{fvalbind} &::= \langle \text{op} \rangle \text{ var } \text{ atpat\textsubscript{1}} \ldots \text{atpat\textsubscript{1\textsubscript{n}} (\cdot ty) = exp\textsubscript{1}} \\
&| \langle \text{op} \rangle \text{ var } \text{atpat\textsubscript{2}} \ldots \text{atpat\textsubscript{2\textsubscript{n}} (\cdot ty) = exp\textsubscript{2}} \\
&| \ldots \ldots \\
&| \langle \text{op} \rangle \text{ var } \text{atpat\textsubscript{m}} \ldots \text{atpat\textsubscript{m\textsubscript{n}} (\cdot ty) = exp\textsubscript{m}} \\
&| \langle \text{and} \ fvalbind \rangle \\
\text{typbind} &::= \text{tyvarseq tycon } = \text{ ty } \langle \text{and typbind} \rangle \\
\text{datbind} &::= \text{tyvarseq tycon } = \text{ constrs } \langle \text{and datbind} \rangle \\
\text{constrs} &::= \langle \text{op} \rangle \text{con } \langle \text{of ty} \rangle \ (| \text{constrs} \\
\text{exnbind} &::= \text{exn } \langle \cdot ty \rangle (\cdot longexn) \ (| \text{exnbind} \\
\end{align*} \]

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

Figure 22: Grammar: Declarations and Bindings
\[\text{atpat} ::= \begin{array}{l}
\text{wildcard} \\
\langle \text{op} \rangle \text{var} \\
\text{variable} \\
\text{longcon} \\
\text{constant} \\
\{ \langle \text{patrow} \rangle \} \\
\text{record} \\
() \\
0\text{-tuple} \\
(\text{pat}_1, \ldots, \text{pat}_n) \\
\text{n-tuple, } n \geq 2 \\
[\text{pat}_1, \ldots, \text{pat}_n] \\
\text{list, } n \geq 0 \\
(\text{pat}) \\
\end{array}\]

\[\text{patrow} ::= \ldots \text{wildcard} \]
\[
\text{lab} = \text{pat} (\ldots, \text{patrow}) \text{pattern row} \\
\text{id}(\langle \text{ty} \rangle) \langle \text{as pat} \rangle (\ldots, \text{patrow}) \text{label as variable} \\
\]

\[\text{pat} ::= \text{atpat} \text{atomic} \]
\[
\langle \text{op} \rangle \text{longcon atpat} \text{construction} \\
\text{pat}_1 \text{ con } \text{pat}_2 \text{ infixed construction} \\
\text{pat : ty} \text{typed} \\
\langle \text{op} \rangle \text{var}(\langle \text{ty} \rangle) \text{as pat} \text{layered} \\
\]

Figure 23: Grammar: Patterns

\[\text{ty} ::= \text{tyvar} \text{type variable} \]
\[
\{ \langle \text{tyrow} \rangle \} \text{record type expression} \\
\text{tyseq longtycon} \text{type construction} \\
\text{ty}_1 * \ldots * \text{ty}_n \text{tuple type, } n \geq 2 \\
\text{ty -> ty'} \text{function type expression} \\
(\text{ty}) \\
\]

\[\text{tyrow} ::= \text{lab} : \text{ty} (\ldots, \text{tyrow}) \text{type-expression row} \]

Figure 24: Grammar: Type expressions
C Appendix: The Initial Static Basis

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

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

where

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

The members of $T_0$ are type names, not type constructors; for convenience we have used type-constructor identifiers to stand also for the type names which are bound to them in the initial static type environment $TE_0$. Of these type names, list and ref have arity 1, the rest have arity 0; all except instream and outstream admit equality.

The components of $E_0$ are as follows:

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

70
<table>
<thead>
<tr>
<th>NONFIX</th>
<th>INFIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{map} \mapsto \forall \alpha \beta. (\alpha \rightarrow \beta) \rightarrow ) (\alpha \text{list} \rightarrow \beta \text{list})</td>
<td>\text{Precedence 7:}\</td>
</tr>
<tr>
<td>(\text{rev} \mapsto \forall \alpha. \text{alist} \rightarrow \text{alist})</td>
<td>(\div \mapsto \text{int} \times \text{int} \rightarrow \text{int})</td>
</tr>
<tr>
<td>(\text{not} \mapsto \text{bool} \rightarrow \text{bool})</td>
<td>(\mod \mapsto \text{int} \times \text{int} \rightarrow \text{int})</td>
</tr>
<tr>
<td>(- \mapsto \text{num} \rightarrow \text{num})</td>
<td>(* \mapsto \text{num} \times \text{num} \rightarrow \text{num})</td>
</tr>
<tr>
<td>(\text{abs} \mapsto \text{num} \rightarrow \text{num})</td>
<td>\text{Precedence 6:}\</td>
</tr>
<tr>
<td>(\text{floor} \mapsto \text{real} \rightarrow \text{int})</td>
<td>(+ \mapsto \text{num} \times \text{num} \rightarrow \text{num})</td>
</tr>
<tr>
<td>(\text{real} \mapsto \text{int} \rightarrow \text{real})</td>
<td>(- \mapsto \text{num} \times \text{num} \rightarrow \text{num})</td>
</tr>
<tr>
<td>(\text{sqrt} \mapsto \text{real} \rightarrow \text{real})</td>
<td>(^ {\wedge} \mapsto \text{string} \times \text{string} \rightarrow \text{string})</td>
</tr>
<tr>
<td>(\text{sin} \mapsto \text{real} \rightarrow \text{real})</td>
<td>\text{Precedence 5:}\</td>
</tr>
<tr>
<td>(\text{cos} \mapsto \text{real} \rightarrow \text{real})</td>
<td>(\vdash \mapsto \forall \alpha. \alpha \rightarrow \text{alist} \rightarrow \text{alist})</td>
</tr>
<tr>
<td>(\arctan \mapsto \text{real} \rightarrow \text{real})</td>
<td>(\odot \mapsto \forall \alpha. \text{alist} \times \text{alist} \rightarrow \text{alist})</td>
</tr>
<tr>
<td>(\exp \mapsto \text{real} \rightarrow \text{real})</td>
<td>\text{Precedence 4:}\</td>
</tr>
<tr>
<td>(\ln \mapsto \text{real} \rightarrow \text{real})</td>
<td>(= \mapsto \forall \eta. \eta \times \eta \rightarrow \text{bool})</td>
</tr>
<tr>
<td>(\text{size} \mapsto \text{string} \rightarrow \text{int})</td>
<td>(&lt; \mapsto \forall \eta. \eta \times \eta \rightarrow \text{bool})</td>
</tr>
<tr>
<td>(\text{chr} \mapsto \text{int} \rightarrow \text{string})</td>
<td>(&lt; \mapsto \text{num} \times \text{num} \rightarrow \text{bool})</td>
</tr>
<tr>
<td>(\text{ord} \mapsto \text{string} \rightarrow \text{int})</td>
<td>(&gt; \mapsto \text{num} \times \text{num} \rightarrow \text{bool})</td>
</tr>
<tr>
<td>(\text{explode} \mapsto \text{string} \rightarrow \text{string list})</td>
<td>(\leq \mapsto \text{num} \times \text{num} \rightarrow \text{bool})</td>
</tr>
<tr>
<td>(\text{implode} \mapsto \text{string list} \rightarrow \text{string})</td>
<td>(\geq \mapsto \text{num} \times \text{num} \rightarrow \text{bool})</td>
</tr>
<tr>
<td>(! \mapsto \forall \alpha. \text{aref} \rightarrow \alpha)</td>
<td>\text{Precedence 3:}\</td>
</tr>
<tr>
<td>(\text{ref} \mapsto \forall \alpha. \alpha \rightarrow \text{aref})</td>
<td>(\vdash \mapsto \forall \alpha. \text{aref} \times \alpha \rightarrow \text{unit})</td>
</tr>
<tr>
<td>(\text{true} \mapsto \text{bool})</td>
<td>(\odot \mapsto \forall \alpha \beta \gamma. (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))</td>
</tr>
<tr>
<td>(\text{false} \mapsto \text{bool})</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

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

- An identifier with type involving \text{num} stands for two functions – one in which \text{num} is replaced by \text{int} in its type, and another in which \text{num} is replaced by \text{real} in its type. In the case that both types can be inferred for an occurrence of the identifier, an explicit type constraint is needed to determine which type is intended.

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

Figure 25: Static \(VE_0\), except for Input/Output
\[
\begin{align*}
\text{std\_in} & \mapsto \text{instream} \\
\text{open\_in} & \mapsto \text{string} \to \text{instream} \\
\text{input} & \mapsto \text{instream} \times \text{int} \to \text{string} \\
\text{lookahead} & \mapsto \text{instream} \to \text{string} \\
\text{close\_in} & \mapsto \text{instream} \to \text{unit} \\
\text{end\_of\_stream} & \mapsto \text{instream} \to \text{bool} \\
\text{std\_out} & \mapsto \text{outstream} \\
\text{open\_out} & \mapsto \text{string} \to \text{outstream} \\
\text{output} & \mapsto \text{outstream} \times \text{string} \to \text{unit} \\
\text{close\_out} & \mapsto \text{outstream} \to \text{unit}
\end{align*}
\]

Figure 26: Static $VE_0$ (Input/Output)

\[
\begin{align*}
\text{unit} & \mapsto \{ \lambda().\}, \{\} \\
\text{bool} & \mapsto \{ \text{bool}, \{\text{true} \mapsto \text{bool}, \text{false} \mapsto \text{bool}\}\} \\
\text{int} & \mapsto \{ \text{int}, \{i \mapsto \text{int}; i \text{ an integer constant}\}\} \\
\text{real} & \mapsto \{ \text{real}, \{r \mapsto \text{real}; r \text{ a real constant}\}\} \\
\text{string} & \mapsto \{ \text{string}, \{s \mapsto \text{string}; s \text{ a string constant}\}\} \\
\text{list} & \mapsto \{ \text{list}, \{\text{nil} \mapsto \forall \alpha.\text{alist}, :: \mapsto \forall \alpha.\alpha \times \text{alist} \to \text{alist}\}\} \\
\text{ref} & \mapsto \{ \text{ref}, \{\text{ref} \mapsto \forall \alpha.\alpha \to \text{aref}\}\} \\
\text{instream} & \mapsto \{ \text{instream}, \{\}\} \\
\text{outstream} & \mapsto \{ \text{outstream}, \{\}\}
\end{align*}
\]

Figure 27: Static $TE_0$
D Appendix: The Initial Dynamic Basis

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

\[ B_0 = F_0, G_0, E_0 \]

where

- \( F_0 = \{\} \)
- \( G_0 = \{\} \)
- \( E_0 = E'_0 + E''_0 \)

\( E'_0 \) contains bindings of identifiers to the basic values BasVal and basic exceptions BasExc; in fact \( E'_0 = SE'_0, VE'_0 , EE'_0 \), where:

- \( SE'_0 = \{\} \)
- \( VE'_0 = \{id \mapsto id ; id \in \text{BasVal}\} \cup \{:=\mapsto :=\} \)
- \( EE'_0 = \{id \mapsto id ; id \in \text{BasExc}\} \)

Note that \( VE'_0 \) is the identity function on BasVal; this is because we have chosen to denote these values by the names of variables initially bound to them. The semantics of these basic values (most of which are functions) lies principally in their behaviour under APPLY, which we describe below. On the other hand the semantics of := is provided by a special semantic rule, rule 109. Similarly, \( EE'_0 \) is the identity function on BasExc, the set of basic exception names, because we have also chosen to denote these exceptions by the exception names initially bound to them. These exceptions are raised by APPLY as described below.

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

\[
\begin{align*}
\text{infix} & \quad 3 \; o \\
\text{infix} & \quad 4 \; = <> < > <= >= \\
\text{infix} & \quad 5 \; @ \\
\text{infixr} & \quad 5 \; :: \\
\text{infix} & \quad 6 \; + - \swarrow \\
\text{infix} & \quad 7 \; \text{div mod} / * \\
\text{fun} & \quad (F \circ G)x = F(G \; x)
\end{align*}
\]
fun nil @ M = M
    | (x::L) @ M = x::(L @ M)

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

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

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

fun not true = false
    | not false = true

fun ! (ref x) = x

We now describe the effect of APPLY upon each value in BasVal. We shall normally use i, r, n, s to range over integers, reals, numbers (integer or real), strings respectively. We also take the liberty of abbreviating “APPLY(abs, r)” to “abs(r)”, “APPLY(mod, {1 → i, 2 → d})” to “i mod d”, etc.

- \( ^{-}(n) \) returns the negation of \( n \).

- \( \text{abs}(n) \) returns the absolute value of \( n \).

- \( \text{floor}(r) \) returns the largest integer \( i \) not greater than \( r \); it returns the packet \([\text{floor},{}]\) if \( i \) is out of range.

- \( \text{real}(i) \) returns the real value equal to \( i \).

- \( \text{sqrt}(r) \) returns the square root of \( r \), or the packet \([\text{sqrt},{}]\) if \( r \) is negative.

- \( \text{sin}(r), \text{cos}(r) \) return the result of the appropriate trigonometric functions.

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

- \( \text{exp}(r), \text{ln}(r) \) return respectively the exponential and the natural logarithm of \( r \), or an exception packet \([\text{exp},{}]\) or \([\text{ln},{}]\) if the result is out of range.

- \( \text{size}(s) \) returns the number of characters in \( s \).
• \text{chr}(i) \text{ returns the } i^{\text{th}} \text{ ASCII character} \text{ – or the packet } [\text{chr}, \{\}] \text{ if none exists.}

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

• \text{explode}(s) \text{ returns the list of characters (as single-character lists) of which } s \text{ consists.}

• \text{implode}(L) \text{ returns the string formed by concatenating all members of the list } L \text{ of strings.}

• The arithmetic functions \(/,*,+,-\) all return the results of the usual arithmetic operations, or exception packets such as \([*,\{\}]\) if the result is out of range.

• \text{i mod } d, \text{ i div } d \text{ return integers } r, q \text{ (remainder, quotient) determined by the equation } d \times q + r = i, \text{ where either } 0 \leq r < d \text{ or } d < r \leq 0. \text{ Thus the remainder has the same sign as the divisor } d.

• The order relations \(<,>,<,=,>\) return boolean values in accord with their usual meanings.

• \text{v}_1 = \text{v}_2 \text{ returns the boolean value of } \text{v}_1 = \text{v}_2, \text{ where the equality of values (}=) \text{ is defined recursively as follows:}

  - If \text{v}_1, \text{v}_2 \text{ are constants (including nullary constructors) or addresses, then } \text{v}_1 = \text{v}_2 \text{ iff } \text{v}_1 \text{ and } \text{v}_2 \text{ are identical.}

  - (\text{con}_1, \text{v}_1) = (\text{con}_2, \text{v}_2) \text{ iff } \text{con}_1, \text{con}_2 \text{ are identical and } \text{v}_1 = \text{v}_2.

  - \text{r}_1 = \text{r}_2 \text{ (for records } \text{r}_1, \text{r}_2 \text{) iff } \text{Dom}\text{r}_1 = \text{Dom}\text{r}_2 \text{ and, for each } \text{lab} \in \text{Dom}\text{r}_1, \text{r}_1(\text{lab}) = \text{r}_2(\text{lab}).

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

• \text{v}_1 \not= \text{v}_2 \text{ returns the opposite boolean value to } \text{v}_1 = \text{v}_2.

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

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

75
— or *outstreams* — are of type *outstream* and will be denoted by *os*. Both these types of stream are *abstract*, in the sense that streams may only be manipulated by the functions provided in BasVal.

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

There are two streams in BasVal:

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

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

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

  \[\text{i o} \_\text{failure}, \text{"Cannot open } s\text{"}]\]

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

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

- *input(is,n)* returns a string *s* containing the first *n* characters of *is*, also removing them from *is*. If only \(k < n\) characters are available on *is*, then
  
  - If *is* is terminated after these *k* characters, the returned string *s* contains them alone, and they are removed from *is*.
  
  - Otherwise no result is returned until the producer of *is* either supplies *n* characters or terminates the stream.

- *lookahead(is)* returns a single-character string *s* containing the next character of *is*, without removing it. If no character is available on *is* then
  
  - If *is* is closed, the empty string is returned.
Otherwise no result is returned until the producer of is either supplies a character or closes the stream.

- `close_in(is)` empties and terminates the instream is.

- `end_of_stream(is)` is equivalent to `(lookahead(is)="")`; it detects the end of the instream is.

- `output(os, s)` writes the characters of s to the outstream os, unless os is closed, in which case it returns the exception packet

  ```
  [io_failure, "Output stream is closed"]
  ```

- `close_out(os)` terminates the outstream os.
Copyright © 1987, Laboratory for Foundations of Computer Science, University of Edinburgh. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.