Notes on the Standard ML Definition

Alex Nelson

August 1, 2024

Mathematics Subject Classification (MSC 2020): 68Q55, 68Q60, 03B70

Draft printing, September 2024.

Compiled: September 19, 2024 at 11:52AM (PDT)

This book was typeset by the author using the LaTeX typesetting system.

This revision: `1c5fc63`. September 19, 2024 11:52am PDT .

Contents

Preface

0.0.1 Remark: *Work in progress.*
This is a perpetual work in progress. Originally I hoped to learn enough of the Definition of Standard ML to recast it in a more palatable form, but that seems now to be a fool's errand.

I suspect I will work on this for a while, get bored, abandon it, then revisit it much later on and iterate this process.

> For small erections may be finished by their first architects; grand ones, true ones, ever leave the copestone to posterity. God keep me from ever completing anything. This whole book is but a draught—nay, but the draught of a draught. Oh, Time, Strength, Cash, and Patience!
>
> — Herman Melville, *Moby Dick*, Chapter 32 (1851)

0.0.2 Remark: *Commentary or reconstruction?*
This is an attempt to understand the 1997 Revised Definition [MTHM97], but I am not sure if it should be a "rational reconstruction" (using anachronistic tools and terms to "build from scratch" the contents of the definition) or a "commentary" (trying to articulate the rather terse text).

Milner and Tofte wrote a commentary [MT91] on the original 1990 Definition [MTH90]. It's arguable how similar the two Definitions are to each other. This makes the usefulness of Milner and Tofte's commentary subject to debate.

I want to be able to reason about Standard ML code with the results of this document. That's my goal. Towards that end, I suppose I need a "bit of both".

Specifically, this document is geared towards ***understanding*** *the Definition.* So I guess that makes it a "commentary" after all...

0.0.3 Remark: *"The Definition" refers to the 1997 definition.*
We will stop citing the 1997 Revised Definition [MTHM97] and henceforth refer to it as "the Definition" (capitalized). If we want to refer to the 1990 Definition [MTH90], then we will refer to it as "the 1990 Definition" (also capitalized).

When quoting explicitly from the Definition, we will still cite the bibliography entry, as well as provide the pages found in the printed version.

0.0.4 Remark: *Aside on pedagogical approaches.*
This is probably not the best way to *learn* the Definition. I suspect Milner and Tofte's Commentary [MT91] has the right approach pedagogically: start with some code snippet, then work through how the Definition elaborates and evaluates the example code. Do this for many different examples.

The ordering and choice in examples is then the difficulty for the author of such a commentary.

0.0.5 Remark: *Organization of this document.*
This commentary is organized "in parallel" to the Definition. That is to say, the chapters from 2 onwards is commenting on their correspondingly numbered chapters in the 1997 revised Definition.

This contrasts with Milner and Tofte's Commentary [MT91], which works through the 1990 Definition starting with evaluating expressions, then the static typechecking. Their commentary is probably more enjoyable (and certainly better written) than mine, but I am trying to understand the 1997 Definition completely.

0.0.6 Remark: *Method of working, species of propositions.*
We will break up the 1997 Definition into atomic propositions, definitions, and inference rules; then we will try to reconstruct what we can from these components. Each atomic proposition is numbered within each section, for ease of reference.

There are different "species" of propositions we offer. Right now, the basic heuristics for their meaning:

(1) "Deviation" means we are intentionally departing from the definition somehow (e.g., simplifying it by refusing to allow "=" to be a valid identifier that could be defined);
(2) "Clauses" are rules, constraints, conditions, etc., imposed or required by the Definition; (this term is chosen to be consistent with the terminology found in, e.g., IEEE standards documents — it is not used by the Definition);
(3) "Rules" are inference rules the Definition gives (plus some of my commentary explaining its signficance);
(4) "Comments" are comments which I am producing for the sake of commenting, clarifying, explicating;
(5) "Remarks" are... remarks — they are asides, not necessarily shedding light on the meaning of a clause or "ur text", but comparisons to how other languages do things, or what we've learned since the Definition was drafted; you know, things we find... remarkable... ;
(6) "Puzzle" are open questions which, as far as I know, do not have a solution (and I would be very excited to learn more about);
(7) "Definition" defines a new phrase or term;
(8) "Convention" usually refers to conventions adopted for metavariables or notation;
(9) "Example" offers an illustrative example of some concept, definition, or term;
(10) "Theorem" is self-explanatory;
(11) "Proposition" is usually a theorem which is not as important;

0.0.7 Deviation: *Fragment of Standard ML.*
We will work with the "fragment" of Standard ML which has the following constraints: in the same scope, we cannot redefine a function, constant, whatever.

When we declare a new structure (or signature), we *can* "reuse" an identifier and any references using that identifier refers to the "new" definition.

Any duplicate definitions — or reusing the same identifier in the same scope — for the same structure, type, variable, or data constructor will be flagged as an error.

This is the same constraint that, e.g., Haskell has on redefinitions.

[Used in: 3.6.1, 3.6.6]

0.0.8 Puzzle: *Formalize the definition using FS_0?*
Can we formalize the Definition using Feferman's FS_0 set theory? Ostensibly, this seems plausible, but I honestly do not know.

Arguably, the more natural metatheory would be some flavor of HOL.

0.0.9 Remark: *Acknowledgements.*
The formatting of propositions is heavily inspired by Alan U. Kennington's book on differential geometry [Ken24].

I am also heavily referring to the source of the Definition, which has been made available online for free.[1] I have borrowed some of their formatting macros. The wildcard pattern macro "_" I borrowed from LHS2TEX (if memory serves me), but the syntactic ellipses "$\cdots$" as well as the wildcard row pattern "..." I have hacked together.

Many of the comments which Kahrs [Kah93, Kah96] and Rossberg [Ros18b] have made have been integrated into this text, with citations to who observed what.

[1] https://github.com/SMLFamily/The-Definition-of-Standard-ML-Revised/

CHAPTER 1

INTRODUCTION

1.0.1 Comment: *Basic judgements in the Definition.*
In the preface to the 1997 Definition [MTHM97, pp.xi-xii], the authors explain in the preface:

> We shall now explain the keystone of our semantic method. First, we need a slight but important refinement. A phrase P is never evaluated *in vacuo* to a meaning M, but always against a background; this background — call it B — is itself a semantic object, being a distillation of the meanings preserved from evaluation of earlier phrases (typically variable declarations, procedure declarations, etc.). In fact evaluation is background-dependent — M depends upon B as well as upon P.
>
> The keystone of the method, then, is a certain kind of assertion about evaluation; it takes the form
>
> $$B \vdash P \Rightarrow M$$
>
> and may be pronounced: "Against the background B, the phrase P evaluates to the meaning M". *The formal purpose of this Definition is no more, and no less, than to decree exactly which assertions of this form are true.* This could be achieved in many ways. We have chosen to do it in a structured way, as others have, by giving rules which allow assertions about a *compound* phrase P to be inferred from assertions about its *constituent* phrases $P_1, \ldots, P_n$.

Unfortunately, programming language semantics evolved beyond this, and this approach seems archaic.

1.0.2 Comment: *Phases to the definition.*
Again, we find in the first chapter of the 1997 Definition, the authors explain (emphasis theirs):

> ML is an interactive language, and a *program* consists of a sequence of *top-level declarations*; the execution of each declaration modifies the top-level environment, which we call a *basis*, and reports the modification to the user.
>
> In the execution of a declaration there are three phases: *parsing*, *elaboration*, and *evaluation*. Parsing determines the grammatical form of a declaration. Elaboration, the *static* phase, determines whether it is well-typed and well-formed in other ways, and records relevant type or form information in the basis. Finally evaluation, the *dynamic* phase, determines the value of the declaration and records relevant value information in the basis. Corresponding to these phases, our formal definition divides into three parts: grammatical rules, elaboration rules, and evaluation rules. Furthermore, the basis is divided into the *static* basis and the *dynamic* basis; for example, a variable which has been declared is associated with a type in the static basis and with a value in the dynamic basis.

This gives us a clean way to order our discussion into three phases (parsing, elaboration, and evaluation) and two sublanguages (Core and Module). Since the Definition itself is structured in this manner, we could use that fact to mirror its structure in our text.

CHAPTER 2

SYNTAX OF THE CORE

2.1. RESERVED WORDS

2.1.1 Definition: *Reserved words.*
We are told the following are reserved words in the Core and may not be used as identifiers (except for "="):

```
abstype   and   andalso   as   case   datatype  do    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
(  )   [  ]   {  }  ,  :  ;  ...    _   |  =   =>   ->   #
```

[Used in: 2.5.1]

2.1.2 Comment: *Ambiguity about reserved words.*
Kahrs [Kah93] points out that there is ambiguity in the notion of "reserved words", since there are reserved words *in the Core* and reserved words *in Modules*. The Definition is not clear with what it means by the term (at least, it is not clear to me).

I interpret "reserved words" to refer to those appearing *either* in Core (listed above) *or* in Modules (§3.1.1).

2.2. SPECIAL CONSTANTS

2.2.1 Definition: *Integer constant.*
An **"Integer constant"** in decimal notation is "an optional negation symbol (`~`) followed by a non-empty sequence of decimal digits `0`, …, `9`." Similarly, in hexadecimal notation, an integer constant is an optional negation symbol followed by `0x` followed by a non-empty sequence of hexadecimal digits `0`, …, `9` and `a`, …, `f` (with `A`, …, `F` also allowed).

2.2.2 Definition: *Word constant.*
A **"Word constant"** in decimal notation is `0w` followed by a nonempty sequence of decimal digits, and in hexadecimal notation `0wx` followed by a non-empty sequence of hexadecimal digits.

2.2.3 Definition: *Real constant.*
A **"Real Constant"** is an integer constant in decimal notation, possibly followed by a point (`.`) and one or more decimal digits, possibly followed by an exponent symbol (`E` or `e`) and an integer constant in decimal notation; at least one of these optional parts must occur, therefore no integer constant is a real constant.

2.2.4 Clause: *Assumption on the character set.*
The Standard assumes an underlying alphabet of $N \geq 256$ characters numbered 0 to $N-1$, which agrees with the ASCII character set on the characters numbered 0 to 127.

2.2.5 Definition: *Ordinal range.*
The **"Ordinal range"** of the alphabet is the closed interval $[0, N-1]$.

2.2.6 Definition: *String constant.*
A **"String constant"** is a sequence, between quotation marks (`"`) of zero or more printable characters (i.e., numbered 33–126), spaces, or escape sequences.

Each escape sequence starts with the character "\" and stands for a character sequence. The escape sequences are:

`\a`	A single character interpreted by the system as alert (ASCII 7)
`\b`	Backspace (ASCII 8)
`\t`	Horizontal tab (ASCII 9)
`\n`	Linefeed, also known as newline (ASCII 10)
`\v`	Vertical tab (ASCII 11)
`\f`	Form feed (ASCII 12)
`\r`	Carriage return (ASCII 13)
`\^`*c*	The control character *c*, where *c* may be any character with number 64–95. The number of `\^`*c* is 64 less than the number of *c*.
\ddd	The single character with number *ddd* (3 decimal digits denoting an integer in the ordinal range of the alphabet).
`\u`*xxxx*	The single character with number *xxxx* (4 hexadecimal digits denoting an integer in the ordinal range of the alphabet).
`\"`	`"`
`\\`	`\`
\\$f \cdot\cdot f$\\	This sequence is ignored, where $f \cdot\cdot f$ stands for a sequence of one or more formatting characters.

2.2.7 Definition: *Formatting characters.*
The **"Formatting Characters"** are a subset of the non-printable characters including at least space, tab, newline, and formfeed. The formfeed 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.
[Used in: 2.5.2]

2.2.8 Definition: *Character constants.*
A **"Character constant"** is a sequence of the form `#`*s*, where *s* is a string constant denoting a string of size one character.

2.2.9 Definition: *Special Constants.*
The Definition denotes by SCon the class of **"Special Constants"**, i.e., the integer, real, word, character, and string constants. The definition uses *scon* to range over SCon.

2.3. COMMENTS

2.3.1 Definition: *Comment.*
A **"Comment"** is any character sequence within comment brackets "`(*`" and "`*)`" in which comment brackets are properly nested.
[Used in: 2.5.2]

2.3.2 Clause: *Space illegal in comment brackets.*
No space is allowed between the two characters which make up a comment bracket — i.e., no space is allowed in "`(*`" nor in "`*)`" if we want them to be comment brackets.

2.3.3 Clause: *Unterminated comments.*
An unmatched "`(*`" should be detected by the compiler.

2.4. IDENTIFIERS

2.4.1 DEFINITION: *Long identifiers, metavariable "longx" ranges over "longX".*
For each syntactic class X marked as "long", there is a class "longX" of **"Long Identifiers"**. If x ranges over X, then *longx* ranges over longX. The syntax of these long identifiers is given by:

$$\begin{array}{rcl} \mathit{longx} & ::= & x \\ & | & \mathit{strid}_1 . \cdots . \mathit{strid}_n . x \\ & & (n \geq 1) \end{array}$$

for an identifier and a long identifier, respectively.

The qualified identifiers link the Core and Modules.

[Used in: 2.5.1]

2.4.2 CLAUSE: *Classes of identifiers.*
There are five classes of identifiers:
(1) Value Identifiers (long) — VId
(2) Type Variables — TyVar
(3) Type Constructors (long) — TyCon
(4) Lab (record labels) — Lab
(5) Structure Identifiers (long) — StrId

2.4.3 DEFINITION: *Valid identifiers.*
An identifier is either:
(1) an **"Alphanumeric identifier"** consisting of any sequence of letters, digits, primes (`'`), and underbars (`_`) starting with a letter or a prime; or
(2) a **"Symbolic identifier"** consisting of any nonempty sequence of the following symbols:

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

In both cases, reserved words are excluded.

2.4.4 DEVIATION: *Equality is not a valid identifier.*
We will consciously deviate from the Definition and, consistent with our choices made earlier, insist that "`=`" is not a valid symbolic identifier. This deviates from the Definition, but it simplifies our life formalizing the Definition.

2.4.5 DEFINITION: *Type Variables.*
A **"Type Variable"** may be any alphanumeric identifier starting with a prime. The class of type variables is denoted "TyVar".

2.4.6 DEFINITION: *Subclass of equality type variables.*
The subclass of **"Equality Type Variables"** EtyVar of TyVar consists of those which start with two or more primes.

[Used in: 4.1.4]

2.4.7 CONVENTION: *VId, TyCon, Lab do not start with primes.*
The convention therefore is that the classes VId, TyCon, and Lab do not start with primes.

Also, "`*`" is excluded from TyCon to avoid confusion with derived form of tuple types.

2.4.8 DEFINITION: *Record labels and numeric labels.*
The syntactic class of **"Record Labels"**, denoted Lab, is the alphanumeric identifiers not starting with a prime, extended to include the **"Numeric Labels"** `1`, `2`, `3`, and so on (i.e., any numeral not starting with `0`). (This is so tuples can be syntactic sugar as a derived type.)

[Used in: 2.5.1]

2.4.9 Definition: *Structure identifiers.*
The identifier class of **"Structure Identifiers"**, denoted StrId, is represented by alphanumeric identifiers not starting with a prime.

2.4.10 Clause: *Determining class of identifier.*
The class of type variables is disjoint from the other four classes. But determining the class of an identifier *id* in a Core phrase is determined thus:
(1) Immediately before "`.`" — i.e., in a long identifier — or in an `open` declaration, *id* is a structure identifier. The rest of the cases assumes *id* is not a structure identifier.
(2) At the start of a component in a record type, record pattern, or record expression, *id* is a record label.
(3) Elsewhere in types, *id* is a type constructor.
(4) Elsewhere, *id* is a value identifier.

2.5. LEXICAL ANALYSIS

2.5.1 Clause: *Items of analysis.*
Each item of lexical analysis is either a reserved word (§2.1.1), a numeric label (§2.4.8), a special constant (Sec 2.2), or a long identifier (§2.4.1). (Remember, an identifier *is* a subclass of long identifiers.)

2.5.2 Clause: *Comments and formatting characters.*
Comments (§2.3.1) and formatting characters (§2.2.7) are separate items (except, of course, within string constants) and are otherwise ignored.

2.5.3 Clause: *Longest next item is always taken.*
At each stage, the longest next item is always taken.

2.6. INFIXED OPERATORS

2.6.1 Definition: *Infix status and infixed operators.*
An identifier may be given **"Infix Status"** by the `infix` or `infixr` directive, which may occur as a declaration. This status only pertains to its use as a *vid* within the scope of the directive, and in these uses it is called an **"Infixed Operator"**.

2.6.2 Clause: *Qualified identifiers never infixed.*
Qualified identifiers **never** have infix status. The Definition is very explicit about this.

2.6.3 Example: *Infixed operator.*
If *vid* has infix status, then "exp_1 *vid* exp_2" (resp., "pat_1 *vid* pat_2") may occur — possibly in parentheses — wherever the application "*vid*`{1=`exp_1`,2=`exp_2`}`" or its derived form "*vid*`(`exp_1`,`exp_2`)`" (resp., "*vid*`(`pat_1`,`pat_2`)`") would otherwise occur.

2.6.4 Clause: *Op-prefix treats identifier as non-infixed.*
An occurrence of *any* long identifier (qualified or not) prefixed by "`op`" is treated as non-infixed.

The only required use of an "`op`" is in prefixing a non-infixed occurrence of an identifier *vid* which has infix status. Elsewhere "`op`", where permitted, has no effect.

2.6.5 Definition: *Nonfix directive.*
Infix status is cancelled by the `nonfix` directive.

2.6.6 Definition: *Fixity directives.*
The keywords "`infix`", "`infixr`", and "`nonfix`" are collectively referred to as **"Fixity Directives"**.

2.6.7 Clause: *Syntax of fixity directives.*
For $n \geq 1$, we have the syntax for fixity directives look like:

$$\texttt{infix}\ \langle d\rangle\ vid_1\ \cdots\ vid_n$$

$$\texttt{infixr}\ \langle d\rangle\ vid_1\ \cdots\ vid_n$$

$$\texttt{nonfix}\ vid_1\ \cdots\ vid_n$$

where $\langle d\rangle$ is an optional decimal digit d indicating binding precedence. A higher value of d indicates tighter binding; the default is `0`.

2.6.8 Comment: *Precedence must be between 0 and 9.*
Observe this part of the Definition explicit constrains the precedence for infixed operators must be between 0 and 9. In principle, there's no reason it cannot be something larger (say, representable as an unsigned byte or word — i.e., between 0 and 256, or between 0 and $2^{64} \approx 18.4 \times 10^{18}$).

For what it's worth, in S4.4.2 of the Haskell 2010 report[1], it appears that Haskell has the precedence [which it calls the "fixity"] "must be in the range 0 to 9". So Haskell is following Standard ML here, neat.

2.6.9 Clause: *Associativity of infixed operators.*
Here "`infix`" dictates left associativity, "`infixr`" dictates right associativity.

The Definition requires in an expression of the form $exp_1\ vid_1\ exp_2\ vid_2\ exp_3$ where vid_1 and vid_2 are infixed operators with the same precedence, either both must associate to the left or both must associate to the right.

2.6.10 Example: *Infixed operator associativity.*
Suppose "`<<`" and "`>>`" are infixed operators of equal precedence, but "`<<`" is left-associative and "`>>`" is right associative. Then

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

2.6.11 Definition: *Scope of fixity directive.*
The **"Scope"** of a fixity directive *dir* is the ensuing program text, except if *dir* occurs in a declaration *dec* in either of the phrases

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

or

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

then the scope of *dir* does not extend beyond these phrases. Further scope limitations are imposed for Modules.
[Used in: 3.3.1]

2.6.12 Clause: *Do not appear in semantic rules.*
The Definition states: These directives and "`op`" are omitted from the semantic rules, since they only affect parsing.

[1] http://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-820004.4.2

2.7. DERIVED FORMS

2.7.1 CLAUSE: *Syntactic sugar atop the bare language.*
Many syntactic forms in Standard ML can be expressed in terms of a smaller number of syntactic forms which the Definition calls the **"Bare"** language. The derived forms and their equivalents are in Appendix A.

2.8. GRAMMAR

2.8.1 CONVENTION: *Pseudo-EBNF syntax.*
The Definition uses an idiosyncratic EBNF-like syntax. The rules governing it:
(1) The brackets $\langle\ \rangle$ enclose optional phrases.
(2) For any syntax class X (over which x ranges) we can define the class Xseq (over which *xseq* ranges) as follows:

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

Here the "$\cdots$" used means syntactic iteration, which is distinct from (and must not be confused with) "..." which is a reserved keyword of the language.
(3) Alternative forms for each phrase class are in order of decreasing precedence; this resolves ambiguity in parsing (or so the Definition assures us, guiding us to Appendix B)
(4) L (resp., R) means left (resp., right) association
(5) Syntax of types binds more tightly than the syntax of expressions
(6) Each iterated construct (e.g., *match*, $\cdots$) extends as far right as possible; therefore, in theory, parentheses may be needed around an expression which terminates with a match (like "`fn` *match*") if this occurs within a larger match. (I guess you could say *care must be taken if you play with matches...*)

Unspoken rules:
(7) "nonterminals" are italicized (*like so*), "terminals" are written in teletype (`like so`).

2.8.2 COMMENT: *"Match extends as far right as possible" context-free?*
I am not sure if this criteria "extends as far right as possible" is context-free, i.e., we might accidentally have a context-sensitive (or worse) grammar accidentally with this seemingly innocuous condition.

Scott, Johnstone, and Walsh [SJW23] have written a paper about this sort of condition. It seems like it is innocent enough, but there may be difficulties or unintended complications with how the Definition uses it.

2.8.3 DEFINITION: *Syntactic classes for Core.*
The syntactic classes for Core, which the Definition calls **"Core Phrase Classes"**, consists of the following:

AtExp	atomic expressions
ExpRow	expression rows
Exp	expressions
Match	matches
Mrule	match rules
Dec	declarations
ValBind	value bindings
TypBind	type bindings
DatBind	datatype bindings
ConBind	constructor bindings
ExBind	exception bindings

AtPat	atomic patterns
PatRow	pattern rows
Pat	patterns
Ty	type expressions
TyRow	type-expression rows

As usual, the convention is to use italicized lowercase names x for metavariables ranging over the syntactic class X.

2.8.4 GRAMMAR: *Syntax for patterns.*
The syntax for patterns:

atpat	::=	`_`	wildcard
	$\mid$	*scon*	special constant
	$\mid$	⟨`op`⟩ *longvid*	value identifier
	$\mid$	`{` ⟨*patrow*⟩ `}`	record
	$\mid$	`{` *pat* `}`	
patrow	::=	...	wildcard
	$\mid$	*lab* `=` *pat* ⟨`,` *patrow*⟩	pattern row
pat	::=	*atpat*	atomic
	$\mid$	⟨`op`⟩ *longvid atpat*	constructed pattern
	$\mid$	pat_1 *vid* pat_2	infixed value construction
	$\mid$	*pat* `:` *ty*	typed
	$\mid$	⟨`op`⟩*vid*⟨`:` *ty*⟩ `as` *pat*	layered

2.8.5 GRAMMAR: *Syntax for types.*
The syntax for types consists of the production rules:

ty	::=	*tyvar*	type variable
	$\mid$	`{` ⟨*tyrow*⟩ `}`	record type expression
	$\mid$	*tyseq longtycon*	type construction
	$\mid$	*ty* `->` *ty′*	function type expression (R)
	$\mid$	`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨`,` *tyrow*⟩	type-expression row

2.8.6 GRAMMAR: *Expressions.*
Remember that "special constants" is what the Definition refers to "literal constants".

atexp	::=	*scon*	special constant
	$\mid$	⟨`op`⟩ *longvid*	value identifier
	$\mid$	`{` ⟨*exprow*⟩ `}`	record
	$\mid$	`let` *dec* `in` *exp* `end`	local declaration
	$\mid$	`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨`,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic
	$\mid$	*exp atexp*	application (L)
	$\mid$	exp_1 *vid* exp_2	infixed application
	$\mid$	*exp* `:` *ty*	typed (L)
	$\mid$	*exp* `handle` *match*	handle exception
	$\mid$	`raise` *exp*	raise exception
	$\mid$	`fn` *match*	function

2.8.7 GRAMMAR: *"Matches".*
In `case`-expressions, and function declarations (and applications), we have pattern matching based on a set of rules. These are usually called "matches".

match	::=	*mrule* ⟨| *match*⟩
mrule	::=	*pat* `=>` *exp*

2.8.8 GRAMMAR: *Core declarations.*
This is the heart of what top-level programs look like, they're a sequence of declarations (of some sort).

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

2.8.9 COMMENT: *Ambiguous grammar for dec.*
Rossberg [Ros18b] observes that the grammar given for *dec* is highly ambiguous. For example, combining the empty declaration *and* sequencing allows the derivation of arbitrary sequences of empty declarations for *any* input.

A second ambiguity Rossberg points out is that a sequence of the form "dec_1 dec_2 dec_3" can be reduced in two ways to *dec*: either via "dec_{12} dec_3" or via "dec_1 dec_{23}". Rossberg cites Kahrs [Kah93, S8.3] for this second ambiguity, which Kahrs expands upon with several more severe ambiguities.

2.8.10 COMMENT: *Proposed alternative grammar for declarations.*
It seems to me the correct grammar for declarations should be to introduce a syntactic class for 'atomic declarations' "*adec*", another syntactic class for a [possibly semicolon separated, right associative[2]] sequence of atomic declarations "*adecseq*", and then just make "*dec*" either empty or an *adecseq*:

adec	::=	`val` *tyvarseq valbind*	value declaration
	|	`type` *typbind*	type declaration
	|	`datatype` *datbind*	datatype declaration
	|	`datatype` *tycon* `-=-` `datatype` *longtycon*	datatype replication
	|	`abstype` *datbind* `with` *dec* `end`	abstype declaration
	|	`exception` *exbind*	exception declaration
	|	`local` dec_1 `in` dec_2 `end`	local declaration
	|	`open` $longstrid_1 \cdots longstrid_n$	open declaration ($n \geq 1$)
	|	`infix` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (L) directive
	|	`infixr` ⟨*d*⟩ $vid_1 \cdots vid_n$	infix (R) directive
	|	`nonfix` ⟨*d*⟩ $vid_1 \cdots vid_n$	nonfix directive
adecseq	::=	*adec* ⟨`;`⟩	a single declaration
	|	*adec* ⟨`;`⟩ *adecseq*	sequential declarations
dec	::=		empty declaration

[2]HaMLet parses sequences of declarations as left associative, according to §4.4 of its documentation.

	adecseq	sequential declarations

2.9. SYNTACTIC RESTRICTIONS

2.9.1 Clause: *No repeated labels in records.*
No expression row, pattern row, or type-expression row may bind the same *lab* twice.
[Used in: 4.10.45]

2.9.2 Clause: *Cannot bind same identifier twice.*
No binding *valbind*, *typbind*, *datbind*, or *exbind* may bind the same identifier twice. This applies also to value identifiers within a *datbind*.
[Used in: 4.10.30, 4.10.32, 4.10.34, 4.10.35]

2.9.3 Clause: *Type variables sequence uniqueness.*
No *tyvarseq* may contain the same *tyvar* twice. So `('a, 'b, 'a)` is illegal since `'a` appears twice.
[Used in: 4.10.32]

2.9.4 Clause: `rec` *restriction for derived function form.*
For each value binding "*pat* `=` *exp*" within `rec`, *exp* must be of the form "`fn` *match*". The derived form of function-value binding given in Appendix A necessarily obeys this restriction.

2.9.5 Clause: *No binding primitive data constructors.*
No *datbind*, *valbind*, or *exbind* may bind `true`, `false`, `nil`, `::`, or `ref`.

2.9.6 Clause: *No data or exception binding to* `it`*.*
No *datbind* or *exbind* may bind `it`.

2.9.7 Clause: *Real constants never appear in patterns.*
No real constant may occur in a pattern. (Really, unless you're a numerical analyst, you should be staying away from floating-point arithmetic altogether…)

2.9.8 Comment: *Problem stems from* `real` *is not equality type.*
The difficulty with the previous clause (real constants cannot appear in patterns) stems from `real` is not an equality type. This (real not being an equality type) has felt wrong to me, since the IEEE 754 standard *does* define equality for floating-point numbers.

Moreover, it's the *only* type which is not an equality type, and that causes serious complications — it's the entire reason we need to distinguish "equality types" from "non-equality types"! But if we allow equality for real constants, the only possible surprise is that `NaN` is never equal to anything else (i.e., we lose reflexivity).

2.9.9 Clause: *Type variable restrictions in value declarations.*
In a value declaration "`val` *tyvarseq valbind*", if *valbind* contains another value declaration "`val` *tyvarseq′ valbind′*", then *tyvarseq* and *tyvarseq′* must be disjoint.

In other words, no type variable may be scoped by two value declarations of which one occurs inside the other.

This restriction applies after *tyvarseq* and *tyvarseq′* have been extended to include implicitly scoped type variables (as explained in Section 4.6).
[Used in: 4.8.3, 4.10.30]

CHAPTER 3

SYNTAX OF MODULES

3.1. RESERVED WORDS

3.1.1 Definition: *Reserved words.*
The modules have additional reservewords, namely:

```
eqtype    functor   include   sharing   sig
signature   struct   structure   where   :>
```

[Used in: 2.1.2]

3.2. IDENTIFIERS

3.2.1 Definition: *Functor identifier syntactic class FunId.*
The functor identifier syntactic class, denoted FunId, consists of alphanumeric identifiers not starting with a prime.

3.2.2 Definition: *Signature Identifier syntactic class SigId.*
The signature identifier syntactic class, denoted SigId, also consists of alphanumeric identifiers not starting with a prime.

3.2.3 Convention: *Disjointness of identifier classes.*
Henceforth, the Definition considers all identifier classes to be disjoint and discernible by grammatical rules.

3.3. INFIXED OPERATORS

3.3.1 Clause: *Additional scoping rules.*
In addition to the rules governing the scope of infixed operators in the Core language (§2.6.11), there is a further scope limitation: if *dir* occurs in a structure-level declaration *strdec* in any of the phrases

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

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

$$\texttt{struct}\ \mathit{strdec}\ \texttt{end}$$

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

An immediate consequence: fixity is local to a basic structure expression (in particular, to such an expression occurring as a functor body).

3.4. GRAMMAR FOR MODULES

3.4.1 DEFINITION: *Syntactic classes for modules.*
The syntactic classes for modules, which the Definition calls **“Module Phrase Classes”**, consists of the following:

StrExp	structure expressions
StrDec	structure-level declarations
StrBind	structure bindings
SigExp	signature expressions
SigDec	signature declarations
SigBind	signature bindings
Spec	specifications
ValDesc	value descriptions
TypDesc	type descriptions
DatDesc	datatype descriptions
ConDesc	constructor descriptions
ExDesc	exception descriptions
StrDesc	structure descriptions
FunDec	functor declarations
FunBind	functor bindings
TopDec	top-level declarations

3.5. GRAMMAR FOR MODULES

3.5.1 GRAMMAR: *Structures.*
The syntax for structures is fairly straightforward. Note that the rule for structure declarations *strdec* includes the rule for core declarations *dec*, which is particularly important later on.

strexp	::=	`struct` *strdec* `end`	basic
	|	*longstrid*	structure identifier
	|	*strexp* `:` *sigexp*	transparent constraint
	|	*strexp* `:>` *sigexp*	opaque constraint
	|	*funid*`(`*strexp*`)`	functor application
	|	`let` *strdec* `in` *strexp* `end`	local declaration
strdec	::=	*dec*	declaration
	|	`let structure` *strbind*	structure
	|	`local` $strdec_1$ `in` $strdec_2$ `end`	local declaration
	|		empty
	|	$strdec_1$ ⟨`;`⟩ $strdec_2$	sequential
strbind	::=	*strid* `=` *strexp* ⟨`and` *strbind*⟩	

3.5.2 COMMENT: *Ambiguous grammar for strdec.*
Rossberg [Ros18b] notes that the production rule for sequential structure declarations “$strdec_1$ ⟨`;`⟩ $strdec_2$” is ambiguous. Is it left-associative or right-associative?

I’d advocate for making it right recursive, so we could write a simple recursive descent parser for Standard ML.

3.5.3 GRAMMAR: *Signatures.*
The syntax for signatures:

sigexp	::=	`sig` *spec* `end`	basic
	|	*sigid*	signature identifier
	|	*sigexp* `where type` *tyvarseq longtycon* `=` *ty*	type realisation

sigdec	::=	`signature` *sigbind*
sigbind	::=	*sigid* `=` *sigexp* ⟨`and` *sigbind*⟩

3.5.4 Grammar: *Specifications.*
The syntax for specifications which appear in the body of a signature:

spec	::=	`val` *valdesc*	value
	|	`type` *typdesc*	type
	|	`eqtype` *typdesc*	equality type
	|	`datatype` *datdesc*	datatype
	|	`datatype` *tycon* `-=-` `datatype` *longtycon*	replication
	|	`exception` *exdesc*	exception
	|	`structure` *strdesc*	structure
	|	`include` *sigexp*	include
	|		empty
	|	$spec_1$ ⟨`;`⟩ $spec_2$	sequential
	|	*spec* `sharing` `type` $longtycon_1$ `=` ⋯ `=` $longtycon_n$	sharing ($n \geq 2$)
valdesc	::=	*vid* `:` *ty* ⟨`and` *valdesc*⟩	
typdesc	::=	*tyvarseq* *tycon* ⟨`and` *typdesc*⟩	
datdesc	::=	*tyvarseq* *tycon* `=` *condesc* ⟨`and` *datdesc*⟩	
condesc	::=	*vid* ⟨`of` *ty*⟩ ⟨`|` *condesc*⟩	
exdesc	::=	*vid* ⟨`of` *ty*⟩ ⟨`|` *exdesc*⟩	
strdesc	::=	*strid* `:` *sigexp* ⟨`and` *strdesc*⟩	

3.5.5 Grammar: *Functor declarations.*
The syntax for declaring a functor reuses the syntax for structure expressions. Also observe that it is impossible to have a functor with 0 arguments, or with $n > 1$ arguments.

fund	::=	`functor` *funbind*	
funbind	::=	*funid* `(`*strid* `:` *sigexp*`)` `=` *strexp* ⟨`and` *funbind*⟩	functor binding

3.5.6 Grammar: *Top-level declarations.*
The syntax for top-level declarations (which forms the basis of a program) consists of a sequence of one or more structure declarations [remember: core-language declarations are swept into structure declarations], signature declarations, and/or functor declarations:

topdec	::=	*strdec* ⟨*topdec*⟩	structure-level declaration
	|	*sigdec* ⟨*topdec*⟩	signature declaration
	|	*fundec* ⟨*topdec*⟩	functor declaration

3.5.7 Puzzle: *LL(1) grammar for Standard ML?.*
Can we revise the grammar for Standard ML (both Core and Modules) to make it explicitly LL(1)?

3.6. SYNTACTIC RESTRICTIONS

3.6.1 Clause: *No rebinding the same identifier in simultaneous declarations.*
No binding *strbind*, *sigbind*, or *funbind* may bind the same identifier twice. The Definition means by this, if we have something like "`functor foo(arg1 : sig1) = ... and foo(arg2 : sig2) = ...`" then we have both functor declarations use the same identifier "`foo`".

However, our deviation (§0.0.7) already imposes a stronger constraint.

3.6.2 Clause: *Specifications cannot reuse identifiers.*
No description *valdesc*, *typdesc*, *datdesc*, *exdesc*, or *strdesc* may describe the same identifier twice. This also applies to value identifiers within a *datdesc*. (Again, we're imposing a stronger constraint…)

3.6.3 Clause: *Type variables must be distinct.*
No *tyvarseq* may contain the same *tyvar* twice.

3.6.4 Clause: *Type variables in data constructor appear in type constructor.*
Any *tyvar* appearing on the right side of a *datdesc* of the form "*tyvarseq tycon* `=` ⋯" must occur in the *tyvarseq*. Similarly, in signature expressions of the form "*sigexp* `where type` *tyvarseq longtycon* `=` *ty*", any *tyvar* occurring in *ty* must occur in *tyvarseq*.

3.6.5 Clause: *Specification identifiers are not reserved words.*
No *datdesc*, *valdesc*, or *exdesc* "may describe" [whatever that means] `true`, `false`, `nil`, `::`, or `ref`.

Further, no *datdesc* or *exdesc* "may describe" `it`.

3.6.6 Comment: *Constraining our deviation.*
So our deviation (§0.0.7) from the "full" language imposed the constraint that redefinitions are illegal, but we can "shadow" an identifier if it is inside a structure. We see that the Definition further limits the possible identifiers we can shadow *in general.*

3.6.7 Clause: *Top declaration restriction.*
No *topdec* may contain, as an initial segment, a *strdec* followed by a semicolon. (Rossberg [Ros18b] observes that the *intention* is to make parsing of top-level semicolons unambiguous, so that they always terminate a program.)
[Used in: 8.0.13]

3.6.8 Comment: *Semicolons treated worse than a dead dog.*
It seems to me that semicolons are being treated worse than a dead dog, and it might actually improve the language if every statement and declaration had to end with a semicolons. Or if semicolons were banned altogether. It's got to be one or the other, because right now it's this muddied middle-of-the-road strategy (which is a great way to get hit by a car).

3.6.9 Comment: *References on Standard ML syntax.*
Scott and Jonstone [SJ23] have re-examined Standard ML's syntax using modern lexical tools.

CHAPTER 4

STATIC SEMANTICS FOR THE CORE

4.0.1 Remark: *Strategy: introduce "semantic objects".*
The Definition will introduce what is vaguely described as "semantic objects", then use these to present the semantics for both Core and Modules, both static and dynamic. We'll probably need to expand the notion of "semantic object" as needed in the process.

Towards that end, we will have "simple" semantic objects (usually associated with an identifier of some kind) which will eventually combine into "compound" semantic objects (like types or environments).

4.0.2 Remark: *Definition describes how to produce trees.*
The Definition describes how to produce (what Milner and Tofte [MT91] call) 'elaboration trees' using the static semantics, and 'evaluation trees' using the dynamic semantics. This may be surprising to readers familiar with type theory having learned it from, e.g., TAPL where evaluation is just a judgement form described using inference rules.

So be forewarned: the static module chapter describes how to form these elaboration trees, the static core chapter describes one particular form of a "structure declaration". The dynamic chapters describe analogous results with the evaluation trees. The last chapter describes how to combine the two for a program.

4.1. SIMPLE OBJECTS

4.1.1 Definition: *Type constructor names.*
The **"Type Constructor Names"** are the values taken by type constructors. The Definition usually refers to them as "type names", but they must be clearly distinguished from "type variables" and "type constructors".

4.1.2 Clause: *Simple objects = Identifiers + type constructors + id status.*
All simple objects in static semantics of the language are built from identifiers, type constructor names, and identifier status descriptors. The Definition offers the following table for the possible simple semantic objects:

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

4.1.3 Comment: *Type names internally track* `datatype` *type operators.*
Just for clarity's sake, the $t \in$ TyName are metasymbols which are freshly generated whenever a `datatype` declaration is encountered. This technically differs from a type operator (which is constructed whenever a `type` alias is encountered).

But we can still meaningfully talk about the "arity" of a type name. The Definition allows for eta-equivalence, so $\Lambda\alpha^{(k)}.t$ is replaceable by t.

4.1.4 Definition: *Equality attribute, admitting equality.*
Each $\alpha \in$ TyVar possesses a Boolean **"Equality"** attribute, which determines whether or not it **"Admits Equality"**, i.e., whether it is a member of EtyVar (§2.4.6)

4.1.5 DEFINITION: *Attributes of type names.*
Each $t \in \mathrm{TyName}$ has an **"Arity"** $k \geq 0$, and also posses an **"Equality Attribute"**. The Definition denotes the class of type names with arity k by $\mathrm{TyName}^{(k)}$.

4.1.6 DEFINITION: *Type of special constants.*
With each special constant *scon* we associate a **"Type Name"**, denoted type(*scon*), which is either `int`, `real`, `word`, `char`, or `string` as indicated by Section 2.2.

4.2. COMPOUND OBJECTS

4.2.1 DEFINITION: *Collection of finite subsets.*
Let A be a set. The Definition uses the notation "Fin A" for the collection of all finite subsets of A.

4.2.2 DEFINITION: *Finite maps.*
Let A, B be sets. Then the Definition denotes by "$A \xrightarrow{\text{fin}} B$" the set of all **"Finite Maps"** (partial functions with finite domain) from A to B.

The domain and range of a finite map f are denoted "Dom f" and "Ran f", respectively.

A finite map may be written explicitly in the form $\{a_1 \mapsto b_1, \cdots, a_k \mapsto b_k\}$, $k \geq 0$; in particular the empty map is $\{\}$.

4.2.3 CONVENTION: *Constructing finite map by set comprehension.*
The Definition uses the notation $\{x \mapsto e\ ;\ \phi\}$ — a form of set comprehension — to stand for the finite map f whose domain is the set of values x which satisfy the condition ϕ, and whose value on this domain is given by $f(x) = e$.
[Used in: 4.8.3]

4.2.4 DEFINITION: *Modifying finite maps.*
When f and g are finite maps the map $f + g$, called f **"modified"** by g, is the finite map with domain $\mathrm{Dom}\, f \cup \mathrm{Dom}\, g$ and values

$$(f+g)(a) = \begin{cases} g(a) & \text{if } a \in \mathrm{Dom}\, g \\ f(a) & \text{otherwise.} \end{cases}$$

[Used in: 4.3.6]

4.2.5 REMARK: *Notation for modifying finite maps.*
In my opinion, the notation $f + g$ is unfortunate because addition is commutative[1], but $f + g \neq g + f$ when $\mathrm{Dom}\, f \cap \mathrm{Dom}\, g \neq \emptyset$. Although I dislike it, and would never use it in a million years in my own personal work, I am trying to adhere to the Definition's conventions.

I have seen other people use the notation $f \mathbin{<\!\!+} g$ (to indicate that g overwrites f, if there is any on $\mathrm{Dom}\, f \cap \mathrm{Dom}\, g \neq \emptyset$). MIZAR uses the notation $f + \cdot g$ (MIZAR code: "`f +* g`", in `FUNCT_4`) for this operation.

4.2.6 CLAUSE: *Compound objects for Core static semantics.*
The compound objects for the static semantics of the Core language are:

$$\begin{aligned} \tau \in \mathrm{Type} &= \mathrm{TyVar} \cup \mathrm{RowType} \cup \mathrm{FunType} \cup \mathrm{ConsType} \\ (\tau_1, \cdots, \tau_k) \text{ or } \tau^{(k)} &\in \mathrm{Type}^k \\ (\alpha_1, \cdots, \alpha_k) \text{ or } \alpha^{(k)} &\in \mathrm{TyVar}^k \\ \varrho \in \mathrm{RowType} &= \mathrm{Lab} \xrightarrow{\text{fin}} \mathrm{Type} \end{aligned}$$

[1]Except for ordinals, where $1 + \omega = \omega$ but $\omega + 1 = \omega \cup \{\omega\}$.

$$
\begin{aligned}
\tau \to \tau' \in \mathrm{FunType} &= \mathrm{Type} \times \mathrm{Type} \\
\mathrm{ConsType} &= \cup_{k \geq 0} \mathrm{ConsType}^{(k)} \\
\tau^{(k)} t \in \mathrm{ConsType}^{(k)} &= \mathrm{Type}^k \times \mathrm{TyName}^{(k)} \\
\theta \text{ or } \Lambda\alpha^{(k)}.\tau \in \mathrm{TypeFcn} &= \cup_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
\sigma \text{ or } \forall\alpha^{(k)}.\tau \in \mathrm{TypeScheme} &= \cup_{k \geq 0} \mathrm{TyVar}^k \times \mathrm{Type} \\
(\theta, \mathit{VE}) \in \mathrm{TyStr} &= \mathrm{TypeFcn} \times \mathrm{ValEnv} \\
\mathit{SE} \in \mathrm{StrEnv} &= \mathrm{StrId} \overset{\text{fin}}{\to} \mathrm{Env} \\
\mathit{TE} \in \mathrm{TyEnv} &= \mathrm{TyCon} \overset{\text{fin}}{\to} \mathrm{TyStr} \\
\mathit{VE} \in \mathrm{ValEnv} &= \mathrm{VId} \overset{\text{fin}}{\to} \mathrm{TypeScheme} \times \mathrm{IdStatus} \\
E \text{ or } (\mathit{SE}, \mathit{TE}, \mathit{VE}) \in \mathrm{Env} &= \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
T \in \mathrm{TyNameSet} &= \mathrm{Fin}(\mathrm{TyName}) \\
U \in \mathrm{TyVarSet} &= \mathrm{Fin}(\mathrm{TyVar}) \\
C \text{ or } T, U, E \in \mathrm{Context} &= \mathrm{TyNameSet} \times \mathrm{TyVarSet} \times \mathrm{Env}
\end{aligned}
$$

Note: the Definition takes $\cup$ to mean the *disjoint union* over semantic object classes. All the defined object classes are understood to be disjoint.

4.2.7 DEFINITION: *Free type names and free type variables.*
For any semantic object A, the Definition denotes by:
(1) "tynames A" the set of type names, and
(2) "tyvars A" the set of type variables occurring free in A.
Note that Λ and $\forall$ bind type variables.

4.2.8 COMMENT: *Universal quantifier.*
The universal quantifier $\forall$ has a similar meaning as used in System F, it appears. But what TAPL calls "type application" is handled implicitly rule (2).

4.2.9 DEFINITION: *Value identifier status.*
There are several things to note here:

A value environment VE maps value identifiers to a pair consisting of a type scheme and an identifier status. If $\mathit{VE}(\mathit{vid}) = (\sigma, \mathit{is})$, we say vid **"has status"** is in VE.

An occurrence of a value identifier which is elaborated in VE is referred to as:
(1) a **"Value Variable"** if its status in VE is `v`,
(2) a **"Value Constructor"** if its status in VE is `c`, and
(3) a **"Exception Constructor"** if its status in VE is `e`.

4.3. PROJECTION, INJECTION, AND MODIFICATION

4.3.1 DEFINITION: *Projection.*
If we have a tuple $x = (x_1, x_2, \ldots, x_n)$, then we will often find ourselves writing the projection as "x_i of x". That is to say, the Definition relies on metavariable names to indicate which component is selected.

For example, "the value environment component of E" is denoted "VE of E"

4.3.2 CONVENTION: *Projection of component that's finite map, applying argument.*
When a tuple contains a finite map, the Definition abuses notation and "applies" the tutple to an argument, again relying on the syntactic class of the argument to determine the relevant function.

For example, $C(\mathit{tycon})$ means $(\mathit{TE}$ of $C)\mathit{tycon}$, and $C(\mathit{vid})$ means $(\mathit{VE}$ of $(E$ of $C))(\mathit{vid})$.

4.3.3 CONVENTION: *Environments applied to long identifiers.*
The Definition extends this notational choice to allow environments to be applied to long identifiers. For example,$\mathit{longvid} = \mathit{strid}_1.\cdots.\mathit{strid}_k.\mathit{vid}$ then $E(\mathit{longvid})$ means

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

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

4.3.5 COMMENT: *Injections initialize all other components to empty set.*
I think what the Definition means to say is that an injection "x in y" amounts to a "constructor" of a new Y tuple whose components are all the empty set *except* the X factor.

4.3.6 CONVENTION: *Modification of environment.*
Extending the notation for modifying finite maps (§4.2.4), the Definition modifies an environment E by an environment E' constructing a new environment $E + E'$, preferring E' but resorting to E when necessary. (Or, overwriting any overlapping data shared by E and E' with E'.) This notation is extended to tuples in general.

Components are often left implicit in modification. For example, $E + \mathit{VE}$ means $E + (\{\}, \{\}, \mathit{VE})$.

For set components, the Definition means "union" for modification, so that $C + (T, \mathit{VE})$ means

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

(I, uh, don't know how to feel about this since it should be $\big(((T\text{ of }C) \setminus T) \cup T\big)$ instead of $(T\text{ of }C) \cup T$.)

4.3.7 DEFINITION: *Extending a tuple coherently.*
When we modify a context C by an environment E (or a type environment TE) while at the same time extending T of C to include the type names of E (resp., of TE), we do this by defining $C \oplus E$ (resp., $C \oplus \mathit{TE}$) to mean $C + (\mathrm{tynames}\,E, E)$ (resp., $C + (\mathrm{tynames}\,\mathit{TE}, \mathit{TE})$).

4.4. TYPES AND TYPE FUNTIONS

4.4.1 DEFINITION: *Equality type, admitting equality.*
A type τ is an **"Equality Type"** (or **"Admits Equality"**) if it is one of the following:
(1) α, where α admits equality;
(2) $\{\mathit{lab}_1 \mapsto \tau_1,\ \cdots,\ \mathit{lab}_n \mapsto \tau_n\}$, where each τ_i admits equality;
(3) $\tau^{(k)}t$, where t and all members of $\tau^{(k)}$ admit equality;
(4) (τ')`ref`.

4.4.2 CLAUSE: *Type functions.*
A type function $\theta = \Lambda\alpha^{(k)}.\tau$ has arity k; the bound variables must be distinct.

4.4.3 CLAUSE: *Equating two type functions.*
Two type functions are considered equal if they only differ in their choice of bound variables (alpha-conversion). In particular, the equality attribute has no significance in a bound variable of a type function. The Definitions gives as an example $\Lambda\alpha.\alpha \to \alpha$ and $\Lambda\beta.\beta \to \beta$ are equal type functions even if α admits equality but β does not.
[Used in: 4.5.3]

4.4.4 Clause: *Eta conversion for type functions.*
If t has arity k, then we write t to mean $\Lambda\alpha^{(k)}.\alpha^{(k)}t$ (eta-conversion); thus TyName $\subseteq$ TypeFcn.

4.4.5 Definition: *Equality type function, admits equality.*
We say $\theta = \Lambda\alpha^{(k)}.\tau$ is an **"Equality"** type function, or **"Admits Equality"**, if when the type variables $\alpha^{(k)}$ are chosen to admit equality then τ also admits equality.

4.4.6 Clause: *Application of type function, beta reduction.*
We write the application of a type function θ to a vector $\tau^{(k)}$ of types as $\tau^{(k)}\theta$. If $\theta = \Lambda\alpha^{(k)}.\tau$ we set $\tau^{(k)}\theta = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ (the Definition calls this "beta-conversion").

4.4.7 Convention: *Notation for substitution.*
We write $\tau\{\theta^{(k)}/t^{(k)}\}$ for the result of substituting type functions $\theta^{(k)}$ for type names $t^{(k)}$ in τ. We assume that all beta-conversions are carried out after substitution, so that for example

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

4.5. TYPE SCHEMES

Type schemes are not fully discussed or formalized explicitly in the Definition. It's just assumed you are familiar with what they mean by them. For the most part, I am relying heavily on Pottier and Rémy [PR05] to make sense of their usage in Standard ML.

4.5.1 Definition: *Generalizes a type.*
A type scheme $\sigma = \forall\alpha^{(k)}.\tau$ **"generalises"** a type τ', written $\sigma \succ \tau'$, if $\tau' = \tau\{\tau^{(k)}/\alpha^{(k)}\}$ for some $\tau^{(k)}$, where each member τ_i of $\tau^{(k)}$ admits equality if α_i does.

4.5.2 Definition: *Generalizes a scheme.*
A type scheme $\sigma = \forall\alpha^{(k)}.\tau$ **"generalizes"** $\sigma' = \forall\beta^{(\ell)}.\tau'$, written $\sigma \succ \sigma'$, if $\sigma \succ \tau'$ and $\beta^{(\ell)}$ contains no free type variable of σ.

It can be shown that $\sigma \succ \sigma'$ iff, for all τ'', whenever $\sigma' \succ \tau''$ then also $\sigma \succ \tau''$.

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

Note: unlike equality of type functions (§4.4.3), here the equality attribute *must be preserved* in renaming. For example $\forall\alpha.\alpha\to\alpha$ and $\forall\beta.\beta\to\beta$ are only equal if either both α and β admit equality, or neither does.

4.5.4 Clause: *Types considered as type schemes.*
The Definition considers a type τ to be a type scheme, identifying it with $\forall().\tau$.

4.5.5 Comment: *Type schemes should be thought of as "most general type"?*
I think that type schemes should be considered as the "most general type" for a given expression e. If that expression e is used in a situation which requires a "less general type" (i.e., e's most general type successfully generalizes the situation's required type), then it typechecks alright.

4.6. SCOPE OF EXPLICIT TYPE VARIABLES

4.6.1 Clause: *Type and datatype bindings explicitly introduce type variables.*
In the Core language, a type binding (*typbind*) or datatype binding (*datbind*) can explicitly introduce type variables whose scope is that binding.

4.6.2 Clause: *Value declarations have optional type variable bindings.*
In a value declaration `val` *tyvarseq* *valbind*, the sequence *tyvarseq* binds type variables: a type variable occurs free in `val` *tyvarseq* *valbind* iff it occurs free in *valbind* and is not in the sequence *tyvarseq*.

However, the Definition states, the explicit binding of type variables at `val` is optional, so the only remaining place in the Core language where type variables occur (which must be dealt with):
(1) the scope of an explicit type variable occurring in the "`:` *ty*" of a typed expression or pattern;
(2) the "`of` *ty*" of an exception binding.
The rest of this section, the Definition considers such free occurrences of type variables only.

4.6.3 Definition: *Scope of explicit type variables.*
Every occurrence of a value declaration is said to **"Scope"** a set of explicit type variables, determined as follows:
(1) a free type variable is unguarded, or
(2) a free type variable is implicitly scoped.

4.6.4 Definition: *Unguarded occurrence of type variable.*
A free occurrence of α in a value declaration `val` *tyvarseq* *valbind* is said to be **"Ungarded"** if the occurrence is not part of a smaller value declaration within *valbind*. In this case we say that α **"occurs unguarded"** in the value declaration.

4.6.5 Definition: *Implicitly scoped type variable.*
We say that α is **"Implicitly scoped"** at a particular declaration `val` *tyvarseq* *valbind* in a program if:
(1) α occurs unguarded in this value declaration, and
(2) α does not occur unguarded in any larger value declaration containing the given one.

4.6.6 Clause: *Assume every explicit type variable implicitly scoped at val.*
Henceforth, the Definition assumes that for every value declaration `val` *tyvarseq* $\cdots$ occurring in the program, every explicit type variable implicitly scoped at the `val` has been added to *tyvarseq* (subject to the syntactic constraint in Section 2.9).

4.6.7 Example: *Type variables with different scopes in value declarations.*
For example, in the two declarations

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

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

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

Then, according to the inference rules in Section 4.10 the first example can be elaborated, but the second cannot since `'a` is bound at the outer value declaration leaving no possibility of two different instantiations of the type of `id` in the application `id id`.

4.7. NON-EXPANSIVE EXPRESSIONS

4.7.1 DEFINITION: *Non-expansive expression in a given context.*
An expression is **"Non-expansive"** in context C if, after replacing infixed forms by their equivalent prefixed forms, and derived forms by their equivalent forms, it can be generated by the following grammar from the non-terminal *nexp*:

$$\begin{array}{lll} nexp & ::= & scon \\ & & \langle \texttt{op} \rangle longvid \\ & & \texttt{\{}\langle nexprow \rangle\texttt{\}} \\ & & \texttt{(}nexp\texttt{)} \\ & & conexp\ nexp \\ & & nexp\,\texttt{:}\,ty \\ & & \texttt{fn}\ match \end{array} \qquad \begin{array}{lll} nexprow & ::= & lab\ \texttt{=}\ nexp\langle \texttt{,}\ nexprow\rangle \\ & & \\ conexp & ::= & \texttt{(}conexp\langle \texttt{:}\,ty\rangle\texttt{)} \\ & & \langle \texttt{op}\rangle longvid \end{array}$$

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

All other expressions are said to be **"Expansive"** in C.

4.7.2 CLAUSE: *Non-expansive expressions are always 'pure'.*
The Definition informs us that the dynamic evaluation of a non-expansive expression will neither (a) generate an exception, nor (b) extend the domain of memory. But the evaluation of an expansive expression might.

4.7.3 COMMENT: *Non-expansive expressions used in closure of value declarations.*
We should note that the notion of "non-expansive expressions" only matters in the definition of the Closure of a value environment relative to a value binding declaration (§4.8.3). This occurs in let-expressions, which is the value restriction rule.

4.8. CLOSURE

4.8.1 DEFINITION: *Closure of a type with respect to a semantic object.*
Let τ be a type and A a semantic object (usually A is a context C). Then $\mathrm{Clos}_A(\tau)$, the **"closure"** of τ with respect to A, is the type scheme $\forall\alpha^{(k)}.\tau$, where $\alpha^{(k)} = \mathrm{tyvars}(\tau) \setminus \mathrm{tyvars}\,A$.

4.8.2 DEFINITION: *Total closure.*
The Definition abbreviates the **"total"** closure $\mathrm{Clos}_{\{\}}(\tau)$ to $\mathrm{Clos}(\tau)$.

4.8.3 CLAUSE: *Closure of value environment.*
If the range of a value environment VE contains only types (rather than arbitrary type schemes) we set

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

Recall the convention for constructing finite maps by set comprehension (§4.2.3), which is done here.

Closing a value environment VE that stems from the elaboration of a value binding *valbind* requires extra care to ensure type security of references and exceptions and correct scoping of explicit type variables. Recall (§2.9.9) that *valbind* is not allowed to bind the same variable twice. Thus, for each $vid \in \mathrm{Dom}\,VE$ there is a unique *pat* = *exp* in *valbind* which binds *vid*. If $VE(vid) = (\tau, is)$, let $\mathrm{Clos}_{C,valbind}\,VE(vid) = (\forall\alpha^{(k)}.\tau, is)$, where

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

[Used in: 4.7.3, 4.10.20]

4.9. TYPE STRUCTURES AND TYPE ENVIRONMENTS

4.9.1 DEFINITION: *Well-formed type structures.*
The definition calls a type structure (θ, VE) **"Well-Formed"** if either $\mathit{VE} = \{\}$ or θ is a type name t. (The latter case arises, with $\mathit{VE} \neq \{\}$, in `datatype` declarations.)

4.9.2 DEFINITION: *Well-formed semantic objects.*
An object or "assembly"[2] A of semantic objects is **"Well-Formed"** if every type structure occurring in A is well-formed.

4.9.3 DEFINITION: *Type structure respects equality.*
A type structure (t, VE) is said to **"Respect Equality"** if, whenever t admits equality, then either $t =$ `ref` (see Appendix C) or, for each $\mathit{VE}(\mathit{vid})$ of the form $(\forall\alpha^{(k)}.(\tau \to \alpha^{(k)}t), \mathit{is})$, the type function $\Lambda\alpha^{(k)}.\tau$ also admits equality.

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

4.9.4 DEFINITION: *Type environment respecting equality.*
A type environment TE **"Respects Equality"** if all its type structures do so.

4.9.5 DEFINITION: *Type environment maximizes equality.*
Let TE be a type environment, and let T be the set of type names t such that (t, VE) occurs in TE for some $\mathit{VE} \neq \{\}$. Then TE is said to **"Maximise Equality"** if

(1) TE respects equality, and also
(2) if any larger subset of T were to admit equality (without any change in the equality attribute of any type names not in T), then TE would cease to respect equality.

4.9.6 DEFINITION: *Environment for abstract type declarations.*
For any TE of the form

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

where no VE_i is the empty map, and for any E we define $\mathrm{Abs}(\mathit{TE}, E)$ to be the environment obtained from E and TE as follows (just three easy steps):

(1) Let $\mathrm{Abs}(\mathit{TE})$ be the type environment $\{\mathit{tycon}_i \mapsto (t_i, \{\})\ ;\ 1 \leq i \leq k\}$ in which all value environments VE_i have been replaced by the empty map.
(2) Let $t'_1, \cdots, t'_k$ be new distinct type names none of which admit equality.
(3) Then $\mathrm{Abs}(\mathit{TE}, E)$ is the result of simultaneously substituting t'_i for t_i, $1 \leq i \leq k$, throughout $\mathrm{Abs}(\mathit{TE}) + E$. (The effect of the latter substitution is to ensure that the use of equality on an `abstype` is restricted to the `with` part.)

4.9.7 COMMENT: *Abs(TE, E) used for abstype declarations only.*
Note that this last definition for $\mathrm{Abs}(\mathit{TE}, E)$ is used only in Rule (19) of the Definition for `abstype` declarations.

[2] Yeah, this is the first time I think the word "assembly" has been used in the Definition, and I'm as confused by its appearance as you probably are... But they will show up again extensively in the chapter on static semantics of modules.

4.10. INFERENCE RULES

4.10.1 Clause: *Judgement form.*
Each rule of the semantics allows inferences among "sentences" [judgements] of the form

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

where A is usually a context, *phrase* is a phrase of the Core language, and A' is a semantic object (usually a type or an environment). The Definition suggests reading this as "*phrase* elaborates to A' in (context) A."

Some rules have extra hypotheses not of this form, which are called **"Side Conditions"**.

4.10.2 Convention: *First and second options.*
The Definition presents the rules with optional phrases within single angle brackets $\langle\ \rangle$ which are called **"First Options"**. There are also optional phrases within double angle brackets $\langle\langle\ \rangle\rangle$ which are called **"Second Options"**. Their usage is governed by 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.

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

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

4.10.4 Comment: *Equation numbers match Definition's rule numbers.*
Again, for the sake of consistency, the number for the rule found in the Definition is written here as its Equation number.

4.10.5 Comment: *Comments swept into the rule.*
Note that I have relocated the "comments" found in the Definition to appear in the relevant rule, when it simplifies the presentation.

Atomic Expressions

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

4.10.6 Rule: *Special constants.*

$$\frac{}{C \vdash \textit{scon} \Rightarrow \mathrm{type}(\textit{scon})} \tag{1}$$

4.10.7 Rule: *Value variable.*
Note this simplified three rules (for value variables, value constructors, and exception constants) in the 1990 Definition into a single rule. The second premise, $\sigma \succ \tau$, is new to the 1997 Definition.

$$\frac{C(\textit{longvar}) = (\sigma, \textit{is}) \qquad \sigma \succ \tau}{C \vdash \textit{longvar} \Rightarrow \tau} \tag{2}$$

The instantiation of type schemes allows different occurrences of a single *longvid* to assume different types. Note that the identifier status is not used in this rule.

4.10.8 Rule: *Record expressions.*

$$\frac{\langle C \vdash \mathit{exprow} \Rightarrow \varrho \rangle}{C \vdash \texttt{\{} \ \langle \mathit{exprow} \rangle \ \texttt{\}} \Rightarrow \{\}\langle + \ \varrho \rangle \text{ in Type}} \tag{3}$$

4.10.9 Rule: *Let expressions.*

$$\frac{C \vdash \mathit{dec} \Rightarrow E \qquad C \oplus E \vdash \mathit{exp} \Rightarrow \tau \qquad \text{tynames}\,\tau \subseteq T \text{ of } C}{C \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow \tau} \tag{4}$$

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. The side condition prevents type names generated by *dec* from escaping outside the local declaration.

4.10.10 Rule: *Parenthetical expression.*

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

Expression Rows

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

4.10.11 Rule: *Expression rows.*

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

Expressions

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

4.10.12 Rule: *Atomic.*

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

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

4.10.13 Rule: *Application.*

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

4.10.14 Rule: *Typed.*

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

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

4.10.15 Rule: *Handle exception.*

$$\frac{C \vdash \mathit{exp} \Rightarrow \tau \qquad C \vdash \mathit{match} \Rightarrow \texttt{exn} \to \tau}{C \vdash \mathit{exp}\ \texttt{handle}\ \mathit{match} \Rightarrow \tau} \tag{10}$$

4.10.16 Rule: *Raise exception.*

$$\frac{C \vdash \mathit{exp} \Rightarrow \texttt{exn}}{C \vdash \texttt{raise}\ \mathit{exp} \Rightarrow \tau} \tag{11}$$

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

4.10.17 Rule: *Function.*

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

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

4.10.18 Rule: *Match.*

$$\frac{C \vdash \mathit{mrule} \Rightarrow \tau \qquad \langle\langle C \vdash \mathit{match} \Rightarrow \tau\rangle}{C \vdash \mathit{mrule}\ \langle\ \texttt{|}\ \mathit{match}\rangle \Rightarrow \tau} \tag{13}$$

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

4.10.19 Rule: *Mrule.*

$$\frac{C \vdash \mathit{pat} \Rightarrow (\mathit{VE}, \tau) \qquad C + \mathit{VE} \vdash \mathit{exp} \Rightarrow \tau' \qquad \text{tynames}\ \mathit{VE} \subseteq T \text{ of } C}{C \vdash \mathit{pat}\ \texttt{=>}\ \mathit{exp}\ \Rightarrow \tau \rightarrow \tau'} \tag{14}$$

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

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

4.10.20 Rule: *Value Declaration.*

$$\frac{\begin{array}{l} U = \text{tyvars}(\mathit{tyvarseq}) \\ C + U \vdash \mathit{valbind} \Rightarrow \mathit{VE} \qquad \mathit{VE}' = \text{Clos}_{C,\mathit{valbind}}\, \mathit{VE} \qquad U \cap \text{tyvars}\ \mathit{VE}' = \emptyset \end{array}}{C \vdash \texttt{val}\ \mathit{tyvarseq}\ \mathit{valbind} \Rightarrow \mathit{VE}' \text{ in Env}} \tag{15}$$

Here *VE* will contain types rather than general type schemes. The closure of *VE* allows value identifiers to be used polymorphically, via Rule (2).

The side-condition on U ensures that the type variables in *tyvarseq* are bound by the closure operation, if they occur free in the range of *VE*.

On the other hand, if the phrase `val` *tyvarseq valbind* occurs inside some larger value binding `val` $\mathit{tyvarseq}'$ $\mathit{valbind}'$ then no type variable α listed in $\mathit{tyvarseq}'$ will become bound by the $\text{Clos}_{C,\mathit{valbind}}\,\mathit{VE}$ operation; for α must be in U of C and hence excluded from closure by the definition of the closure operation (Section 4.8, (§4.8.3), page 33) since U of $C \subseteq \text{tyvars}\, C$.

4.10.21 Rule: *Type declaration.*

$$\frac{C \vdash \mathit{typbind} \Rightarrow \mathit{TE}}{C \vdash \texttt{type}\ \mathit{typbind} \Rightarrow \mathit{TE} \text{ in Env}} \tag{16}$$

4.10.22 Rule: *Datatype declaration.*

$$\frac{\begin{array}{c} C \oplus \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \\ \forall (t, \mathit{VE}') \in \text{Ran}\, \mathit{TE},\ t \notin (T \text{ of } C) \\ \mathit{TE} \text{ maximises equality} \end{array}}{C \vdash \texttt{datatype}\ \mathit{datbind} \Rightarrow (\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{17}$$

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

4.10.23 Rule: *Datatype replication.*

$$\frac{C(\mathit{longtycon}) = (\theta,\, \mathit{VE}) \qquad \mathit{TE} = \{\mathit{tycon} \mapsto (\theta,\, \mathit{VE})\}}{C \vdash \texttt{datatype}\; \mathit{tycon}\; \texttt{-=-}\; \texttt{datatype}\; \mathit{longtycon} \Rightarrow (\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{18}$$

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

4.10.24 Rule: *Abstype declaration.*

$$\frac{\begin{array}{c} C \oplus \mathit{TE} \vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \\ \forall (t,\, \mathit{VE}') \in \mathrm{Ran}\, \mathit{TE},\ t \notin (T \text{ of } C) \\ C \oplus (\mathit{VE}, \mathit{TE}) \vdash \mathit{dec} \Rightarrow E \\ \mathit{TE} \text{ maximises equality} \end{array}}{C \vdash \texttt{abstype}\; \mathit{datbind}\; \texttt{with}\; \mathit{dec}\; \texttt{end} \Rightarrow \mathrm{Abs}(\mathit{TE}, E)} \tag{19}$$

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

4.10.25 Rule: *Exception declaration.*

$$\frac{C \vdash \mathit{exbind} \Rightarrow \mathit{VE}}{C \vdash \texttt{exception}\; \mathit{exbind} \Rightarrow \mathit{VE} \text{ in Env}} \tag{20}$$

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

4.10.26 Rule: *Local declaration.*

$$\frac{C \vdash \mathit{dec}_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash \mathit{dec}_2 \Rightarrow E_2}{C \vdash \texttt{local}\; \mathit{dec}_1\; \texttt{in}\; \mathit{dec}_2\; \texttt{end} \Rightarrow E_2} \tag{21}$$

4.10.27 Rule: *Open declaration.*

$$\frac{C(\mathit{longstrid}_1) = E_1 \qquad \cdots \qquad C(\mathit{longstrid}_n) = E_n}{C \vdash \texttt{open}\; \mathit{longstrid}_1 \cdots \mathit{longstrid}_n \Rightarrow E_1 + \cdots + E_n} \tag{22}$$

4.10.28 Rule: *Empty declaration.*

$$\frac{}{C \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{23}$$

4.10.29 Rule: *Sequential declaration.*

$$\frac{C \vdash \mathit{dec}_1 \Rightarrow E_1 \qquad C \oplus E_1 \vdash \mathit{dec}_2 \Rightarrow E_2}{C \vdash \mathit{dec}_1\ \langle\texttt{;}\rangle\ \mathit{dec}_2 \Rightarrow E_1 + E_2} \tag{24}$$

Value Bindings

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

4.10.30 RULE: *Value binding.*

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C \vdash exp \Rightarrow \tau \qquad \langle C \vdash valbind \Rightarrow VE' \rangle}{C \vdash pat \texttt{ = } exp \ \langle \texttt{and}\ valbind \rangle \Rightarrow VE\ \langle +\ VE' \rangle} \tag{25}$$

When the option is present we have $\mathrm{Dom}\, VE \cap \mathrm{Dom}\, VE' = \emptyset$ by the syntactic restrictions (§§2.9.2, 2.9.9).

4.10.31 RULE: *Recursive value binding.*

$$\frac{C + VE \vdash valbind \Rightarrow VE \qquad \text{tynames}\ VE \subseteq T \text{ of } C}{C \vdash \texttt{rec}\ valbind \Rightarrow VE} \tag{26}$$

Modifying C by VE on the left captures the recursive nature of the binding. From Rule (25) 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 assigned the same type. Also note that $C + VE$ may overwrite identifier status. For example, the program `datatype t = f; val rec f = fn x => x;` is legal.

Type Bindings

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

4.10.32 RULE: *Type binding.*

$$\frac{tyvarseq = \alpha^{(k)} \qquad C \vdash ty \Rightarrow \tau \qquad \langle C \vdash typbind \Rightarrow TE \rangle}{C \vdash tyvarseq\ tycon \texttt{ = } ty\ \langle \texttt{and}\ typbind \rangle \Rightarrow \{tycon \mapsto (\Lambda\alpha^{(k)}.\tau, \{\})\}\ \langle + TE \rangle} \tag{27}$$

The syntactic restrictions (§§2.9.2–2.9.3) ensure that the type function $\Lambda\alpha^{(k)}.\tau$ satisfies the well-formedness constraint of Section 4.4 and they ensure $tycon \notin \mathrm{Dom}\, TE$.

4.10.33 REMARK: *Type operators enter here.*
This is the only way for type operators to appear, it's through type binding rules.

Datatypes introduce new "primitive notions", as it were: a "type name" which acts like a tagged pair. This resembles a type operator, but does not expand the "tag" part at all. The Definition makes a distinction between type operators ["type functions"] and "type names", but this seems more like a distinction without a difference.

Datatype Bindings

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

4.10.34 RULE: *Datatype binding.*

$$\frac{\begin{array}{c} tyvarseq = \alpha^{(k)} \\ C, \alpha^{(k)}t \vdash conbind \Rightarrow VE \\ \text{arity}\, t = k \\ \langle C \vdash datbind' \Rightarrow VE', TE' \qquad \forall (t', VE'') \in \mathrm{Ran}\, TE', t \neq t' \rangle \end{array}}{\begin{array}{l} C \vdash tyvarseq\ tycon \texttt{ = } conbind\ \langle \texttt{and}\ datbind' \rangle \Rightarrow \\ \qquad (\mathrm{Clos}\, VE \langle +\ VE' \rangle,\ \{tycon \mapsto (t, \mathrm{Clos}\, VE)\}\ \langle + TE' \rangle) \end{array}} \tag{28}$$

The syntactic restrictions (§2.9.2) ensure $\mathrm{Dom}\, VE \cap \mathrm{Dom}\, VE' = \emptyset$ and $tycon \notin \mathrm{Dom}\, TE'$.

NOTE: there is a typo in the Definition, namely there is no closing right parentheses in the conclusion to this rule.

Constructor Bindings

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

4.10.35 Rule: *Data constructors.*

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash \mathit{conbind} \Rightarrow \mathit{VE} \rangle\rangle}{\begin{array}{c} C, \tau \vdash \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle\langle\ \texttt{|}\ \mathit{conbind} \rangle\rangle \Rightarrow \\ \{\mathit{vid} \mapsto (\tau, \mathtt{c})\}\ \langle +\ \{\mathit{vid} \mapsto (\tau' \to \tau, \mathtt{c})\}\ \rangle\ \langle\langle +\ \mathit{VE} \rangle\rangle \end{array}} \tag{29}$$

By the syntactic restrictions (§2.9.2) $\mathit{vid} \notin \mathrm{Dom}\ \mathit{VE}$.

Exception Bindings

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

4.10.36 Rule: *Exception binding.*

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau \rangle \qquad \langle\langle C \vdash \mathit{exbind} \Rightarrow \mathit{VE} \rangle\rangle}{\begin{array}{c} C \vdash \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle\langle \texttt{and}\ \mathit{exbind} \rangle\rangle \Rightarrow \\ \{\mathit{vid} \mapsto (\mathtt{exn}, \mathtt{e})\}\ \langle +\ \{\mathit{vid} \mapsto (\tau \to \mathtt{exn}, \mathtt{e})\}\ \rangle\ \langle\langle +\ \mathit{VE} \rangle\rangle \end{array}} \tag{30}$$

Comments:
(1) Notice that τ may contain type variables.
(2) For each C and *exbind*, there is at most one VE satisfying $C \vdash \mathit{exbind} \Rightarrow \mathit{VE}$.

4.10.37 Rule: *Exception binding.*

$$\frac{C(\mathit{longvid}) = (\tau, \mathtt{e}) \qquad \langle C \vdash \mathit{exbind} \Rightarrow \mathit{VE} \rangle}{C \vdash \mathit{vid}\ \texttt{=}\ \mathit{longvid}\ \langle \texttt{and}\ \mathit{exbind} \rangle \Rightarrow \{\mathit{vid} \mapsto (\tau, \mathtt{e})\}\ \langle +\ \mathit{VE} \rangle} \tag{31}$$

For each C and *exbind*, there is at most one VE satisfying $C \vdash \mathit{exbind} \Rightarrow \mathit{VE}$.

Atomic Patterns

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

4.10.38 Rule: *Wildcard pattern.*

$$\frac{}{C \vdash \_ \Rightarrow (\{\}, \tau)} \tag{32}$$

4.10.39 Rule: *Special constant in pattern.*

$$\frac{}{C \vdash \mathit{scon} \Rightarrow (\{\}, \mathrm{type}(\mathit{scon}))} \tag{33}$$

4.10.40 Rule: *Variable pattern.*

$$\frac{\mathit{vid} \notin \mathrm{Dom}(C) \text{ or } \mathit{is} \text{ of } C(\mathit{vid}) = \mathtt{v}}{C \vdash \mathit{vid} \Rightarrow (\{\mathit{vid} \mapsto (\tau, \mathtt{v})\}, \tau)} \tag{34}$$

The context C determines whether to apply this rule or the next rule. Note that *vid* can assume a type, not a general type scheme.

4.10.41 Rule: *Constant pattern.*

$$\frac{C(\mathit{longvid}) = (\sigma, \mathit{is}) \qquad \mathit{is} \neq \mathtt{v} \qquad \sigma \succ \tau^{(k)}t}{C \vdash \mathit{longvid} \Rightarrow (\{\}, \tau^{(k)}t)} \tag{35}$$

The context C determines whether to apply this rule or the previous rule.

4.10.42 Rule: *Record pattern.*

$$\frac{\langle C \vdash \mathit{patrow} \Rightarrow (\mathit{VE}, \varrho) \rangle}{C \vdash \texttt{\{}\ \langle \mathit{patrow} \rangle\ \texttt{\}} \Rightarrow (\ \{\}\langle +\ \mathit{VE} \rangle,\ \{\}\langle +\ \varrho \rangle \text{ in Type }\ )} \tag{36}$$

4.10.43 RULE: *Parenthesised pattern.*

$$\frac{C \vdash pat \Rightarrow (VE, \tau)}{C \vdash \texttt{(}\ pat\ \texttt{)} \Rightarrow (VE, \tau)} \tag{37}$$

Pattern Rows

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

4.10.44 RULE: *Wildcard record.*

$$\frac{}{C \vdash \texttt{...} \Rightarrow (\{\}, \varrho)} \tag{38}$$

4.10.45 RULE: *Record component.*

$$\frac{\begin{array}{c} C \vdash pat \Rightarrow (VE, \tau) \\ \langle C \vdash patrow \Rightarrow (VE', \varrho) \qquad \mathrm{Dom}\ VE \cap \mathrm{Dom}\ VE' = \emptyset\rangle \end{array}}{C \vdash lab\ \texttt{=}\ pat\ \langle\ \texttt{,}\ patrow\rangle \Rightarrow (VE\langle +\ VE'\rangle,\ \{lab \mapsto \tau\}\langle + \varrho\rangle)} \tag{39}$$

The syntactic restrictions (§2.9.1) ensure $lab \notin \mathrm{Dom}\ \varrho$.

Patterns

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

4.10.46 RULE: *Atomic pattern.*

$$\frac{C \vdash atpat \Rightarrow (VE, \tau)}{C \vdash atpat \Rightarrow (VE, \tau)} \tag{40}$$

4.10.47 RULE: *Construction pattern.*

$$\frac{C(longvid) = (\sigma, is) \qquad is \neq \mathtt{v} \qquad \sigma \succ \tau' \to \tau \qquad C \vdash atpat \Rightarrow (VE, \tau')}{C \vdash longvid\ atpat \Rightarrow (VE, \tau)} \tag{41}$$

4.10.48 RULE: *Typed pattern.*

$$\frac{C \vdash pat \Rightarrow (VE, \tau) \qquad C \vdash ty \Rightarrow \tau}{C \vdash pat\ \texttt{:}\ ty \Rightarrow (VE, \tau)} \tag{42}$$

4.10.49 RULE: *Layered pattern.*

$$\frac{\begin{array}{c} vid \notin \mathrm{Dom}(C) \text{ or } is \text{ of } C(vid) = \mathtt{v} \\ \langle C \vdash ty \Rightarrow \tau\rangle \qquad C \vdash pat \Rightarrow (VE, \tau) \qquad vid \notin \mathrm{Dom}\ VE \end{array}}{C \vdash vid\langle\texttt{:}\ ty\rangle\ \texttt{as}\ pat \Rightarrow (\{vid \mapsto (\tau, \mathtt{v})\} + VE, \tau)} \tag{43}$$

Type Expressions

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

4.10.50 RULE: *Type variable.*
The name used in the LaTeX comments for the Definition call this the "atype variable" rule — I'm uncertain why "atype" is used (atomic type? Or because α in Standard ML is `'a` as a type variable?)

$$\frac{tyvar = \alpha}{C \vdash tyvar \Rightarrow \alpha} \tag{44}$$

4.10.51 RULE: *Record type.*

$$\frac{\langle C \vdash tyrow \Rightarrow \varrho\rangle}{C \vdash \texttt{\{}\ \langle tyrow\rangle\ \texttt{\}} \Rightarrow \{\}\langle + \varrho\rangle \text{ in Type}} \tag{45}$$

4.10.52 RULE: *Constructed type.*

$$\frac{\mathit{tyseq} = \mathit{ty}_1 \cdots \mathit{ty}_k \qquad C \vdash \mathit{ty}_i \Rightarrow \tau_i \ (1 \leq i \leq k) \\ C(\mathit{longtycon}) = (\theta, \mathit{VE})}{C \vdash \mathit{tyseq}\ \mathit{longtycon} \Rightarrow \tau^{(k)}\theta} \tag{46}$$

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

4.10.53 RULE: *Function type.*

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau \qquad C \vdash \mathit{ty}' \Rightarrow \tau'}{C \vdash \mathit{ty}\ \texttt{->}\ \mathit{ty}' \Rightarrow \tau \to \tau'} \tag{47}$$

4.10.54 RULE: *Parenthesised type.*

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau}{C \vdash \texttt{(}\ \mathit{ty}\ \texttt{)} \Rightarrow \tau} \tag{48}$$

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

4.10.55 RULE: *Record type components.*

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau \qquad \langle C \vdash \mathit{tyrow} \Rightarrow \varrho \rangle}{C \vdash \mathit{lab}\ \texttt{:}\ \mathit{ty}\ \langle\ \texttt{,}\ \mathit{tyrow}\rangle \Rightarrow \{\mathit{lab} \mapsto \tau\}\langle +\ \varrho\rangle} \tag{49}$$

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

4.11. FURTHER RESTRICTIONS

4.11.1 COMMENT: *Additional impositions.*
In addition to the inference rules given above, there are three restrictions the Definition imposes. These all concern pattern matching, which would be tedious to write down using inference rules (and I, at least, would find them confusingly misplaced among typing rules).

4.11.2 CLAUSE: *Record pattern must determine uniquely the set of labels.*
For each occurrence of a record pattern containing a record wildcard (i.e., of the form `{`lab_1`=`pat_1`,`$\cdots$`,`lab_m`=`pat_m`,...}`) the program context must determine uniquely the domain $\{\mathit{lab}_1, \cdots, \mathit{lab}_n\}$ of its row type, where $m \leq n$; thus, the context must determine the labels $\{\mathit{lab}_{m+1}, \cdots, \mathit{lab}_n\}$ of the fields to be matched by the wildcard. For this purpose, an explicit type constraint may be needed.

4.11.3 CLAUSE: *Irredundant patterns.*
In a match of the form pat_1 `=>` exp_1 `|` $\cdots$ `|` pat_n `=>` exp_n the pattern sequence $\mathit{pat}_1, \ldots, \mathit{pat}_n$ should be *irredundant*; that is, each pat_j must match some value (of the right type) which is not matched by pat_i for any $i < j$.

In the context `fn` *match*, the *match* must also be *exhaustive*; that is, every value (of the right type) must be matched by some pat_i.

The compiler must give warning on violation of these restrictions, but should still compile the match.

The restrictions are inherited by derived forms; in particular, this means that in the function-value binding *vid* $\mathit{atpat}_1 \cdots \mathit{atpat}_n\langle\texttt{:}\ \mathit{ty}\rangle$ `=` *exp* (consisting of one clause only), each separate atpat_i should be exhaustive by itself.

4.11.4 CLAUSE: *Warn about nonexhaustive patterns, but still compile.*
For each value binding *pat* `=` *exp* the compiler must issue a report (but still compile) if *pat* is not exhaustive. This will detect a mistaken declaration like `val nil =` *exp* in which the user expects to declare a new variable `nil` (whereas

the language dictates that `nil` is here a constant pattern, so no variable gets declared). However, this warning should not be given when the binding is a component of a top-level declaration `val` *valbind*; e.g. `val x::l =` exp_1 `and y =` exp_2 is not faulted by the compiler at top level, but may of course generate a `Bind` exception (see Section 6.5).

[Used in: 6.5.2]

CHAPTER 5

STATIC SEMANTICS FOR MODULES

5.1. SEMANTIC OBJECTS

5.1.1 Clause: *Simple semantic objects.*
The simple semantic objects for Modules static semantics are exactly those for the Core from Section 4.1.

5.1.2 Definition: *Compound semantic objects.*
The compound objects are those for Core (Section 4.2), augmented by the following:

$$\begin{aligned}
\Sigma \text{ or } (T)E &\in \mathrm{Sig} = \mathrm{TyNameSet} \times \mathrm{Env} \\
\Phi \text{ or } (T)(E,(T')E') &\in \mathrm{FunSig} = \mathrm{TyNameSet} \times (\mathrm{Env} \times \mathrm{Sig}) \\
G &\in \mathrm{SigEnv} = \mathrm{SigId} \overset{\mathrm{fin}}{\to} \mathrm{Sig} \\
F &\in \mathrm{FunEnv} = \mathrm{FunId} \overset{\mathrm{fin}}{\to} \mathrm{FunSig} \\
B \text{ or } T,F,G,E &\in \mathrm{Basis} = \mathrm{TyNameSet} \times \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env}
\end{aligned}$$

The prefix (T), in signatures and functor signatures, binds type names.
[Used in: 8.0.5]

5.1.3 Clause: *Projection, injection, modification.*
These operations are analogous 'in the obvious way' to how they were defined for Core's static semantics.

5.1.4 Definition: *Context of basis.*
The Definition introduces the notation $C \,\mathrm{of}\, B$ to be the context $(T \,\mathrm{of}\, B, \emptyset, E \,\mathrm{of}\, B)$, i.e. with an empty set of explicit type variables.

5.1.5 Definition: *Modifying a basis by an environment.*
We frequently need to modify a basis B by an environment E (or a structure environment SE say), at the same time extending T of B to include the type names. We therefore define $B \oplus SE$, for example, to mean $B + (\mathrm{tynames}\, SE, SE)$.

5.1.6 Clause: *Structures not represented by semantic objects.*
There is no separate kind of semantic object to represent structures: structure expressions elaborate to environments, just as structure-level declarations do. Thus, notions which are commonly associated with structures (for example the notion of matching a structure against a signature) are defined in terms of environments.

5.2. TYPE REALISATION

5.2.1 Definition: *Type realisation, or simply just a "realisation".*
We define a **"Type Realisation"** is a map $\varphi : \mathrm{TyName} \to \mathrm{TypeFcn}$ such that t and $\varphi(t)$ have the same arity, and if t admits equality then so does $\varphi(t)$.

5.2.2 Definition: *Support of a type realisation.*
The **"Support"** $\mathrm{Supp}\,\varphi$ of a type realisation φ is the set of type names t for which $\varphi(t) \neq t$.

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

5.2.4 Clause: *Extend realisations to all semantic objects.*
Realisations φ are extended to apply to all semantic objects; their effect is to replace each name t by $\varphi(t)$. In applying φ to an object with bound names, such as a signature $(T)E$, first bound names must be changed so that, for each binding prefix (T),

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

5.3. SIGNATURE INSTANTIATION

5.3.1 Definition:
An environment E_2 is an **"Instance"** of a signature $\Sigma_1 = (T_1)E_1$, written $\Sigma_1 \geq E_2$, if there exists a realisation φ such that $\varphi(E_1) = E_2$ and $\mathrm{Supp}\,\varphi \subseteq T_1$.

5.4. FUNCTOR SIGNATURE INSTANTIATION

5.4.1 Definition:
A pair $(E, (T')E')$ is called a **"Functor Instance"**. Given $\Phi = (T_1)(E_1, (T_1')E_1')$, a functor instance $(E_2, (T_2')E_2')$ is an **"Instance"** of Φ, written $\Phi \geq (E_2, (T_2')E_2')$, if there exists a realisation φ such that $\varphi(E_1, (T_1')E_1') = (E_2, (T_2')E_2')$ and $\mathrm{Supp}\,\varphi \subseteq T_1$.

5.5. ENRICHMENT

5.5.1 Clause:
In matching an environment to a signature, the environment will be allowed both to have more components, and to be more polymorphic, than (an instance of) the signature. More precisely, we define enrichment of environments and type structures recursively as follows:

5.5.2 Definition: *Environment enrichment.*
An environment $E_1 = (SE_1, TE_1, VE_1)$ **"Enriches"** another environment $E_2 = (SE_2, TE_2, VE_2)$, written $E_1 \succ E_2$, if
(1) $\mathrm{Dom}\,SE_1 \supseteq \mathrm{Dom}\,SE_2$, and $SE_1(strid) \succ SE_2(strid)$ for all $strid \in \mathrm{Dom}\,SE_2$
(2) $\mathrm{Dom}\,TE_1 \supseteq \mathrm{Dom}\,TE_2$, and $TE_1(tycon) \succ TE_2(tycon)$ for all $tycon \in \mathrm{Dom}\,TE_2$
(3) $\mathrm{Dom}\,VE_1 \supseteq \mathrm{Dom}\,VE_2$, and $VE_1(vid) \succ VE_2(vid)$ for all $vid \in \mathrm{Dom}\,VE_2$, where $(\sigma_1, is_1) \succ (\sigma_2, is_2)$ means $\sigma_1 \succ \sigma_2$ and

$$is_1 = is_2 \quad \text{or} \quad is_2 = \mathrm{v}$$

5.5.3 Definition: *Type structure enrichment.*
Finally, a type structure (θ_1, VE_1) **"Enriches"** another type structure (θ_2, VE_2), written $(\theta_1, VE_1) \succ (\theta_2, VE_2)$, if
(1) $\theta_1 = \theta_2$
(2) Either $VE_1 = VE_2$ or $VE_2 = \{\}$.

5.6. SIGNATURE MATCHING

5.6.1 DEFINITION: *Environment matches a signature.*
An environment E **"Matches"** a signature Σ_1 if there exists an environemnt E^- such that $\Sigma_1 \geq E^- \prec E$. Thus matching is a combination of instantiation and enrichment. There is at most one such E^-, given Σ_1 and E.

5.7. INFERENCE RULES

5.7.1 CLAUSE: *Judgement form for static semantics of Modules.*
The rules of the Modules static semantics allow sentences of the form

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

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.

5.7.2 CLAUSE: *Assumption about basis for top-level declarations.*
Although not assumed in our definitions, it is intended that every basis $B = T, F, G, E$ in which a *topdec* is elaborated has the property that $\mathrm{tynames}\, F \cup \mathrm{tynames}\, G \cup \mathrm{tynames}\, E \subseteq T$.

5.7.3 THEOREM: *Static semantics preserves the assumption.*
Let S be an inferred sentence $B \vdash topdec \Rightarrow B'$ in which $B = T, F, G, E$ satisfies the condition $\mathrm{tynames}\, F \cup \mathrm{tynames}\, G \cup \mathrm{tynames}\, E \subseteq T$. Then B' also satisfies the condition.

Moreover, if S' is a sentence of the form $B'' \vdash phrase \Rightarrow A$ occurring in a proof of S, where phrase is any Modules phrase, then B'' also satisfies the condition.

Finally, if $T, U, E \vdash phrase \Rightarrow A$ occurs in a proof of S, where phrase is a phrase of Modules or of the Core, then $\mathrm{tynames}\, E \subseteq T$.

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

5.7.4 RULE: *Generative structure expression.*

$$\frac{B \vdash strdec \Rightarrow E}{B \vdash \texttt{struct}\ strdec\ \texttt{end} \Rightarrow E} \qquad (50)$$

5.7.5 RULE: *Long structure identifier.*

$$\frac{B(longstrid) = S}{B \vdash longstrid \Rightarrow S} \qquad (51)$$

5.7.6 RULE: *Transparent constraint rule.*

$$\frac{B \vdash strexp \Rightarrow E \qquad B \vdash sigexp \Rightarrow \Sigma \qquad \Sigma \geq E' \prec E}{B \vdash strexp : sigexp \Rightarrow E'} \qquad (52)$$

5.7.7 RULE: *Opaque constraint rule.*

$$\frac{\begin{array}{c} B \vdash strexp \Rightarrow E \\ B \vdash sigexp \Rightarrow (T')E' \\ (T')E' \geq E'' \prec E \\ T' \cap (T \text{ of } B) = \emptyset \end{array}}{B \vdash strexp\, \texttt{:>} sigexp \Rightarrow E'} \qquad (53)$$

5.7.8 Rule: *Functor application.*

$$\frac{\begin{array}{c} B \vdash \mathit{strexp} \Rightarrow E \\ B(\mathit{funid}) \geq (E'', (T')E')\ ,\ E \succ E'' \\ (\mathrm{tynames}\, E\ \cup\ T \text{ of } B) \cap T' = \emptyset \end{array}}{B \vdash \mathit{funid}\ \texttt{(}\ \mathit{strexp}\ \texttt{)}\ \Rightarrow E'} \tag{54}$$

The side condition $(\mathrm{tynames}\, E \cup T \text{ of } B) \cap T' = \emptyset$ can always be satisfied by renaming bound names in $(T')E'$; it ensures that the generated datatypes receive new names.

Let $B(\mathit{funid}) = (T)(E_f, (T')E'_f)$. Let φ be a realisation such that $\varphi(E_f, (T')E'_f) = (E'', (T')E')$. Sharing between argument and result specified in the declaration of the functor *funid* is represented by the occurrence of the same name in both E_f and E'_f, and this repeated occurrence is preserved by φ, yielding sharing between the argument structure E and the result structure E' of this functor application.

5.7.9 Rule: *Let-expression in structures.*

$$\frac{B \vdash \mathit{strdec} \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strexp} \Rightarrow E_2}{B \vdash \texttt{let}\ \mathit{strdec}\ \texttt{in}\ \mathit{strexp}\ \texttt{end} \Rightarrow E_2} \tag{55}$$

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

Structure-level Declarations

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

5.7.10 Rule: *Core declaration.*

$$\frac{C \text{ of } B \vdash \mathit{dec} \Rightarrow E}{B \vdash \mathit{dec} \Rightarrow E} \tag{56}$$

5.7.11 Rule: *Structure declaration.*

$$\frac{B \vdash \mathit{strbind} \Rightarrow \mathit{SE}}{B \vdash \texttt{structure}\ \mathit{strbind} \Rightarrow \mathit{SE} \text{ in Env}} \tag{57}$$

5.7.12 Rule: *Local structure-level declaration.*

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2} \tag{58}$$

5.7.13 Rule: *Empty declaration.*

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{59}$$

5.7.14 Rule: *Sequential declaration.*

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \mathit{strdec}_1\ \langle\texttt{;}\rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2} \tag{60}$$

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

5.7.15 Rule: *Structure binding.*

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \qquad \langle B + \mathrm{tynames}\, E \vdash \mathit{strbind} \Rightarrow \mathit{SE}\rangle}{B \vdash \mathit{strid}\ \texttt{=}\ \mathit{strexp}\ \langle \texttt{and}\ \mathit{strbind}\rangle \Rightarrow \{\mathit{strid} \mapsto E\}\ \langle + \mathit{SE}\rangle} \tag{61}$$

Signature Expressions $\boxed{B \vdash \mathit{sigexp} \Rightarrow E}$

5.7.16 Rule: *Encapsulation signature expression.*

$$\frac{B \vdash \mathit{spec} \Rightarrow E}{B \vdash \texttt{sig}\ \mathit{spec}\ \texttt{end} \Rightarrow E} \tag{62}$$

5.7.17 Rule: *Signature identifier.*

$$\frac{B(\mathit{sigid}) = (T)E \qquad T \cap (T \text{ of } B) = \emptyset}{B \vdash \mathit{sigid} \Rightarrow E} \tag{63}$$

The bound names of $B(\mathit{sigid})$ can always be renamed to satisfy $T \cap (T \text{ of } B) = \emptyset$, if necessary.

5.7.18 Rule: *"Where type".*
Note this rule has $3 + 2 + 3 = 8$ premises.

$$\frac{\begin{array}{c} B \vdash \mathit{sigexp} \Rightarrow E \qquad \mathit{tyvarseq} = \alpha^{(k)} \qquad C \text{ of } B \vdash \mathit{ty} \Rightarrow \tau \\ E(\mathit{longtycon}) = (t, \mathit{VE}) \qquad t \notin T \text{ of } B \\ \varphi = \{t \mapsto \Lambda\alpha^{(k)}.\tau\} \qquad \Lambda\alpha^{(k)}.\tau \text{ admits equality, if } t \text{ does} \qquad \varphi(E) \text{ well-formed} \end{array}}{B \vdash \mathit{sigexp}\ \texttt{where type}\ \mathit{tyvarseq}\ \mathit{longtycon}\ \texttt{=}\ \mathit{ty} \Rightarrow \varphi(E)} \tag{64}$$

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

The Definition has apparently introduced a new subsection consisting of a single rule, but has not named the subsection. This was particularly confusing to me until I read the LaTeX source for the Definition.

5.7.19 Rule: *Topmost signature expression.*

$$\frac{B \vdash \mathit{sigexp} \Rightarrow E \qquad T = \mathrm{tynames}\, E \setminus (T \text{ of } B)}{B \vdash \mathit{sigexp} \Rightarrow (T)E} \tag{65}$$

A signature expression *sigexp* which is an immediate constituent of a signature binding, a signature constraint, or a functor binding is elaborated to a signature, see rules 52, 53, 67, and 86.

Signature Declarations

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

5.7.20 RULE: *Single signature declaration.*

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

Signature Bindings

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

5.7.21 RULE: *Signature binding.*

$$\frac{B \vdash \mathit{sigexp} \Rightarrow \Sigma \qquad \langle B \vdash \mathit{sigbind} \Rightarrow G \rangle}{B \vdash \mathit{sigid}\ \texttt{=}\ \mathit{sigexp}\ \langle \texttt{and}\ \mathit{sigbind} \rangle \Rightarrow \{\mathit{sigid} \mapsto \Sigma\}\ \langle + \ G \rangle} \tag{67}$$

Specifications

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

5.7.22 RULE: *Value specification.*

$$\frac{C \text{ of } B \vdash \mathit{valdesc} \Rightarrow \mathit{VE}}{B \vdash \texttt{val}\ \mathit{valdesc} \Rightarrow \text{Clos}\,\mathit{VE} \text{ in Env}} \tag{68}$$

VE is determined by B and *valdesc*.

5.7.23 RULE: *Type specification.*

$$\frac{C \text{ of } B \vdash \mathit{typdesc} \Rightarrow \mathit{TE} \qquad \forall (t,\ \mathit{VE}) \in \text{Ran}\,\mathit{TE},\ t \text{ does not admit equality}}{B \vdash \texttt{type}\ \mathit{typdesc} \Rightarrow \mathit{TE} \text{ in Env}} \tag{69}$$

The typenames in TE are new.

5.7.24 RULE: *Equality type specification.*

$$\frac{C \text{ of } B \vdash \mathit{typdesc} \Rightarrow \mathit{TE} \qquad \forall (t,\ \mathit{VE}) \in \text{Ran}\,\mathit{TE},\ t \text{ admits equality}}{B \vdash \texttt{eqtype}\ \mathit{typdesc} \Rightarrow \mathit{TE} \text{ in Env}} \tag{70}$$

The typenames in TE are new.

5.7.25 RULE: *Datatype specification.*

$$\frac{\begin{array}{c}\forall (t,\ \mathit{VE}') \in \text{Ran}\,\mathit{TE}, t \notin T \text{ of } B \\ C \text{ of } B \oplus \mathit{TE} \vdash \mathit{datdesc} \Rightarrow \mathit{VE}, \mathit{TE} \\ \mathit{TE} \text{ maximises equality}\end{array}}{B \vdash \texttt{datatype}\ \mathit{datdesc} \Rightarrow (\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{71}$$

The typenames in TE are new.

5.7.26 RULE: *Datatype replication specification.*

$$\frac{B(\mathit{longtycon}) = (\theta,\ \mathit{VE}) \qquad \mathit{TE} = \{\mathit{tycon} \mapsto (\theta,\ \mathit{VE})\}}{B \vdash \texttt{datatype}\,\mathit{tycon}\ \texttt{-=-}\ \texttt{datatype}\,\mathit{longtycon} \Rightarrow (\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{72}$$

5.7.27 RULE: *Exception specification.*

$$\frac{C \text{ of } B \vdash \mathit{exdesc} \Rightarrow \mathit{VE}}{B \vdash \texttt{exception}\ \mathit{exdesc} \Rightarrow \mathit{VE} \text{ in Env}} \tag{73}$$

VE is determined by B and *exdesc* and contains monotypes only.

5.7.28 Rule: *Structure specification.*

$$\frac{B \vdash \mathit{strdesc} \Rightarrow \mathit{SE}}{B \vdash \texttt{structure}\ \mathit{strdesc} \Rightarrow \mathit{SE} \text{ in Env}} \tag{74}$$

5.7.29 Rule: *Include signature specification.*

$$\frac{B \vdash \mathit{sigexp} \Rightarrow E}{B \vdash \texttt{include}\ \mathit{sigexp} \Rightarrow E} \tag{75}$$

5.7.30 Rule: *Empty specification.*

$$\frac{}{B \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{76}$$

5.7.31 Rule: *Sequential specification.*

$$\frac{B \vdash \mathit{spec}_1 \Rightarrow E_1 \qquad B \oplus E_1 \vdash \mathit{spec}_2 \Rightarrow E_2 \qquad \mathrm{Dom}(E_1) \cap \mathrm{Dom}(E_2) = \emptyset}{B \vdash \mathit{spec}_1\ \langle\texttt{;}\rangle\ \mathit{spec}_2 \Rightarrow E_1 + E_2} \tag{77}$$

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

5.7.32 Rule: *Sharing type specification.*
Note this has $2 + 2 + 2 = 6$ premises.

$$\frac{\begin{array}{cc} B \vdash \mathit{spec} \Rightarrow E & E(\mathit{longtycon}_i) = (t_i, \mathit{VE}_i),\ i = 1..n \\ t \in \{t_1, \ldots, t_n\} & t \text{ admits equality, if some } t_i \text{ does} \\ \{t_1, \ldots, t_n\} \cap T \text{ of } B = \emptyset & \varphi = \{t_1 \mapsto t, \ldots, t_n \mapsto t\} \end{array}}{B \vdash \mathit{spec}\ \texttt{sharing type}\ \mathit{longtycon}_1\ \texttt{=} \cdots \texttt{=}\ \mathit{longtycon}_n \Rightarrow \varphi(E)} \tag{78}$$

Value Descriptions

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

5.7.33 Rule: *Value description.*

$$\frac{C \vdash \mathit{ty} \Rightarrow \tau \qquad \langle C \vdash \mathit{valdesc} \Rightarrow \mathit{VE}\rangle}{C \vdash \mathit{vid}\ \texttt{:}\ \mathit{ty}\ \langle\texttt{and}\ \mathit{valdesc}\rangle \Rightarrow \{\mathit{vid} \mapsto (\tau, \mathtt{v})\}\ \langle + \mathit{VE}\rangle} \tag{79}$$

Type Descriptions

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

5.7.34 Rule: *Type description.*

$$\frac{\begin{array}{c} \langle C \vdash \mathit{typdesc} \Rightarrow \mathit{TE} \qquad t \notin \mathrm{tynames}\,\mathit{TE}\rangle \\ \mathit{tyvarseq} = \alpha^{(k)} \quad t \notin T \text{ of } C \quad \mathrm{arity}\, t = k \end{array}}{C \vdash \mathit{tyvarseq}\ \mathit{tycon}\ \langle\texttt{and}\ \mathit{typdesc}\rangle \Rightarrow \{\mathit{tycon} \mapsto (t, \{\})\}\ \langle + \mathit{TE}\rangle} \tag{80}$$

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

Datatype Descriptions

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

5.7.35 Rule: *Datatype description.*
There are $2 + 3 = 5$ premises to this rule.

$$\frac{\begin{array}{c}\langle C \vdash \mathit{datdesc}' \Rightarrow \mathit{VE}', \mathit{TE}' \qquad \forall (t', \mathit{VE}'') \in \mathrm{Ran}\, \mathit{TE}',\, t \neq t' \rangle \\ \mathit{tyvarseq} = \alpha^{(k)} \qquad C, \alpha^{(k)} t \vdash \mathit{condesc} \Rightarrow \mathit{VE} \quad \mathrm{arity}\, t = k\end{array}}{\begin{array}{l}C \vdash \mathit{tyvarseq}\ \mathit{tycon} = \mathit{condesc}\ \langle \mathtt{and}\ \mathit{datdesc}' \rangle \Rightarrow \\ \qquad \mathrm{Clos}\, \mathit{VE} \langle +\ \mathit{VE}' \rangle,\ \{\mathit{tycon} \mapsto (t, \mathrm{Clos}\, \mathit{VE})\}\ \langle +\ \mathit{TE}' \rangle\end{array}} \tag{81}$$

Constructor Descriptions

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

5.7.36 Rule:

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau' \rangle \qquad \langle\langle C, \tau \vdash \mathit{condesc} \Rightarrow \mathit{VE} \rangle\rangle}{\begin{array}{l}C, \tau \vdash \mathit{vid}\ \langle \mathtt{of}\ \mathit{ty} \rangle\ \langle\langle\ \mathtt{|}\ \mathit{condesc} \rangle\rangle \Rightarrow \\ \qquad \{\mathit{vid} \mapsto (\tau, \mathtt{c})\}\ \langle +\ \{\mathit{vid} \mapsto (\tau' \to \tau, \mathtt{c})\}\ \rangle\ \langle\langle +\ \mathit{VE} \rangle\rangle\end{array}} \tag{82}$$

Exception Descriptions

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

5.7.37 Rule: *Exception description.*

$$\frac{\langle C \vdash \mathit{ty} \Rightarrow \tau \qquad \mathrm{tyvars}(\tau) = \emptyset \rangle \qquad \langle\langle C \vdash \mathit{exdesc} \Rightarrow \mathit{VE} \rangle\rangle}{\begin{array}{l}C \vdash \mathit{vid}\ \langle \mathtt{of}\ \mathit{ty} \rangle\ \ \langle\langle \mathtt{and}\ \mathit{exdesc} \rangle\rangle \Rightarrow \\ \qquad \{\mathit{vid} \mapsto (\mathtt{exn}, \mathtt{e})\}\ \langle +\ \{\mathit{vid} \mapsto (\tau \to \mathtt{exn}, \mathtt{e})\} \rangle\ \langle\langle +\ \mathit{VE} \rangle\rangle\end{array}} \tag{83}$$

Structure Descriptions

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

5.7.38 Rule: *Structure description.*

$$\frac{B \vdash \mathit{sigexp} \Rightarrow E \qquad \langle B + \mathrm{tynames}\, E \vdash \mathit{strdesc} \Rightarrow \mathit{SE} \rangle}{B \vdash \mathit{strid}\ \mathtt{:}\ \mathit{sigexp}\ \langle \mathtt{and}\ \mathit{strdesc} \rangle \Rightarrow \{\mathit{strid} \mapsto E\}\ \langle +\ \mathit{SE} \rangle} \tag{84}$$

Functor Declarations

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

5.7.39 Rule: *Single functor declaration.*

$$\frac{B \vdash \mathit{funbind} \Rightarrow F}{B \vdash \mathtt{functor}\ \mathit{funbind} \Rightarrow F} \tag{85}$$

Functor Bindings

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

5.7.40 Rule: *Functor binding.*

$$\frac{\begin{array}{c}B \vdash \mathit{sigexp} \Rightarrow (T)E \qquad B \oplus \{\mathit{strid} \mapsto E\} \vdash \mathit{strexp} \Rightarrow E' \\ T \cap (T\ \mathrm{of}\ B) = \emptyset \quad T' = \mathrm{tynames}\, E' \setminus ((T\ \mathrm{of}\ B) \cup T) \\ \langle B \vdash \mathit{funbind} \Rightarrow F \rangle\end{array}}{\begin{array}{l}B \vdash \mathit{funid}\ \mathtt{(}\ \mathit{strid}\ \mathtt{:}\ \mathit{sigexp}\ \mathtt{)}\ \mathtt{=}\ \mathit{strexp}\ \langle \mathtt{and}\ \mathit{funbind} \rangle \Rightarrow \\ \qquad\qquad \{\mathit{funid} \mapsto (T)(E, (T')E')\}\ \langle\langle +\ F \rangle\rangle\end{array}} \tag{86}$$

Since $\oplus$ is used, any type name t in E acts like a constant in the functor body; in particular, it ensures that further names generated during elaboration of the body are distinct from t. The set T' is chosen such that every name free in $(T)E$ or $(T)(E, (T')E')$ is free in B.

Top-level Declarations

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

5.7.41 Rule: *Structure-level declaration.*

$$\frac{\begin{array}{c} B \vdash \mathit{strdec} \Rightarrow E \qquad \langle B \oplus E \vdash \mathit{topdec} \Rightarrow B' \rangle \\ B'' = (\mathrm{tynames}\, E, E) \text{ in Basis } \langle + B' \rangle \qquad \mathrm{tyvars}\, B'' = \emptyset \end{array}}{B \vdash \mathit{strdec} \quad \langle \mathit{topdec} \rangle \Rightarrow B''} \tag{87}$$

No free type variables enter the basis: if $B \vdash \mathit{topdec} \Rightarrow B'$ then $\mathrm{tyvars}(B') = \emptyset$.

5.7.42 Rule: *Signature Declaration.*

$$\frac{\begin{array}{c} B \vdash \mathit{sigdec} \Rightarrow G \qquad \langle B \oplus G \vdash \mathit{topdec} \Rightarrow B' \rangle \\ B'' = (\mathrm{tynames}\, G, G) \text{ in Basis } \langle + B' \rangle \end{array}}{B \vdash \mathit{sigdec} \quad \langle \mathit{topdec} \rangle \Rightarrow B''} \tag{88}$$

No free type variables enter the basis: if $B \vdash \mathit{topdec} \Rightarrow B'$ then $\mathrm{tyvars}(B') = \emptyset$.

5.7.43 Rule: *Functor declaration.*

$$\frac{\begin{array}{c} B \vdash \mathit{fundec} \Rightarrow F \qquad \langle B \oplus F \vdash \mathit{topdec} \Rightarrow B' \rangle \\ B'' = (\mathrm{tynames}\, F, F) \text{ in Basis } \langle + B' \rangle \qquad \mathrm{tyvars}\, B'' = \emptyset \end{array}}{B \vdash \mathit{fundec} \quad \langle \mathit{topdec} \rangle \Rightarrow B''} \tag{89}$$

No free type variables enter the basis: if $B \vdash \mathit{topdec} \Rightarrow B'$ then $\mathrm{tyvars}(B') = \emptyset$.

CHAPTER 6

DYNAMIC SEMANTICS FOR THE CORE

6.1. REDUCED SYNTAX

6.1.1 Clause: *Core syntax reduced.*
Since types are mostly dealt with in the static semantics, the Core syntax is reduced by the following transformations, for the purpose of the dynamic semantics:
(1) All explicit type ascriptions "`:` *ty*" are omitted, and qualifications "`of` *ty*" are omitted from constructor and exception bindings.
(2) The Core phrase classes Ty and TyRow are omitted.

6.2. SIMPLE OBJECTS

6.2.1 Definition: *Classes of simple objects.*
All objects in the dynamic semantics are built from the identifier classes together with the simple object classes listed below (note that basic values are discussed in their own section):

a	$\in$	Addr	addresses
en	$\in$	ExName	exception names
b	$\in$	BasVal	basic values
sv	$\in$	SVal	special values
		{FAIL}	failure

6.2.2 Clause: *Infinite set of addresses.*
Addr forms an infinite set.

6.2.3 Clause: *Infinite set of exception names.*
ExName forms an infinite set.

6.2.4 Clause: *Special values are values of special constants.*
SVal is the class of values denoted by the special constants SCon. Each integer, word, or real constant denotes a value according to normal mathematical conventions; each string or character constant denotes a sequence of characters as explained in Section 2.2.

6.2.5 Definition: *Value of a special constant.*
The value denoted by *scon* is written val(*scon*).

6.2.6 Clause: *Failure "value".*
FAIL is the result of a failing attempt to match a value and a pattern. Thus FAIL is neither a value nor an exception, but simply a semantic object used in the rules to express operationally how matching proceeds.

6.2.7 Clause: *Exception constructors evaluate to names.*
Exception constructors evaluate to exception names. This accomodates the generative nature of exception bindings: each evaluation of a declaration of an exception constructor binds it to a new unique name.

6.3. COMPOUND OBJECTS

6.3.1 DEFINITION: *Compound objects.*
The compound objects for dynamic semantics are listed thus:

$$\begin{array}{rcl}
v & \in & \mathrm{Val} = \{\texttt{:=}\} \cup \mathrm{SVal} \cup \mathrm{BasVal} \cup \mathrm{VId} \\
 & & \quad \cup\, (\mathrm{VId} \times \mathrm{Val}) \cup \mathrm{ExVal} \\
 & & \quad \cup\, \mathrm{Record} \cup \mathrm{Addr} \cup \mathrm{FcnClosure} \\
r & \in & \mathrm{Record} = \mathrm{Lab} \overset{\mathrm{fin}}{\to} \mathrm{Val} \\
e & \in & \mathrm{ExVal} = \mathrm{ExName} \cup (\mathrm{ExName} \times \mathrm{Val}) \\
[e] \text{ or } p & \in & \mathrm{Pack} = \mathrm{ExVal} \\
(\mathit{match}, E, \mathit{VE}) & \in & \mathrm{FcnClosure} = \mathrm{Match} \times \mathrm{Env} \times \mathrm{ValEnv} \\
\mathit{mem} & \in & \mathrm{Mem} = \mathrm{Addr} \overset{\mathrm{fin}}{\to} \mathrm{Val} \\
\mathit{ens} & \in & \mathrm{ExNameSet} = \mathrm{Fin}(\mathrm{ExName}) \\
(\mathit{mem}, \mathit{ens}) \text{ or } s & \in & \mathrm{State} = \mathrm{Mem} \times \mathrm{ExNameSet} \\
(\mathit{SE}, \mathit{TE}, \mathit{VE} \text{ or } E & \in & \mathrm{Env} = \mathrm{StrEnv} \times \mathrm{TyEnv} \times \mathrm{ValEnv} \\
\mathit{SE} & \in & \mathrm{StrEnv} = \mathrm{StrId} \overset{\mathrm{fin}}{\to} \mathrm{Env} \\
\mathit{TE} & \in & \mathrm{TyEnv} = \mathrm{TyCon} \overset{\mathrm{fin}}{\to} \mathrm{ValEnv} \\
\mathit{VE} & \in & \mathrm{ValEnv} = \mathrm{VId} \overset{\mathrm{fin}}{\to} \mathrm{Val} \times \mathrm{IdStatus}
\end{array}$$

6.3.2 CONVENTION: *Projection, injection, modification.*
The notions of projection, injection, and modification are adapted from the static semantics.

6.3.3 CLAUSE: *Object classes disjoint.*
The object classes thus defined are all disjoint, so there is no ambiguity in taking unions as disjoint unions.

6.3.4 CLAUSE: *Exception values and packets.*
Observe that ExVal and Pack (exception values and packets) are isomorphic classes, but the latter class corresponds to exceptions which have been raised, and therefore has different semantic significance from the former, which is just a subclass of values.

6.3.5 COMMENT: *Caution about overloaded notation.*
Although the same names (e.g., E for an environment) are used as in the static semantics, the objects denoted are different. This need cause no confusion since the static and dynamic semantics are presented separately.

6.4. BASIC VALUES

6.4.1 CLAUSE: *Simplified set of basic values.*
The Definition takes BasVal to be the singleton set $\{\texttt{=}\}$, but allows for a larger set of basic values defined by libraries.

6.4.2 CLAUSE: *APPLY gives meaning to basic values.*
The semantical meaning of basic values is represented by a function

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

which satisfies $\mathrm{APPLY}(\texttt{=}, \{1 \mapsto v_1, 2 \mapsto v_2\})$ is `true` if v_1 and v_2 are identical values, and `false` otherwise.

6.4.3 COMMENT: *Problems with APPLY.*
As Rossberg [Ros18b, §6] and Kahrs [Kah93, §§11.1–3] note, the APPLY function has no access to program state. This implies the library primitives may not be stateful, and moreover that a lot of interesting primitives (e.g., arrays,

I/O streams, etc.) could not be added to the language without extending the Definition itself.

But Rossberg observes, any nontrivial library type requires an extension of the Definition of values or state anyways (and equality types — for, e.g., `array`). The Definition should have contained a comment regarding this.

6.5. BASIC EXCEPTIONS

6.5.1 CLAUSE: *Initial basis.*
A subset BasExName $\subset$ ExName of the exception names are bound to predefined exception constructors in the initial dynamic basis. These names are denoted by the identifiers to which they are bound in the initial basis, and are as follows:

`Match   Bind`

These exceptions are raised upon failure of pattern-matching in evaluating a function or a *valbind*, as detailed inthe rules that follow.

6.5.2 CLAUSE: *Match raised by nonexhaustive patterns.*
The exception `Match` can only be raised for a match which is not exhaustive, and has therefore been flagged by the compiler (§4.11.4).

6.6. FUNCTION CLOSURE

6.6.1 COMMENT: *Closures are "function values".*
Intuitively, a function closure is precisely the "value" for a function expression. This is critical when trying to use, e.g., equational reasoning for Standard ML.

6.6.2 CLAUSE: *Informal explanation of closures.*
A function closure $(match, E, VE)$ may be informally described as follows: when the function closure is applied to a value v, *match* will be evaluated against v, in the environment E modified in a special sense by VE (to allow for recursive functions). This happens by evaluating *match* in $E + \mathrm{Rec}\ VE$, which will be defined shortly.

6.6.3 DEFINITION: *Rec operator on value environments.*
The function

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

is defined as follows:
(1) $\mathrm{Dom}(\mathrm{Rec}\ VE) = \mathrm{Dom}\ VE$
(2) If $VE(vid) \notin \mathrm{FcnClosure} \times \{\mathtt{v}\}$, then $(\mathrm{Rec}\ VE)(vid) = VE(vid)$
(3) If $VE(vid) = ((match', E', VE'), \mathtt{v})$ then $(\mathrm{Rec}\ VE)(vid) = ((match', E', VE), \mathtt{v})$

6.6.4 COMMENT: Rec *VE "unrolls" function closures once.*
The effect of this definition is that — before the application of the closure $(match, E, VE)$ to v — the function closures in Ran VE [not a type: Ran VE is correct, see next example] are "unrolled" once, to prepare for their possible recursive application during the evaluation of *match* upon v.

This ensures all semantic objects are finite (by controlling the unrolling of recursion).

6.6.5 EXAMPLE: *Example unrolling recursive function calls.*
It makes more sense to see this rule in action. I'm adapting the example from the Commentary [MT91, pp.1–12]. Consider the example code:

```
datatype NAT = Zero | Succ of NAT;

fun twice(Zero) = Zero
  | twice(Succ x) = Succ(Succ(twice(x)));

twice(Zero);
```

The desugared code "`twice = fn` *match*" will evaluate to

$$VE = \{\texttt{twice} \mapsto (\mathit{match}, E_1, \{\})\}.$$

The problem is that, looking at the VE appearing in the closure bound to `twice`: it's precisely $VE_{\texttt{twice}} = \{\}$! So trying to lookup the identifier `twice` for a recursive function call will result in a failure.

However, we don't look up `twice` in $VE_{\texttt{twice}} = \{\}$ but instead we're using

$$\mathrm{Rec}\ VE = \{\texttt{twice} \mapsto (\mathit{match}, E_1, VE)\}.$$

This allows for proper recursive function calls.

6.6.6 Comment: *Used only twice.*
The Rec operator is used only twice: in Rules (102) [the rule for evaluating an application expression "*exp atexp*"], and (126) [governing recursive value bindings of the form "`rec` *valbind*"].

6.7. INFERENCE RULES

6.7.1 Definition: *Judgement form.*
The semantic rules govern judgemental forms like

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

where A is usually an environment, A' is some semantic object, and here s, s' are states before and after the evaluation which occurs in the judgement.

6.7.2 Definition: *Side-conditions.*
Some hypotheses in the rules are not of the form we just described. They are called **"Side-Conditions"**.

6.7.3 Clause: *Convention for options.*
The convention for options is the same as for the Core static semantics.

6.7.4 Convention: *State convention.*
In most cases, the state can be omitted from the inference rule. When omitted, the conventions for restoring them as is follows: If the rule is presented in the form

$$\frac{A_1 \vdash \mathit{phrase}_1 \Rightarrow A'_1 \qquad A_2 \vdash \mathit{phrase}_2 \Rightarrow A'_2 \quad \cdots \quad \cdots \quad A_n \vdash \mathit{phrase}_n \Rightarrow A'_n}{A \vdash \mathit{phrase} \Rightarrow A'}$$

then the full form is intended to be

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

(Any side-conditions are left unaltered).

Thus the left-to-right order of the hypotheses indicates the order of evaluation. Note that in the case $n = 0$, when there are no hypotheses (except possibly side-conditions), we have $s_n = s_0$; this implies that the rule causes no side effect.

The Definition calls this convention the **"State Convention"**, and it must be applied to each version of a rule obtained by inclusion or omission of its options.
[Used in: 7.3.4]

6.7.5 CONVENTION: *Exception convention.*
The **"Exception Convention"** is adopted to deal with the propagation of exception packets p. For each rule whose full form (ignoring side-conditions) is

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

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

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

where p' does not occur in the original rule. [The only exception is rule 104, describing how exception handlers work.] 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.
[Used in: 7.3.4]

6.7.6 CONVENTION: *Compound variables.*
The last convention allows for "compound variables" (built from the metavariables and a slash "/") to range over unions of semantic objects.

For instance the compound variable v/p ranges over $\mathrm{Val} \cup \mathrm{Pack}$. We also allow x/FAIL to range over $X \cup \{\mathrm{FAIL}\}$ where x ranges over X; furthermore, we extend environment modification to allow for failure as follows:

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

Atomic Expressions

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

6.7.7 RULE: *Special constant.*

$$\frac{}{E \vdash \mathit{scon} \Rightarrow \mathrm{val}(\mathit{scon})} \tag{90}$$

6.7.8 RULE: *Value variable.*

$$\frac{E(\mathit{longvid}) = (v, \mathit{is})}{E \vdash \mathit{longvid} \Rightarrow v} \tag{91}$$

As in the static semantics, value identifiers are looked up in the environment and the identifier status is not used.

6.7.9 RULE: *Record expression.*

$$\frac{\langle E \vdash \mathit{exprow} \Rightarrow r \rangle}{E \vdash \texttt{\{}\ \langle \mathit{exprow} \rangle\ \texttt{\}} \Rightarrow \{\}\langle + r \rangle \text{ in Val}} \tag{92}$$

6.7.10 RULE: *Let expressions.*

$$\frac{E \vdash \mathit{dec} \Rightarrow E' \qquad E + E' \vdash \mathit{exp} \Rightarrow v}{E \vdash \texttt{let}\ \mathit{dec}\ \texttt{in}\ \mathit{exp}\ \texttt{end} \Rightarrow v} \tag{93}$$

6.7.11 RULE: *Parenthesised expression.*

$$\frac{E \vdash \mathit{exp} \Rightarrow v}{E \vdash \texttt{(}\ \mathit{exp}\ \texttt{)} \Rightarrow v} \tag{94}$$

Expression Rows

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

6.7.12 RULE: *Labelled expressions.*

$$\frac{E \vdash exp \Rightarrow v \qquad \langle E \vdash exprow \Rightarrow r \rangle}{E \vdash lab \texttt{ = } exp \ \langle \ \texttt{,} \ exprow \rangle \Rightarrow \{lab \mapsto v\} \langle + \ r \rangle} \tag{95}$$

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

Expressions

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

6.7.13 RULE: *Atomic expressions.*

$$\frac{E \vdash atexp \Rightarrow v}{E \vdash atexp \Rightarrow v} \tag{96}$$

6.7.14 RULE: *Constructor application.*

$$\frac{E \vdash exp \Rightarrow vid \qquad vid \neq \texttt{ref} \qquad E \vdash atexp \Rightarrow v}{E \vdash exp \ atexp \Rightarrow (vid, v)} \tag{97}$$

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.

6.7.15 RULE: *Exception constructor application.*

$$\frac{E \vdash exp \Rightarrow en \qquad E \vdash atexp \Rightarrow v}{E \vdash exp \ atexp \Rightarrow (en, v)} \tag{98}$$

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.

6.7.16 RULE: *Reference application.*

$$\frac{s, E \vdash exp \Rightarrow \ \texttt{ref} \ , s' \qquad s', E \vdash atexp \Rightarrow v, s'' \qquad a \notin \mathrm{Dom}(mem \text{ of } s'')}{s, E \vdash exp \ atexp \Rightarrow a, \ s'' + \{a \mapsto v\}} \tag{99}$$

(1) The side condition ensures that a new address is chosen. There are no rules concerning disposal of inaccessible addresses.
(2) 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.

6.7.17 RULE: *Assignment application.*

$$\frac{s, E \vdash exp \Rightarrow \ \texttt{:=} \ , s' \qquad s', E \vdash atexp \Rightarrow \{\texttt{1} \mapsto a, \ \texttt{2} \mapsto v\}, s''}{s, E \vdash exp \ atexp \Rightarrow \{\} \text{ in Val}, \ s'' + \{a \mapsto v\}} \tag{100}$$

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.

6.7.18 RULE: *Basic function application.*

$$\frac{E \vdash exp \Rightarrow b \qquad E \vdash atexp \Rightarrow v \qquad \mathrm{APPLY}(b, v) = v'/p}{E \vdash exp\ atexp \Rightarrow v'/p} \tag{101}$$

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.

6.7.19 RULE: *Closure application.*

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\ VE,\ v \vdash match \Rightarrow v' \end{array}}{E \vdash exp\ atexp \Rightarrow v'} \tag{102}$$

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.

6.7.20 RULE: *Failing closure application.*

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\ VE,\ v \vdash match \Rightarrow \mathrm{FAIL} \end{array}}{E \vdash exp\ atexp \Rightarrow [\texttt{Match}]} \tag{103}$$

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.

6.7.21 RULE: *Unexceptional handler.*

$$\frac{E \vdash exp \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \tag{104}$$

This is the only rule to which the exception convention does not apply. If the operator evaluates to a packet then rule 105 or rule 106 must be used.

6.7.22 RULE: *Handled exception.*

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \tag{105}$$

6.7.23 RULE: *Unhandled exception.*

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow \mathrm{FAIL}}{E \vdash exp\ \texttt{handle}\ match \Rightarrow [e]} \tag{106}$$

Packets that are not handled by the *match* propagate.

6.7.24 RULE: *Raise exception.*

$$\frac{E \vdash exp \Rightarrow e}{E \vdash \texttt{raise}\ exp \Rightarrow [e]} \tag{107}$$

6.7.25 RULE: *Function.*
This evaluates a function expression to a function value.

$$\frac{}{E \vdash \texttt{fn}\ match \Rightarrow (match, E, \{\})} \tag{108}$$

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

Matches $\boxed{E, v \vdash \mathit{match} \Rightarrow v'/p/\mathrm{FAIL}}$

6.7.26 Rule: *Match 1.*
This intuitively formalizes "Go with the first match possible" semantics.

$$\frac{E, v \vdash \mathit{mrule} \Rightarrow v'}{E, v \vdash \mathit{mrule} \ \langle \ \texttt{|} \ \mathit{match}\rangle \Rightarrow v'} \tag{109}$$

6.7.27 Rule: *Match 2.*

$$\frac{E, v \vdash \mathit{mrule} \Rightarrow \mathrm{FAIL}}{E, v \vdash \mathit{mrule} \Rightarrow \mathrm{FAIL}} \tag{110}$$

6.7.28 Rule: *Match 3.*
Given a leading *mrule* which fails to match, but there is a rule in the rest of the *match* which either matches or fails. . . we go with that instead.

$$\frac{E, v \vdash \mathit{mrule} \Rightarrow \mathrm{FAIL} \qquad E, v \vdash \mathit{match} \Rightarrow v'/\mathrm{FAIL}}{E, v \vdash \mathit{mrule} \ \texttt{|} \ \mathit{match} \Rightarrow v'/\mathrm{FAIL}} \tag{111}$$

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

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

6.7.29 Rule: *Match a clause successfully.*

$$\frac{E, v \vdash \mathit{pat} \Rightarrow \mathit{VE} \qquad E + \mathit{VE} \vdash \mathit{exp} \Rightarrow v'}{E, v \vdash \mathit{pat} \ \texttt{=>} \ \mathit{exp} \ \ \Rightarrow v'} \tag{112}$$

6.7.30 Rule: *Failing to match a pattern.*

$$\frac{E, v \vdash \mathit{pat} \Rightarrow \mathrm{FAIL}}{E, v \vdash \mathit{pat} \ \texttt{=>} \ \mathit{exp} \ \ \Rightarrow \mathrm{FAIL}} \tag{113}$$

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

6.7.31 Rule: *Value declaration.*

$$\frac{E \vdash \mathit{valbind} \Rightarrow \mathit{VE}}{E \vdash \texttt{val} \ \mathit{tyvarseq} \ \mathit{valbind} \Rightarrow \mathit{VE} \text{ in Env}} \tag{114}$$

6.7.32 Rule: *Type alias.*

$$\frac{\vdash \mathit{typbind} \Rightarrow \mathit{TE}}{E \vdash \texttt{type} \ \mathit{typbind} \Rightarrow \mathit{TE} \text{ in Env}} \tag{115}$$

6.7.33 Rule: *Datatype declaration.*

$$\frac{\vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE}}{E \vdash \texttt{datatype} \ \mathit{datbind} \Rightarrow (\mathit{VE}, \mathit{TE}) \text{ in Env}} \tag{116}$$

6.7.34 Rule: *Datatype replication.*

$$\frac{E(\mathit{longtycon}) = \mathit{VE}}{E \vdash \texttt{datatype}\ \mathit{tycon} \ \texttt{-=-} \ \texttt{datatype}\ \mathit{longtycon} \Rightarrow (\mathit{VE}, \{\mathit{tycon} \mapsto \mathit{VE}\}) \text{ in Env}} \tag{117}$$

6.7.35 Rule: *Abstract typing declaration.*

$$\frac{\vdash \mathit{datbind} \Rightarrow \mathit{VE}, \mathit{TE} \qquad E + \mathit{VE} \vdash \mathit{dec} \Rightarrow E'}{E \vdash \texttt{abstype}\ \mathit{datbind}\ \texttt{with}\ \mathit{dec}\ \texttt{end} \Rightarrow E'} \tag{118}$$

6.7.36 Rule: *Exception declaration.*

$$\frac{E \vdash \mathit{exbind} \Rightarrow \mathit{VE}}{E \vdash \texttt{exception}\ \mathit{exbind} \Rightarrow \mathit{VE} \text{ in Env}} \tag{119}$$

6.7.37 Rule: *Local declaration.*

$$\frac{E \vdash \mathit{dec}_1 \Rightarrow E_1 \qquad E + E_1 \vdash \mathit{dec}_2 \Rightarrow E_2}{E \vdash \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} \Rightarrow E_2} \tag{120}$$

6.7.38 Rule: *Open declaration.*

$$\frac{E(\mathit{longstrid}_1) = E_1 \qquad \cdots \qquad E(\mathit{longstrid}_n) = E_n}{E \vdash \texttt{open}\ \mathit{longstrid}_1 \cdots \mathit{longstrid}_n \Rightarrow E_1 + \cdots + E_n} \tag{121}$$

6.7.39 Rule: *Empty declaration.*

$$\frac{}{E \vdash \qquad \Rightarrow \{\} \text{ in Env}} \tag{122}$$

6.7.40 Rule: *Sequential declaration.*

$$\frac{E \vdash \mathit{dec}_1 \Rightarrow E_1 \qquad E + E_1 \vdash \mathit{dec}_2 \Rightarrow E_2}{E \vdash \mathit{dec}_1\ \langle\texttt{;}\rangle\ \mathit{dec}_2 \Rightarrow E_1 + E_2} \tag{123}$$

Value Bindings

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

6.7.41 Rule: *Nonrecursive, possibly simultaneous, bindings.*

$$\frac{E \vdash \mathit{exp} \Rightarrow v \qquad E, v \vdash \mathit{pat} \Rightarrow \mathit{VE} \qquad \langle E \vdash \mathit{valbind} \Rightarrow \mathit{VE}'\rangle}{E \vdash \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle\texttt{and}\ \mathit{valbind}\rangle \Rightarrow \mathit{VE}\ \langle+\ \mathit{VE}'\rangle} \tag{124}$$

6.7.42 Rule: *Binding exception during declaration.*

$$\frac{E \vdash \mathit{exp} \Rightarrow v \qquad E, v \vdash \mathit{pat} \Rightarrow \text{FAIL}}{E \vdash \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle\texttt{and}\ \mathit{valbind}\rangle \Rightarrow [\texttt{Bind}]} \tag{125}$$

6.7.43 Rule: *Recursive value binding.*

$$\frac{E \vdash \mathit{valbind} \Rightarrow \mathit{VE}}{E \vdash \texttt{rec}\ \mathit{valbind} \Rightarrow \text{Rec}\ \mathit{VE}} \tag{126}$$

Type Bindings

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

6.7.44 Rule: *Type binding.*

$$\frac{\langle\vdash \mathit{typbind} \Rightarrow \mathit{TE}\rangle}{\vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle\texttt{and}\ \mathit{typbind}\rangle \Rightarrow \{\mathit{tycon} \mapsto \{\}\}\langle+\mathit{TE}\rangle} \tag{127}$$

Datatype Bindings

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

6.7.45 RULE: *Datatype binding.*

$$\frac{\vdash \mathit{conbind} \Rightarrow \mathit{VE} \qquad \langle \vdash \mathit{datbind}' \Rightarrow \mathit{VE}', \mathit{TE}' \rangle}{\vdash \mathit{tyvarseq}\ \mathit{tycon} \texttt{=} \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind}' \rangle \Rightarrow \mathit{VE}\langle + \mathit{VE}'\rangle, \{\mathit{tycon} \mapsto \mathit{VE}\}\langle + \mathit{TE}' \rangle} \tag{128}$$

Constructor Bindings

$\boxed{\vdash \mathit{conbind} \Rightarrow \mathit{VE}}$

6.7.46 RULE: *Constructor binding.*

$$\frac{\langle \vdash \mathit{conbind} \Rightarrow \mathit{VE} \rangle}{\vdash \mathit{vid} \langle \texttt{|}\ \mathit{conbind} \rangle \Rightarrow \{\mathit{vid} \mapsto (\mathit{vid}, \texttt{c})\}\ \langle + \mathit{VE} \rangle} \tag{129}$$

Exception Bindings

$\boxed{E \vdash \mathit{exbind} \Rightarrow \mathit{VE}}$

6.7.47 RULE: *Exception binding.*

$$\frac{\mathit{en} \notin \mathit{ens}\ \mathrm{of}\ s \qquad s' = s + \{\mathit{en}\} \qquad \langle s', E \vdash \mathit{exbind} \Rightarrow \mathit{VE}, s'' \rangle}{s, E \vdash \mathit{vid}\ \langle \texttt{and}\ \mathit{exbind} \rangle \Rightarrow \{\mathit{vid} \mapsto (\mathit{en}, \texttt{e})\}\langle +\ \mathit{VE} \rangle,\ s'\langle ' \rangle} \tag{130}$$

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

6.7.48 RULE: *Looking up exception binding in environment.*
When a *longvid* is bound to an exception.

$$\frac{E(\mathit{longvid}) = (\mathit{en}, \texttt{e}) \qquad \langle E \vdash \mathit{exbind} \Rightarrow \mathit{VE} \rangle}{E \vdash \mathit{vid}\ \texttt{=}\ \mathit{longvid}\ \langle \texttt{and}\ \mathit{exbind} \rangle \Rightarrow \{\mathit{vid} \mapsto (\mathit{en}, \texttt{e})\}\langle +\ \mathit{VE} \rangle} \tag{131}$$

Atomic Patterns

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

6.7.49 RULE: *Wildcard pattern.*

$$\frac{}{E, v \vdash \_ \Rightarrow \{\}} \tag{132}$$

6.7.50 RULE: *Successfully match against special constant.*

$$\frac{v = \mathrm{val}(\mathit{scon})}{E, v \vdash \mathit{scon} \Rightarrow \{\}} \tag{133}$$

6.7.51 RULE: *Fail to match against special constant.*

$$\frac{v \neq \mathrm{val}(\mathit{scon})}{E, v \vdash \mathit{scon} \Rightarrow \mathrm{FAIL}} \tag{134}$$

Any evaluation resulting in FAIL must do so because rule 134, rule 137, rule 145, or rule 147 has been applied.

6.7.52 RULE: *Variable pattern.*

$$\frac{\mathit{vid} \notin \mathrm{Dom}(E)\ \text{or}\ \mathit{is}\ \text{of}\ E(\mathit{vid}) = \texttt{v}}{E, v \vdash \mathit{vid} \Rightarrow \{\mathit{vid} \mapsto (v, \texttt{v})\}} \tag{135}$$

6.7.53 RULE: *Value/exception constructor pattern.*

$$\frac{E(\mathit{longvid}) = (v, \mathit{is}) \qquad \mathit{is} \neq \texttt{v}}{E, v \vdash \mathit{longvid} \Rightarrow \{\}} \tag{136}$$

6.7.54 Rule: *Value/exception constructor pattern fail.*

$$\frac{E(longvid) = (v', is) \qquad is \neq \mathtt{v} \qquad v \neq v'}{E, v \vdash longvid \Rightarrow \mathrm{FAIL}} \tag{137}$$

Any evaluation resulting in FAIL must do so because rule 134, rule 137, rule 145, or rule 147 has been applied.

6.7.55 Rule: *Record pattern.*

$$\frac{v = \{\}\langle +r\rangle \text{ in Val} \qquad \langle E, r \vdash patrow \Rightarrow VE/\mathrm{FAIL}\rangle}{E, v \vdash \mathtt{\{}\ \langle patrow\rangle\ \mathtt{\}} \Rightarrow \{\}\langle + VE/\mathrm{FAIL}\rangle} \tag{138}$$

6.7.56 Rule: *Parenthesised pattern.*

$$\frac{E, v \vdash pat \Rightarrow VE/\mathrm{FAIL}}{E, v \vdash \mathtt{(}\ pat\ \mathtt{)} \Rightarrow VE/\mathrm{FAIL}} \tag{139}$$

Pattern Rows

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

6.7.57 Rule: *Wildcard record.*

$$\frac{}{E, r \vdash \mathtt{...} \Rightarrow \{\}} \tag{140}$$

6.7.58 Rule: *Record component with inherited FAIL.*

$$\frac{E, r(lab) \vdash pat \Rightarrow \mathrm{FAIL}}{E, r \vdash lab\ \mathtt{=}\ pat\ \langle\ \mathtt{,}\ patrow\rangle \Rightarrow \mathrm{FAIL}} \tag{141}$$

For well-typed programs *lab* will be in the domain of r.

6.7.59 Rule: *Record component.*

$$\frac{E, r(lab) \vdash pat \Rightarrow VE \qquad \langle E, r \vdash patrow \Rightarrow VE'/\mathrm{FAIL}\rangle}{E, r \vdash lab\ \mathtt{=}\ pat\ \langle\ \mathtt{,}\ patrow\rangle \Rightarrow VE\langle +\ VE'/\mathrm{FAIL}\rangle} \tag{142}$$

For well-typed programs *lab* will be in the domain of r.

Patterns

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

6.7.60 Rule: *Atomic pattern.*

$$\frac{E, v \vdash atpat \Rightarrow VE/\mathrm{FAIL}}{E, v \vdash atpat \Rightarrow VE/\mathrm{FAIL}} \tag{143}$$

6.7.61 Rule: *Construction pattern.*

$$\frac{\begin{array}{c} E(longvid) = (vid, \mathtt{c}) \qquad vid \neq \mathtt{ref} \qquad v = (vid, v') \\ E, v' \vdash atpat \Rightarrow VE/\mathrm{FAIL} \end{array}}{E, v \vdash longvid\ atpat \Rightarrow VE/\mathrm{FAIL}} \tag{144}$$

6.7.62 Rule: *Construction pattern failure.*

$$\frac{E(longvid) = (vid, \mathtt{c}) \qquad vid \neq \mathtt{ref} \qquad v \notin \{vid\} \times \mathrm{Val}}{E, v \vdash longvid\ atpat \Rightarrow \mathrm{FAIL}} \tag{145}$$

Any evaluation resulting in FAIL must do so because rule 134, rule 137, rule 145, or rule 147 has been applied.

6.7.63 Rule: *Exception constructor match.*

$$\frac{E(\mathit{longvid}) = (\mathit{en}, \mathtt{e}) \qquad v = (\mathit{en}, v') \qquad E, v' \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\mathrm{FAIL}}{E, v \vdash \mathit{longvid}\ \mathit{atpat} \Rightarrow \mathit{VE}/\mathrm{FAIL}} \tag{146}$$

6.7.64 Rule: *Exception constructor match fail.*

$$\frac{E(\mathit{longvid}) = (\mathit{en}, \mathtt{e}) \qquad v \notin \{\mathit{en}\} \times \mathrm{Val}}{E, v \vdash \mathit{longvid}\ \mathit{atpat} \Rightarrow \mathrm{FAIL}} \tag{147}$$

Any evaluation resulting in FAIL must do so because rule 134, rule 137, rule 145, or rule 147 has been applied.

6.7.65 Rule: *Reference pattern.*

$$\frac{s(a) = v \qquad s, E, v \vdash \mathit{atpat} \Rightarrow \mathit{VE}/\mathrm{FAIL}, s}{s, E, a \vdash \mathtt{ref}\ \mathit{atpat} \Rightarrow \mathit{VE}/\mathrm{FAIL}, s} \tag{148}$$

6.7.66 Rule: *Layered pattern.*

$$\frac{E, v \vdash \mathit{pat} \Rightarrow \mathit{VE}/\mathrm{FAIL}}{E, v \vdash \mathit{vid}\ \mathtt{as}\ \mathit{pat} \Rightarrow \{\mathit{vid} \mapsto (v, \mathtt{v})\} + \mathit{VE}/\mathrm{FAIL}} \tag{149}$$

CHAPTER 7

DYNAMIC SEMANTICS FOR MODULES

7.1. REDUCED SYNTAX

7.1.1 Clause:
The syntax for Modules is reduced in the following manner, *in addition* to the reduced syntax made for Core.

7.1.2 Clause: *Omit qualifications from descriptions.*
Qualifications "`of` *ty*" are omitted from constructor and exception descriptions.

7.1.3 Clause: *Omit sharing/where type qualifications.*
Any qualification "`sharing type` ⋯" on a specification, or "`where type` ⋯" on a signature expression, both are omitted.

7.2. COMPOUND OBJECTS

7.2.1 Definition: *Compound objects.*
The compound objects for the Modules dynamic semantics, in addition to those for the Core dynamic semantics, are listed below:

$$\begin{array}{rcl}
(\mathit{strid} : I, \mathit{strexp}, B) & \in & \mathrm{FunctorClosure} = (\mathrm{StrId} \times \mathrm{Int}) \times \mathrm{StrExp} \times \mathrm{Basis} \\
I \text{ or } (\mathit{SI}, \mathit{TI}, \mathit{VI}) & \in & \mathrm{Int} = \mathrm{StrInt} \times \mathrm{TyInt} \times \mathrm{ValInt} \\
\mathit{SI} & \in & \mathrm{StrInt} = \mathrm{StrId} \xrightarrow{\mathrm{fin}} \mathrm{Int} \\
\mathit{TI} & \in & \mathrm{TyInt} = \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{ValInt} \\
\mathit{VI} & \in & \mathrm{ValInt} = \mathrm{VId} \xrightarrow{\mathrm{fin}} \mathrm{IdStatus} \\
G & \in & \mathrm{SigEnv} = \mathrm{SigId} \xrightarrow{\mathrm{fin}} \mathrm{Int} \\
F & \in & \mathrm{FunEnv} = \mathrm{FunId} \xrightarrow{\mathrm{fin}} \mathrm{FunctorClosure} \\
(F, G, E) \text{ or } B & \in & \mathrm{Basis} = \mathrm{FunEnv} \times \mathrm{SigEnv} \times \mathrm{Env} \\
(G, I) \text{ or } \mathit{IB} & \in & \mathrm{IntBasis} = \mathrm{SigEnv} \times \mathrm{Int}
\end{array}$$

[Used in: 8.0.5]

7.2.2 Definition: *Interface.*
An **"Interface"** $I \in \mathrm{Int}$ represents a "view" of a structure.

7.2.3 Clause: *Signatures evaluate to interfaces.*
Specifications and signature expressions will evaluate to interfaces; moreover, during the evaluation of a specification or signature expression, structures (to which a specification or signature expression may refer via datatype replicating specifications) are represented only by their interfaces.

7.2.4 Definition: *Extracting an interface from an environment.*
To extract a value interface from a dynamic value environment we define the operation $\mathrm{Inter} : \mathrm{ValEnv} \to \mathrm{ValInt}$ as follows:

$$\mathrm{Inter}(\mathit{VE}) = \{\mathit{vid} \mapsto \mathit{is}\ ;\ \mathit{VE}(\mathit{vid}) = (v, \mathit{is})\}$$

In other words, $\mathrm{Inter}(\mathit{VE})$ is the value interface obtained from VE by removing all values from VE.

We then extend Inter to a function Inter : Env $\rightarrow$ Int as follows:

$$\mathrm{Inter}(SE, TE, VE) \;=\; (SI, TI, VI)$$

where $VI = \mathrm{Inter}(VE)$ and

$$SI = \{strid \mapsto \mathrm{Inter}\, E \;;\; SE(strid) = E\}$$
$$TI = \{tycon \mapsto \mathrm{Inter}\, VE' \;;\; TE(tycon) = VE'\}$$

7.2.5 Definition: *Interface basis.*
An **"Interface Basis"** $IB = (G, I)$ is a value-free part of a basis, sufficient to evaluate signature expressions and specifications.

7.2.6 Clause: *Extending Inter to create interface bases.*
The function Inter is extended to create an interface basis from a basis B as follows:

$$\mathrm{Inter}(F, G, E) \;=\; (G, \mathrm{Inter}\, E)$$

7.2.7 Definition: *Cutting down an environment to an interface.*
A further operation

$$\downarrow :\; \mathrm{Env} \times \mathrm{Int} \rightarrow \mathrm{Env}$$

is required, to cut down an environment E to a given interface I, representing the effect of an explicit signature ascription. We first define $\downarrow$: ValEnv $\times$ ValInt $\rightarrow$ ValEnv by

$$VE \downarrow VI = \{vid \mapsto (v, is) \;;\; VE(vid) = (v, is') \text{ and } VI(vid) = is\}$$

(Note that VE and VI need not have the same domain and that the identifier status is taken from VI.) We then define $\downarrow$: StrEnv $\times$ StrInt $\rightarrow$ StrEnv, $\downarrow$: TyEnv $\times$ TyInt $\rightarrow$ TyEnv and $\downarrow$: Env $\times$ Int $\rightarrow$ Env simultaneously as follows:[1]

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

$$TE \downarrow TI = \{tycon \mapsto VE' \downarrow VI' \;;\; TE(tycon) = VE' \text{ and } TI(tycon) = VI'\}$$

$$(SE, TE, VE) \downarrow (SI, TI, VI) = (SE \downarrow SI, TE \downarrow TI, VE \downarrow VI)$$

7.2.8 Clause: *Static elaboration naturally implements interfaces.*
It is important to note that an interface can also be obtained from the *static* value Σ of a signature expression; it is obtained by first replacing every type structure (θ, VE) in the range of every type environment TE by VE and then replacing each pair (σ, is) in the range of every value environment VE by is.

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.

[1]Note: there is a typo in the third equation $(SE, TE, VE) \downarrow (SI, TI, VI)$ which the Definition, as printed, reads erroneously as $(SE, TE, VE) \downarrow (SI, TE, VI)$.

7.3. INFERENCE RULES

7.3.1 DEFINITION: *Judgement form.*
The semantic rules allow sentences of the form

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

to be inferred, where A is either a basis, a signature environment, or empty; A' is some semantic object; and s, s' are states before and after the evaluation represented by the judgement.

7.3.2 DEFINITION: *Side-conditions.*
Some hypotheses in rules are not of this form; they are called **"side-conditions"**.

7.3.3 CONVENTION: *Options same as for Core.*
The conventions for options are the same as for Core (those in single brackets are all taken together, or none are taken together; separately, those in double brackets are all taken together, or none are taken).

7.3.4 CONVENTION: *State and exception conventions.*
The state convention (§6.7.4) and exception convention (§6.7.5) are adopted as in the Core dynamics. However, it may be shown that the only Modules phrases whose evaluation may cause a side-effect or generate an exception packet are of the form *strexp*, *strdec*, *strbind* or *topdec*.
[Used in: 8.0.7]

Structure Expressions

$\boxed{B \vdash \textit{strexp} \Rightarrow E/p}$

7.3.5 RULE: *Generative structure expression.*

$$\frac{B \vdash \textit{strdec} \Rightarrow E}{B \vdash \texttt{struct}\ \textit{strdec}\ \texttt{end} \Rightarrow E} \qquad (150)$$

7.3.6 RULE: *Structure long id.*

$$\frac{B(\textit{longstrid}) = E}{B \vdash \textit{longstrid} \Rightarrow E} \qquad (151)$$

7.3.7 RULE: *Transparent signature constraint.*

$$\frac{B \vdash \textit{strexp} \Rightarrow E \qquad \mathrm{Inter}\, B \vdash \textit{sigexp} \Rightarrow I}{B \vdash \textit{strexp}\,\texttt{:}\,\textit{sigexp} \Rightarrow E \downarrow I} \qquad (152)$$

7.3.8 RULE: *Opaque signature constraint.*

$$\frac{B \vdash \textit{strexp} \Rightarrow E \qquad \mathrm{Inter}\, B \vdash \textit{sigexp} \Rightarrow I}{B \vdash \textit{strexp}\,\texttt{:>}\textit{sigexp} \Rightarrow E \downarrow I} \qquad (153)$$

7.3.9 RULE: *Functor application.*

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

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.

7.3.10 Rule: *Let-structure expression.*

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

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

7.3.11 Rule: *Core declaration.*

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

7.3.12 Comment: *For the "Core fragment" of ML.*
We can ignore the next two rules, and use these structure-level declarations rules to describe the dynamics of the "Core" fragment of Standard ML. See (§8.0.14).

7.3.13 Rule: *Structure declaration.*

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

7.3.14 Rule: *Local structure declaration.*

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \texttt{local}\ \mathit{strdec}_1\ \texttt{in}\ \mathit{strdec}_2\ \texttt{end} \Rightarrow E_2} \tag{158}$$

7.3.15 Rule: *Empty declaration.*

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

7.3.16 Rule: *Sequential declaration.*

$$\frac{B \vdash \mathit{strdec}_1 \Rightarrow E_1 \qquad B + E_1 \vdash \mathit{strdec}_2 \Rightarrow E_2}{B \vdash \mathit{strdec}_1\ \langle\texttt{;}\rangle\ \mathit{strdec}_2 \Rightarrow E_1 + E_2} \tag{160}$$

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

7.3.17 Rule: *Structure binding.*

$$\frac{B \vdash \mathit{strexp} \Rightarrow E \qquad \langle B \vdash \mathit{strbind} \Rightarrow \mathit{SE}\rangle}{B \vdash \mathit{strid}\ \texttt{=}\ \mathit{strexp}\ \langle\texttt{and}\ \mathit{strbind}\rangle \Rightarrow \{\mathit{strid} \mapsto E\}\ \langle + \mathit{SE}\rangle} \tag{161}$$

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

7.3.18 Rule: *Encapsulation signature expressions.*

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

7.3.19 Rule: *Signature identifier.*

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

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

7.3.20 RULE: *Single signature declaration.*

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

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

7.3.21 RULE: *Signature binding.*

$$\frac{IB \vdash \mathit{sigexp} \Rightarrow I \qquad \langle IB \vdash \mathit{sigbind} \Rightarrow G\rangle}{IB \vdash \mathit{sigid}\ \texttt{=}\ \mathit{sigexp}\ \langle\texttt{and}\ \mathit{sigbind}\rangle \Rightarrow \{\mathit{sigid} \mapsto I\}\ \langle + \ G\rangle} \tag{165}$$

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

7.3.22 RULE: *Value specification.*

$$\frac{IB \vdash \texttt{val}\ \mathit{valdesc} \Rightarrow \mathit{VI}\ \text{in Int}}{\vdash \mathit{valdesc} \Rightarrow \mathit{VI}} \tag{166}$$

7.3.23 RULE: *Type specification.*

$$\frac{\vdash \mathit{typdesc} \Rightarrow \mathit{TI}}{IB \vdash \texttt{type}\ \mathit{typdesc} \Rightarrow \mathit{TI}\ \text{in Int}} \tag{167}$$

7.3.24 RULE: *Equality type specification.*

$$\frac{\vdash \mathit{typdesc} \Rightarrow \mathit{TI}}{IB \vdash \texttt{eqtype}\ \mathit{typdesc} \Rightarrow \mathit{TI}\ \text{in Int}} \tag{168}$$

7.3.25 RULE: *Datatype specification.*

$$\frac{\vdash \mathit{datdesc} \Rightarrow \mathit{VI}, \mathit{TI}}{IB \vdash \texttt{datatype}\ \mathit{datdesc} \Rightarrow (\mathit{VI}, \mathit{TI})\ \text{in Int}} \tag{169}$$

7.3.26 RULE: *Datatype replication specification.*

$$\frac{IB(\mathit{longtycon}) = \mathit{VI} \qquad \mathit{TI} = \{\mathit{tycon} \mapsto \mathit{VI}\}}{IB \vdash \texttt{datatype}\ \mathit{tycon}\ \texttt{-=-}\ \texttt{datatype}\ \mathit{longtycon} \Rightarrow (\mathit{VI}, \mathit{TI})\ \text{in Int}} \tag{170}$$

7.3.27 RULE: *Exception specification.*

$$\frac{\vdash \mathit{exdesc} \Rightarrow \mathit{VI}}{IB \vdash \texttt{exception}\ \mathit{exdesc} \Rightarrow \mathit{VI}\ \text{in Int}} \tag{171}$$

7.3.28 RULE: *Structure specification.*

$$\frac{IB \vdash \mathit{strdesc} \Rightarrow \mathit{SI}}{IB \vdash \texttt{structure}\ \mathit{strdesc} \Rightarrow \mathit{SI}\ \text{in Int}} \tag{172}$$

7.3.29 RULE: *Include signature specification.*

$$\frac{IB \vdash \mathit{sigexp} \Rightarrow I}{IB \vdash \texttt{include}\ \mathit{sigexp} \Rightarrow I} \tag{173}$$

7.3.30 RULE: *Empty specification.*

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

7.3.31 RULE: *Sequential specification.*

$$\frac{IB \vdash \mathit{spec}_1 \Rightarrow I_1 \qquad IB + I_1 \vdash \mathit{spec}_2 \Rightarrow I_2}{IB \vdash \mathit{spec}_1\ \langle\texttt{;}\rangle\ \mathit{spec}_2 \Rightarrow I_1 + I_2} \tag{175}$$

Value Descriptions

$\boxed{\vdash \mathit{valdesc} \Rightarrow \mathit{VI}}$

7.3.32 RULE: *Value description.*

$$\frac{\langle \vdash \mathit{valdesc} \Rightarrow \mathit{VI}\rangle}{\vdash \mathit{vid}\ \langle \texttt{and}\ \mathit{valdesc}\rangle \Rightarrow \{\mathit{vid} \mapsto \texttt{v}\}\ \langle +\,\mathit{VI}\rangle} \tag{176}$$

Type Descriptions

$\boxed{\vdash \mathit{typdesc} \Rightarrow \mathit{TI}}$

7.3.33 RULE: *Type description.*

$$\frac{\langle \vdash \mathit{typdesc} \Rightarrow \mathit{TI}\rangle}{\vdash \mathit{tyvarseq}\ \mathit{tycon}\ \langle \texttt{and}\ \mathit{typdesc}\rangle \Rightarrow \{\mathit{tycon} \mapsto \{\}\}\langle +\mathit{TI}\rangle} \tag{177}$$

Datatype Descriptions

$\boxed{\vdash \mathit{datdesc} \Rightarrow \mathit{VI}, \mathit{TI}}$

7.3.34 RULE: *Datatype description.*

$$\frac{\vdash \mathit{condesc} \Rightarrow \mathit{VI} \qquad \langle \vdash \mathit{datdesc}' \Rightarrow \mathit{VI}', \mathit{TI}'\rangle}{\vdash \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{condesc}\ \langle \texttt{and}\ \mathit{datdesc}'\rangle \Rightarrow \mathit{VI}\ \langle +\,\mathit{VI}'\rangle, \{\mathit{tycon} \mapsto \mathit{VI}\}\langle +\mathit{TI}'\rangle} \tag{178}$$

Constructor Descriptions

$\boxed{\vdash \mathit{condesc} \Rightarrow \mathit{VI}}$

7.3.35 RULE: *Constructor description.*

$$\frac{\langle \vdash \mathit{condesc} \Rightarrow \mathit{VI}\rangle}{\vdash \mathit{vid}\ \langle\ \texttt{|}\ \ \mathit{condesc}\rangle \Rightarrow \{\mathit{vid} \mapsto \texttt{c}\}\ \langle +\mathit{VI}\rangle} \tag{179}$$

Exception Descriptions

$\boxed{\vdash \mathit{exdesc} \Rightarrow \mathit{VI}}$

7.3.36 RULE:

$$\frac{\langle \vdash \mathit{exdesc} \Rightarrow \mathit{VI}\rangle}{\vdash \mathit{vid}\ \langle \texttt{and}\ \ \mathit{exdesc}\rangle \Rightarrow \{\mathit{vid} \mapsto \texttt{e}\}\ \langle +\mathit{VI}\rangle} \tag{180}$$

Structure Descriptions

$\boxed{\mathit{IB} \vdash \mathit{strdesc} \Rightarrow \mathit{SI}}$

7.3.37 RULE: *Structure description.*

$$\frac{\mathit{IB} \vdash \mathit{sigexp} \Rightarrow I \qquad \langle \mathit{IB} \vdash \mathit{strdesc} \Rightarrow \mathit{SI}\rangle}{\mathit{IB} \vdash \mathit{strid}\ \texttt{:}\ \ \mathit{sigexp}\ \langle \texttt{and}\ \mathit{strdesc}\rangle \Rightarrow \{\mathit{strid} \mapsto I\}\ \langle +\ \mathit{SI}\rangle} \tag{181}$$

Functor Bindings

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

Note: we have fixed the typo reported by Rossberg [Ros18b].

7.3.38 Rule: *Functor binding.*

$$\frac{\text{Inter}\, B \vdash \textit{sigexp} \Rightarrow I \qquad \langle B \vdash \textit{funbind} \Rightarrow F\rangle}{\begin{array}{c} B \vdash \textit{funid}\ \texttt{(}\ \textit{strid}\ \texttt{:}\ \textit{sigexp}\ \texttt{)}\ \texttt{=}\ \textit{strexp}\ \langle\texttt{and}\ \textit{funbind}\rangle \Rightarrow \\ \{\textit{funid} \mapsto (\textit{strid} : I, \textit{strexp}, B)\}\ \langle +\ F\rangle \end{array}} \tag{182}$$

Functor Declarations

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

7.3.39 Rule: *Functor declaration.*

$$\frac{B \vdash \textit{funbind} \Rightarrow F}{B \vdash \texttt{functor}\ \textit{funbind} \Rightarrow F} \tag{183}$$

Top-level Declarations

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

7.3.40 Rule: *Structure-level declaration.*
Rossberg [Ros18b] correctly identifies a typo in the conclusion: the Definition writes "$B'\langle'\rangle$", but what it means is really "$B'\langle+B''\rangle$". We have fixed it here.

$$\frac{B \vdash \textit{strdec} \Rightarrow E \qquad B' = E \text{ in Basis} \qquad \langle B + B' \vdash \textit{topdec} \Rightarrow B''\rangle}{B \vdash \textit{strdec} \quad \langle\textit{topdec}\rangle \Rightarrow B'\langle+B''\rangle} \tag{184}$$

7.3.41 Rule: *Signature declaration.*
Rossberg [Ros18b] correctly identifies a typo in the conclusion: the Definition writes "$B'\langle'\rangle$", but what it means is really "$B'\langle+B''\rangle$". We have fixed it here.

$$\frac{\text{Inter}\, B \vdash \textit{sigdec} \Rightarrow G \qquad B' = G \text{ in Basis} \qquad \langle B + B' \vdash \textit{topdec} \Rightarrow B''\rangle}{B \vdash \textit{sigdec} \quad \langle\textit{topdec}\rangle \Rightarrow B'\langle+B''\rangle} \tag{185}$$

7.3.42 Rule: *Functor [top] declaration.*
Rossberg [Ros18b] correctly identifies a typo in the conclusion: the Definition writes "$B'\langle'\rangle$", but what it means is really "$B'\langle+B''\rangle$". We have fixed it here.

$$\frac{B \vdash \textit{fundec} \Rightarrow F \qquad B' = F \text{ in Basis} \qquad \langle B + B' \vdash \textit{topdec} \Rightarrow B''\rangle}{B \vdash \textit{fundec} \quad \langle\textit{topdec}\rangle \Rightarrow B'\langle+B''\rangle} \tag{186}$$

CHAPTER 8

PROGRAMS

8.0.1 Definition: *Syntactic class of programs.*
The phrase class Program of programs is defined as follows

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

8.0.2 Clause: *Inclusion of files is implementation-dependent.*
The Definition observes that sometimes the Standard ML compiler will be reading in a file, and at other times we will be working with a REPL. In either case, if an error is found, we should wish to avoid creating a distinction between, say, batch programs and interactive programs, and so the Definition tacitly regards all programs as interactive. This leaves it to implementers to clarify the inclusion of files.

Specifically, how including files affects updating the static and dynamic basis — this is implementation dependent behaviour.

8.0.3 Clause: *Parsing error reporting is implementation dependent.*
How parsing errors are handled is left to the implementers (since it depends on the kind of parser employed).

8.0.4 Clause: *Execution of programs.*
In this section, the **"Execution"** of a program means the combined elaboration and evaluation of the program.

8.0.5 Definition: *Class of Basis, StaticBasis, DynamicBasis.*
The Definition wishes to disambiguate the usage of B for bases (since it was used as the metavariable for both static and dynamic bases). For ease of discussion, the Definition does the following:

(1) In giving the semantics of programs, however, let us rename as StaticBasis the class Basis defined in the static semantics of modules (§5.1.2), Section 5.1, and let us use B_{STAT} to range over StaticBasis.
(2) Let us rename as DynamicBasis the class Basis defined in the dynamic semantics of modules (§7.2.1), Section 7.2, and let us use B_{DYN} to range over DynamicBasis.
(3) We now define

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

8.0.6 Definition: *Judgement forms.*
The Definition uses $\vdash_{\mathrm{STAT}}$ for elaboration as defined in Section [Chapter] 5, and $\vdash_{\mathrm{DYN}}$ for evaluation as defined in Section [Chapter] 7. Then $\vdash$ will be reserved for the execution of programs, which thus is expressed by a sentence of the form

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

This may be read as follows: starting in basis B with state s the execution of *program* results in a basis B' and a state s'.

8.0.7 Clause: *Programs never result in exceptions.*
Executing a program never results in an exception. If the evaluation of a *topdec* yields an exception (for instance because of a `raise` expression), then the result of executing the program "*topdec* `;`" is the original basis together with the state which is in force when the exception is generated.

In particular, this means the exception convention (§7.3.4) is not applicable to the rules the Definition describes in this section [chapter].

8.0.8 Convention: *Non-elaboration of top-level declaration.*
The Definition represents the non-elaboration of a top-level declaration by "$\dots\vdash_{\mathrm{STAT}}$ *topdec* $\not\Rightarrow$".

Programs

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

8.0.9 Rule: *Failing elaboration.*

$$\frac{B_{\mathrm{STAT}}\ \text{of}\ B \vdash_{\mathrm{STAT}} \mathit{topdec} \not\Rightarrow \qquad \langle s, B \vdash \mathit{program} \Rightarrow B', s'\rangle}{s, B \vdash \mathit{topdec}\ \texttt{;}\ \langle \mathit{program}\rangle \Rightarrow B\langle'\rangle, s\langle'\rangle} \tag{187}$$

A failing elaboration has no effect whatever.

8.0.10 Comment: *Implications of negative premise in rule 187.*
Rossberg [Ros18b] observes the negative premise in rule 187 has unfortunate implications.

If we interpret the rule strictly, it precludes any conforming implementation from proving any sort of conservative semantic extension to the language. Any extension that allows declarations to elaborate that would be illegal according to the Definition (e.g., polymorphic records) can be observed through this rule and change the behaviour of consecutive declarations. This could alter the observed behaviour of a program. Consider Rossberg's example:

```
val s = "no";
strdec
val s = "yes";
print s;
```

where the *strdec* only elaborates if some extension is supported. In that case, the program will print `yes`, otherwise it will print `no`.

Rossberg interprets this as suggesting that formalizing an interactive toplevel is not worth the trouble. He's probably right.

8.0.11 Rule: *Evaluation raising exception.*

$$\frac{\begin{array}{l} B_{\mathrm{STAT}}\ \text{of}\ B \vdash_{\mathrm{STAT}} \mathit{topdec} \Rightarrow B^{(1)}_{\mathrm{STAT}} \\ s, B_{\mathrm{DYN}}\ \text{of}\ B \vdash_{\mathrm{DYN}} \mathit{topdec} \Rightarrow p, s' \qquad \langle s', B \vdash \mathit{program} \Rightarrow B', s''\rangle \end{array}}{s, B \vdash \mathit{topdec}\ \texttt{;}\ \langle \mathit{program}\rangle \Rightarrow B\langle'\rangle, s'\langle'\rangle} \tag{188}$$

An evaluation which yields an exception nullifies the change in the static basis, but does not nullify side-effects on the state which may have occurred before the exception was raised.

8.0.12 Rule: *Success.*

$$\frac{\begin{array}{ll} B_{\mathrm{STAT}}\ \text{of}\ B \vdash_{\mathrm{STAT}} \mathit{topdec} \Rightarrow B^{(1)}_{\mathrm{STAT}} & \\ s, B_{\mathrm{DYN}}\ \text{of}\ B \vdash_{\mathrm{DYN}} \mathit{topdec} \Rightarrow B^{(1)}_{\mathrm{DYN}}, s' & B' = B \oplus (B^{(1)}_{\mathrm{STAT}}, B^{(1)}_{\mathrm{DYN}}) \\ & \langle s', B' \vdash \mathit{program} \Rightarrow B'', s''\rangle \end{array}}{s, B \vdash \mathit{topdec}\ \texttt{;}\ \langle \mathit{program}\rangle \Rightarrow B'\langle'\rangle, s'\langle'\rangle} \tag{189}$$

8.0.13 Comment: *Ambiguity in semicolons.*
Recall (§3.6.7) "No *topdec* may contain, as an initial segment, a *strdec* followed by a semicolon." Kahr [Kah93, §8.3] and Rossberg [Ros18b] both point out this is a problem.

Kahr points out that "dec_1 ; dec_2 ; dec_3" can be parsed in two different ways (at least), taking dec_{12} as dec_1 ; dec_2 and dec_{23} as dec_2 ; dec_3, these evaluate to the same thing provided in rule 24 and rule 123 we can have

$$C \oplus (E_1 \oplus E_2) = (C \oplus E_1) \oplus E_2.$$

Unfortunately, this does not hold, and the right-hand side of this equation can contain more type names than the one of the left-hand side. This means that the associativity of $\langle$;$\rangle$ is not obvious.

Rossberg resolves this problem by having HaMLet's parser reduce to *strdec* as soon as possible (since this solves other problems as well). This seems to be the "spirit of the Law", even if the "letter of the Law" leaves much to be desired.

8.0.14 Definition: *Core language programs.*
A program is called a **"Core Language Program"** if it can be parsed in the reduced grammar defined as follows:
(1) Replace the definition of top-level declarations by

$$topdec \ ::= \ strdec$$

(2) Replace the definition of structure-level declarations by

$$strdec \ ::= \ dec$$

[Used in: 7.3.12]

8.0.15 Puzzle: *Do first-class structures simplify things?*
Russo [Rus98, Rus00] and later inspired by this Rossberg [Ros18a] proposed first-class structures (modules, signatures, functors) for Standard ML. Since facilitating the second-class nature of the module system complicates things greatly, do we simplify life by making modules first-class?

Bibliography

[Kah93] Stefan Kahrs, *Mistakes and Ambiguities in the definition of Standard ML*, LFCS Report ECS-LFCS-93-257, University of Edinburgh, April 1993, Available at `http://www.cs.kent.ac.uk/pubs/1993/569`. ix, 13, 20, 56, 77

[Kah96] ______, *Mistakes and Ambiguities in the definition of Standard ML — Addenda*, `http://www.cs.kent.ac.uk/~smk/errors-new.ps.Z`, 1996. ix

[Ken24] Alan U Kennington, *Differential geometry reconstructed: A unified systematic framework*, 2024, draft available `http://www.topology.org/tex/conc/dgstats.php`. viii

[MT91] Robin Milner and Mads Tofte, *Commentary on Standard ML*, MIT press, 1991. vii, viii, 27, 57

[MTH90] Robin Milner, Mads Tofte, and Robert Harper, *The Definition of Standard ML*, MIT press, 1990. vii

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen, *The Definition of Standard ML: Revised*, MIT press, 1997. vii, 11

[Pie05] Benjamin C Pierce, *Advanced Topics in Types and Programming Languages*, MIT press, 2005. 79

[PR05] François Pottier and Didier Rémy, *The Essence of ML Type Inference*, in Pierce [Pie05], Extended version available `http://cristal.inria.fr/attapl/emlti-long.pdf`, pp. 389–489. 31

[Ros18a] Andreas Rossberg, *1ML—Core and Modules United*, Journal of Functional Programming **28** (2018), e22. 77

[Ros18b] ______, *Defects in the Revised Definition of Standard ML*, `https://people.mpi-sws.org/~rossberg/papers/sml-defects-2013-09-18.pdf`, 2018. ix, 20, 24, 26, 56, 73, 76, 77

[Rus98] Claudio V Russo, *Types for Modules*, Ph.D. thesis, University of Edinburgh, 1998. 77

[Rus00] ______, *First-class structures for Standard ML*, European Symposium on Programming, Springer, 2000, pp. 336–350. 77

[SJ23] Elizabeth Scott and Adrian Johnstone, *Analysing the SML97 Definition: Lexicalisation*, Eelco Visser Commemorative Symposium (EVCS 2023), Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023. 26

[SJW23] Elizabeth Scott, Adrian Johnstone, and Robert Walsh, *Multiple input parsing and lexical analysis*, ACM Transactions on Programming Languages and Systems **45** (2023), no. 3, 1–44. 18

Index

Summary Statatistics:

– Total items: 424
– clauses 78
– comments 29
– conventions 16
– definitions 81
– deviations 2
– examples 4
– grammars 10
– inference-rules 189
– puzzles 3
– remarks 11
– theorems 1