The Definition of Standard ML

Revised 1996

October 31, 1996
Version: 7.4.4

Robin Milner, Mads Tofte, Robert Harper and David MacQueen

Prelude to the Revised Definition

Standard ML is an industrial strength programming language, one of very few with a fully formal definition. **The Definition of Standard ML** was published in 1990. Since then the implementation technology of the language has advanced enormously, and its users have multiplied. The language and its Definition have therefore incited close scrutiny, evaluation, much approval, sometimes strong criticism.

The originators of the language have sifted this response, and found that there are inadequacies in the original language and its formal Definition. They are of three kinds: missing features which many users want; complex and little-used features which most users can do without; and mistakes of definition. What is remarkable is that these inadequacies are rather few, and that they are rather uncontroversial.

This new version of the Definition addresses the three kinds of inadequacy respectively by additions, subtractions and corrections. But we have only made such amendments when one or more aspects of SML – the language itself, its usage, its implementation, its formal Definition – have thus become simpler, without complicating the other aspects. It is worth noting that even the additions meet this criterion; for example we have introduced type abbreviations in signatures to simplify the use of the language, but the way we have done it has even simplified the Definition too. In fact, after our changes the formal Definition has fewer rules.

In this exercise we have consulted the major implementers and several users, and have found broad agreement. In the 1990 Definition it was predicted that further versions of the Definition would be produced as the language develops, with the intention to minimise the number of versions. This is the first revised version, and we foresee no others.

The shape of this new version of the Definition is as follows. The 1990 Definition, fully revised to deal with the amendments, has been sandwiched between this Prelude at the front, and a postlude (Appendix G) at the back which enumerates all the amendments which have been done, giving the rationale for each, and outlining the changes which it implies for the language and for the 1990 Definition.

We wish to thank Dave Berry, Lars Birkedal, Martin Elsman, Stefan Kahrs and John Reppy for many detailed comments and suggestions.

Robin Milner Mads Tofte Robert Harper David MacQueen

October 1996

Preface

A precise description of a programming language is a prerequisite for its implementation and for its use. The description can take many forms, each suited to a different purpose. A common form is a reference manual, which is usually a careful narrative description of the meaning of each construction in the language, often backed up with a formal presentation of the grammar (for example, in Backus-Naur form). This gives the programmer enough understanding for many of his purposes. But it is ill-suited for use by an implementer, or by someone who wants to formulate laws for equivalence of programs, or by a programmer who wants to design programs with mathematical rigour.

This document is a formal description of both the *grammar* and the *meaning* of a language which is both designed for large projects and widely used. As such, it aims to serve the whole community of people seriously concerned with the language. At a time when it is increasingly understood that programs must withstand rigorous analysis, particular for systems where safety is critical, a rigorous language presentation is even important for negotiators and contractors; for a robust program written in an insecure language is like a house built upon sand.

Most people have not looked at a rigorous language presentation before. To help them particularly, but also to put the present work in perspective for those more theoretically prepared, it will be useful here to say something about three things: the nature of Standard ML, the task of language definition in general, and the form of the present Definition.

Standard ML

Standard ML is a functional programming language, in the sense that the full power of mathematical functions is present. But it grew in response to a particular programming task, for which it was equipped also with full imperative power, and a sophisticated exception mechanism. It has an advanced form of parametric modules, aimed at organised development of large programs. Finally it is strongly typed, and it was the first language to provide a particular form of polymorphic type which makes the strong typing remarkably flexible. This combination of ingredients has not made it unduly large, but their novelty has been a fascinating challenge to semantic method (of which we say more below).

ML has evolved over twenty years as a fusion of many ideas from many people. This evolution is described in some detail in Appendix F of the book, where also we acknowledge all those who have contributed to it, both in design and in implementation.

'ML' stands for *meta language*; this is the term logicians use for a language in which other (formal or informal) languages are discussed and analysed. Originally ML was conceived as a medium for finding and performing proofs in a logical language. Conducting rigorous argument as dialogue between person and machine has been a growing research topic throughout these twenty years. The difficulties are enormous, and make stern demands upon the programming language which is used for this dialogue. Those who are not familiar with computer-assisted reasoning may be surprised that a programming language, which was designed for this rather esoteric activity, should ever lay claim to being *generally* useful. On reflection, they should not be surprised. LISP is a prime example of a language invented for esoteric purposes and becoming widely used. LISP was invented for use in artificial intelligence (AI); the important thing about AI here is not that it is esoteric, but that it is difficult and varied; so much so, that anything which works well for it must work well for many other applications too.

The same can be said about the initial purpose of ML, but with a different emphasis. Rigorous proofs are complex things, which need varied and sophisticated presentation – particularly on the screen in interactive mode. Furthermore the proof methods, or strategies, involved are some of the most complex algorithms which we know. This all applies equally to AI, but one demand is made more strongly by proof than perhaps by any other application: the demand for rigour.

This demand established the character of ML. In order to be sure that, when the user and the computer claim to have together performed a rigorous argument, their claim is justified, it was seen that the language must be strongly typed. On the other hand, to be useful in a difficult application, the type system had to be rather flexible, and permit the machine to guide the user rather than impose a burden upon him. A reasonable solution was found, in which the machine helps the user significantly by inferring his types for him. Thereby the machine also confers complete reliability on his programs, in this sense: If a program claims that a certain result follows from the rules of reasoning which the user has supplied, then the claim may be fully trusted.

The principle of inferring useful structural information about programs is also represented, at the level of program modules, by the inference of *signatures*. Signatures describe the interfaces between modules, and are vital for robust large-scale programs. When the user combines modules, the signature discipline prevents him from mismatching their interfaces. By programming with interfaces and parametric modules, it becomes possible to focus on the structure of a large system, and to compile parts of it in isolation from one another – even when the system is incomplete.

This emphasis on types and signatures has had a profound effect on the language Definition. Over half this document is devoted to inferring types and signatures for programs. But the method used is exactly the same as for inferring what *values* a program delivers; indeed, a type or signature is the result of a kind of abstract evaluation of a program phrase.

In designing ML, the interplay among three activities – language design, definition and implementation – was extremely close. This was particularly true for the newest part, the parametric modules. This part of the language grew from an initial proposal by David MacQueen, itself highly developed; but both formal definition and implementation had a strong influence on the detailed design. In general, those who took part in the three activities cannot now imagine how they could have been properly done separately.

Language Definition

Every programming language presents its own conceptual view of computation. This view is usually indicated by the names used for the phrase classes of the language, or by its keywords: terms like package, module, structure, exception, channel, type, procedure, reference, sharing, ... These terms also have their abstract counterparts, which may be called *semantic objects*; these are what people really have in mind when they use the language, or discuss it, or think in it. Also, it is these objects, not the syntax, which represent the particular conceptual view of each language; they are the character of the language. Therefore a definition of the language must be in terms of these objects.

As is commonly done in programming language semantics, we shall loosely talk of these semantic objects as *meanings*. Of course, it is perfectly possible to understand the semantic theory of a language, and yet be unable to understand the meaning of a particular program, in the sense of its *intention* or *purpose*. The aim of a language definition is not to formalise everything which could possibly be called the meaning of a program, but to establish a theory of semantic objects upon which the understanding of particular programs may rest.

The job of a language-definer is twofold. First – as we have already suggested – he must create a world of meanings appropriate for the language, and must find a way of saying what these meanings precisely are. Here, he meets a problem; notation of *some* kind must be used to denote and describe these meanings – but not a *programming language* notation, unless he is passing the buck and defining one programming language in terms of another. Given a concern for rigour, mathematical notation is an obvious choice. Moreover, it is not enough just to write down mathematical definitions. The world of meanings only becomes meaningful if the objects possess nice properties, which make them tractable. So the language-definer really has to develop a small *theory* of his meanings, in the same way that a mathematician develops a theory. Typically, after initially defining some objects, the mathematician goes on to verify properties which indicate that they are objects worth studying. It is this part, a kind of scene-setting, which the language-definer shares with the mathematician. Of course he can take many objects and their theories directly from mathematics, such as functions, relations, trees, sequences, ... But he must also give some special theory for the objects which make his language particular, as we do for types, structures and signatures in this book; otherwise his language definition may be formal but will give no insight.

The second part of the definer's job is to define *evaluation* precisely. This means that he must define at least *what* meaning, M, results from evaluating any phrase P of his language (though he need not explain exactly *how* the meaning results; that is he need not give the full detail of every computation). This part of his job must be formal to some extent, if only because the phrases P of his language are indeed formal objects. But there is another reason for formality. The task is complex and error-prone, and therefore demands a high level of explicit organisation (which is, largely, the meaning of 'formality'); moreover, it will be used to specify an equally complex, error-prone and formal construction: an implementation.

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

1 Introduction

This document formally defines Standard ML.

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

In the execution of a declaration there are three phases: *parsing*, *elaboration*, and *evaluation*. Parsing determines the grammatical form of a declaration. Elaboration, the *static* phase, determines whether it is well-typed and well-formed in other ways, and records relevant type or form information in the basis. Finally evaluation, the *dynamic* phase, determines the value of the declaration and records relevant value information in the basis. Corresponding to these phases, our formal 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.

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

A further factoring of the formal definition is possible, because of the structure of the language. ML consists of a lower level called the *Core language* (or *Core* for short), a middle level concerned with programming-in-the-large called *Modules*, and a very small upper level called *Programs*. With the three phases described above, there is therefore a possibility of nine components in the complete language definition. We have allotted one section to each of these components, except that we have combined the parsing, elaboration and evaluation of Programs in one section. The scheme for the ensuing seven sections is therefore as follows:

	Core	*Modules*	*Programs*
Syntax	Section 2	Section 3	Section 8
Static Semantics	Section 4	Section 5	
Dynamic Semantics	Section 6	Section 7	

The Core provides many phrase classes, for programming convenience. But about half of these classes are derived forms, whose meaning can be given by translation into the other half which we call the *Bare* language. Thus each of the three parts for the

Core treats only the bare language; the derived forms are treated in Appendix A. This appendix also contains a few derived forms for Modules. A full grammar for the language is presented in Appendix B.

In Appendices C and D the *initial basis* is detailed. This basis, divided into its static and dynamic parts, contains the static and dynamic meanings of a small set of predefined identifiers. A richer basis is defined in a separate document[18].

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

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

to be inferred, where A is often a basis (static or dynamic) and A' a semantic object – often a type in the static semantics and a value in the dynamic semantics. One should read such a sentence as follows: "against the background provided by A, the phrase $phrase$ elaborates – or evaluates – to the object A'". Although the rules themselves are formal the semantic objects, particularly the static ones, are the subject of a mathematical theory which is presented in a succinct form in the relevant sections.

The robustness of the semantics depends upon theorems. Usually these have been proven, but the proof is not included.

2 Syntax of the Core

2.1 Reserved Words

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

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

2.2 Special constants

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

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

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

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

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

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

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

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

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

2.3 Comments

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

2.4 Identifiers

The classes of *identifiers* for the Core are shown in Figure 1. We use *vid*, *tyvar* to range over VId, TyVar etc. For each class X marked "long" there is a class longX of *long identifiers*; if x ranges over X then *longx* ranges over longX. The syntax of these long identifiers is given by the following:

longx	::=	x	identifier
		$strid_1.\cdots.strid_n.x$	qualified identifier ($n \geq 1$)

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

Figure 1: Identifiers

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

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

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

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

A type variable *tyvar* may be any alphanumeric identifier starting with a prime; the subclass EtyVar of TyVar, the *equality* type variables, consists of those which start with two or more primes. The classes VId, TyCon and Lab are represented by identifiers not starting with a prime. However, `*` is excluded from TyCon, to avoid confusion with the derived form of tuple type (see Figure 23). The class Lab is extended to include the *numeric* labels `1 2 3` $\cdots$, i.e. any numeral not starting with `0`. The identifier class StrId is represented by alphanumeric identifiers not starting with a prime.

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

1. Immediately before "." – i.e. in a long identifier – or in an `open` declaration, *id* is a structure identifier. The following rules assume that all occurrences of structure identifiers have been removed.
2. At the start of a component in a record type, record pattern or record expression, *id* is a record label.
3. Elsewhere in types *id* is a type constructor.
4. Elsewhere, *id* is a value identifier.

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

2.5 Lexical analysis

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

2.6 Infixed operators

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

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

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

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

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

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`. `infix` and `infixr` dictate left and right associativity respectively. 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. For example, suppose that `<<` and `>>` have equal precedence, but associate to the left and right respectively; then

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

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

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

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

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

then the scope of *dir* does not extend beyond the phrase. Further scope limitations are imposed for Modules (see Section 3.3).

These directives and `op` are omitted from the semantic rules, since they affect only parsing.

strexp	$::=$	$\mathtt{struct}\ \mathit{strdec}\ \mathtt{end}$	basic
		$\mathit{longstrid}$	structure identifier
		$\mathit{strexp}\,\mathtt{:}\,\mathit{sigexp}$	transparent constraint
		$\mathit{strexp}\,\mathtt{:>}\mathit{sigexp}$	opaque constraint
		$\mathit{funid}\ \mathtt{(}\ \mathit{strexp}\ \mathtt{)}$	functor application
		$\mathtt{let}\ \mathit{strdec}\ \mathtt{in}\ \mathit{strexp}\ \mathtt{end}$	local declaration
strdec	$::=$	dec	declaration
		$\mathtt{structure}\ \mathit{strbind}$	structure
		$\mathtt{local}\ \mathit{strdec}_1\ \mathtt{in}\ \mathit{strdec}_2\ \mathtt{end}$	local
			empty
		$\mathit{strdec}_1\ \langle\mathtt{;}\rangle\ \mathit{strdec}_2$	sequential
$\mathit{strbind}$	$::=$	$\mathit{strid}\ \mathtt{=}\ \mathit{strexp}\ \langle\mathtt{and}\ \mathit{strbind}\rangle$	
sigexp	$::=$	$\mathtt{sig}\ \mathit{spec}\ \mathtt{end}$	basic
		sigid	signature identifier
		$\mathit{sigexp}\ \mathtt{where\ type}$ $\mathit{tyvarseq}\ \mathit{longtycon}\ \mathtt{=}\ \mathit{ty}$	type realisation
sigdec	$::=$	$\mathtt{signature}\ \mathit{sigbind}$	
$\mathit{sigbind}$	$::=$	$\mathit{sigid}\ \mathtt{=}\ \mathit{sigexp}\ \langle\mathtt{and}\ \mathit{sigbind}\rangle$	

Figure 6: Grammar: Structure and Signature Expressions

spec	::=	`val` *valdesc*	value
		`type` *typdesc*	type
		`eqtype` *typdesc*	eqtype
		`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`	sharing
		$longtycon_1$ `=` $\cdots$ `=` $longtycon_n$	$(n \geq 2)$
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*⟩ ⟨`and` *exdesc*⟩	
strdesc	::=	*strid* `:` *sigexp* ⟨`and` *strdesc*⟩	

Figure 7: Grammar: Specifications

fundec	::=	`functor` *funbind*	
funbind	::=	*funid* `(` *strid* `:` *sigexp* `)` `=` *strexp* ⟨`and` *funbind*⟩	functor binding
topdec	::=	*strdec* ⟨*topdec*⟩	structure-level declaration
		sigdec ⟨*topdec*⟩	signature declaration
		fundec ⟨*topdec*⟩	functor declaration

Restriction: No *topdec* may contain, as an initial segment, a *strdec* followed by a semicolon.

Figure 8: Grammar: Functors and Top-level Declarations

atexp	::=	*scon*	special constant
		⟨`op`⟩*longvid*	value identifier
		`{` ⟨*exprow*⟩ `}`	record
		`#` *lab*	record selector
		`()`	0-tuple
		`(`exp_1 `,` $\cdots$ `,` exp_n`)`	n-tuple, $n \geq 2$
		`[`exp_1 `,` $\cdots$ `,` exp_n`]`	list, $n \geq 0$
		`(`exp_1 `;` $\cdots$ `;` exp_n`)`	sequence, $n \geq 2$
		`let` *dec* `in` exp_1 `;` $\cdots$ `;` exp_n `end`	local declaration, $n \geq 1$
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
appexp	::=	*atexp*	
		appexp atexp	application expression
infexp	::=	*appexp*	
		$infexp_1$ *vid* $infexp_2$	infix expression
exp	::=	*infexp*	
		exp `:` *ty*	typed (L)
		exp_1 `andalso` exp_2	conjunction
		exp_1 `orelse` exp_2	disjunction
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`if` exp_1 `then` exp_2 `else` exp_3	conditional
		`while` exp_1 `do` exp_2	iteration
		`case` *exp* `of` *match*	case analysis
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	
mrule	::=	*pat* `=>` *exp*	

Figure 20: Grammar: Expressions and Matches

$$
\begin{array}{llll}
\mathit{dec} & ::= & \texttt{val}\ \mathit{tyvarseq}\ \mathit{valbind} & \text{value declaration} \\
 & & \texttt{fun}\ \mathit{tyvarseq}\ \mathit{fvalbind} & \text{function declaration} \\
 & & \texttt{type}\ \mathit{typbind} & \text{type declaration} \\
 & & \texttt{datatype}\ \mathit{datbind}\ \langle\texttt{withtype}\ \mathit{typbind}\rangle & \text{datatype declaration} \\
 & & \texttt{datatype}\ \mathit{tycon}\ \texttt{=}\ \texttt{datatype}\ \mathit{longtycon} & \text{datatype replication} \\
 & & \texttt{abstype}\ \mathit{datbind}\ \langle\texttt{withtype}\ \mathit{typbind}\rangle & \text{abstype declaration} \\
 & & \quad\texttt{with}\ \mathit{dec}\ \texttt{end} & \\
 & & \texttt{exception}\ \mathit{exbind} & \text{exception declaration} \\
 & & \texttt{local}\ \mathit{dec}_1\ \texttt{in}\ \mathit{dec}_2\ \texttt{end} & \text{local declaration} \\
 & & \texttt{open}\ \mathit{longstrid}_1 \cdots \mathit{longstrid}_n & \text{open declaration, } n \geq 1 \\
 & & & \text{empty declaration} \\
 & & \mathit{dec}_1\ \langle\texttt{;}\rangle\ \mathit{dec}_2 & \text{sequential declaration} \\
 & & \texttt{infix}\ \langle d\rangle\ \mathit{vid}_1 \cdots \mathit{vid}_n & \text{infix (L) directive, } n \geq 1 \\
 & & \texttt{infixr}\ \langle d\rangle\ \mathit{vid}_1 \cdots \mathit{vid}_n & \text{infix (R) directive, } n \geq 1 \\
 & & \texttt{nonfix}\ \mathit{vid}_1 \cdots \mathit{vid}_n & \text{nonfix directive, } n \geq 1 \\
\mathit{valbind} & ::= & \mathit{pat}\ \texttt{=}\ \mathit{exp}\ \langle\texttt{and}\ \mathit{valbind}\rangle & \\
 & & \texttt{rec}\ \mathit{valbind} & \\
\mathit{fvalbind} & ::= & \quad\langle\texttt{op}\rangle\mathit{vid}\ \mathit{atpat}_{11}\cdots\mathit{atpat}_{1n}\langle\texttt{:}\,\mathit{ty}\rangle\texttt{=}\mathit{exp}_1 & m, n \geq 1 \\
 & & \texttt{|}\langle\texttt{op}\rangle\mathit{vid}\ \mathit{atpat}_{21}\cdots\mathit{atpat}_{2n}\langle\texttt{:}\,\mathit{ty}\rangle\texttt{=}\mathit{exp}_2 & \text{See also note below} \\
 & & \texttt{|}\qquad\cdots\qquad\cdots & \\
 & & \texttt{|}\langle\texttt{op}\rangle\mathit{vid}\ \mathit{atpat}_{m1}\cdots\mathit{atpat}_{mn}\langle\texttt{:}\,\mathit{ty}\rangle\texttt{=}\mathit{exp}_m & \\
 & & \qquad\qquad\qquad\langle\texttt{and}\ \mathit{fvalbind}\rangle & \\
\mathit{typbind} & ::= & \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle\texttt{and}\ \mathit{typbind}\rangle & \\
\mathit{datbind} & ::= & \mathit{tyvarseq}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle\texttt{and}\ \mathit{datbind}\rangle & \\
\mathit{conbind} & ::= & \langle\texttt{op}\rangle\mathit{vid}\ \langle\texttt{of}\ \mathit{ty}\rangle\ \langle\ \texttt{|}\ \mathit{conbind}\rangle & \\
\mathit{exbind} & ::= & \langle\texttt{op}\rangle\mathit{vid}\ \langle\texttt{of}\ \mathit{ty}\rangle\ \langle\texttt{and}\ \mathit{exbind}\rangle & \\
 & & \langle\texttt{op}\rangle\mathit{vid}\ \texttt{=}\ \langle\texttt{op}\rangle\mathit{longvid}\ \langle\texttt{and}\ \mathit{exbind}\rangle &
\end{array}
$$

Note: In the *fvalbind* form, if *vid* has infix status then either `op` must be present, or *vid* must be infixed. Thus, at the start of any clause, "`op` *vid* `(`*atpat*`,`*atpat′*`)` ⋯" may be written "`(`*atpat vid atpat′*`)` ⋯"; the parentheses may also be dropped if "`:`*ty*" or "`=`" follows immediately.

Figure 21: Grammar: Declarations and Bindings

atpat	::=	`_`	wildcard
		scon	special constant
		⟨`op`⟩*longvid*	value identifier
		`{` ⟨*patrow*⟩ `}`	record
		`()`	0-tuple
		`(`pat_1 `,` $\cdots$ `,` pat_n`)`	n-tuple, $n \geq 2$
		`[`pat_1 `,` $\cdots$ `,` pat_n`]`	list, $n \geq 0$
		`(` *pat* `)`	
patrow	::=	`...`	wildcard
		lab `=` *pat* ⟨ `,` *patrow*⟩	pattern row
		vid⟨`:`*ty*⟩ ⟨`as` *pat*⟩ ⟨`,` *patrow*⟩	label as variable
pat	::=	*atpat*	atomic
		⟨`op`⟩*longvid atpat*	constructed value
		pat_1 *vid* pat_2	constructed value (infix)
		pat `:` *ty*	typed
		⟨`op`⟩*vid*⟨`:` *ty*⟩ `as` *pat*	layered

Figure 22: Grammar: Patterns

ty	::=	*tyvar*	type variable
		`{` ⟨*tyrow*⟩ `}`	record type expression
		tyseq longtycon	type construction
		ty_1 `*` $\cdots$ `*` ty_n	tuple type, $n \geq 2$
		ty `->` *ty′*	function type expression (R)
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow*⟩	type-expression row

Figure 23: Grammar: Type expressions

C Appendix: The Initial Static Basis

In this appendix (and the next) we define a minimal initial basis for execution. Richer bases may be provided by libraries. We shall indicate components of the initial basis by the subscript 0. The initial static basis is $B_0 = T_0, F_0, G_0, E_0$, where $F_0 = \{\}$, $G_0 = \{\}$ and

$$T_0 \;=\; \{\mathtt{bool}, \mathtt{int}, \mathtt{real}, \mathtt{string}, \mathtt{char}, \mathtt{word}, \mathtt{list}, \mathtt{ref}, \mathtt{exn}\}$$

The members of T_0 are type names, not type constructors; for convenience we have used type-constructor identifiers to stand also for the type names which are bound to them in the initial static type environment TE_0. Of these type names, `list` and `ref` have arity 1, the rest have arity 0; all except `exn` and `real` admit equality. Finally, $E_0 = (SE_0, TE_0, VE_0)$, where $SE_0 = \{\}$, while TE_0 and VE_0 are shown in Figures 24 and 25, respectively.

$tycon$	$\mapsto$	$(\ \theta,$	$\{vid_1 \mapsto (\sigma_1, is_1), \ldots, vid_n \mapsto (\sigma_n, is_n)\}\ )$	$(n \geq 0)$
`unit`	$\mapsto$	$(\ \Lambda().\{\},$	$\{\}\ )$	
`bool`	$\mapsto$	$(\ \mathtt{bool},$	$\{\mathtt{true} \mapsto (\mathtt{bool}, \mathtt{c}),\ \mathtt{false} \mapsto (\mathtt{bool}, \mathtt{c})\}\ )$	
`int`	$\mapsto$	$(\ \mathtt{int},$	$\{\}\ )$	
`word`	$\mapsto$	$(\ \mathtt{word},$	$\{\}\ )$	
`real`	$\mapsto$	$(\ \mathtt{real},$	$\{\}\ )$	
`string`	$\mapsto$	$(\ \mathtt{string},$	$\{\}\ )$	
`char`	$\mapsto$	$(\ \mathtt{char},$	$\{\}\ )$	
`list`	$\mapsto$	$(\ \mathtt{list},$	$\{\mathtt{nil} \mapsto (\forall\mathtt{'a}\ .\ \mathtt{'a\ list}, \mathtt{c}),$	
			$\mathtt{::} \mapsto (\forall\mathtt{'a}\ .\ \mathtt{'a} * \mathtt{'a\ list} \to \mathtt{'a\ list}, \mathtt{c})\}\ )$	
`ref`	$\mapsto$	$(\ \mathtt{ref},$	$\{\mathtt{ref} \mapsto (\forall\ \mathtt{'a}\ .\ \mathtt{'a} \to \mathtt{'a\ ref}, \mathtt{c})\}\ )$	
`exn`	$\mapsto$	$(\ \mathtt{exn},$	$\{\}\ )$	

Figure 24: Static TE_0

NONFIX			INFIX		
vid	$\mapsto$	(σ, is)	vid	$\mapsto$	(σ, is)
`ref`	$\mapsto$	$(\forall\ \mathtt{'a}\ .\ \mathtt{'a} \to \mathtt{'a\ ref},\ \mathtt{c})$	Precedence 5, right associative :		
`nil`	$\mapsto$	$(\forall\mathtt{'a}.\ \mathtt{'a\ list},\ \mathtt{c})$	`::`	$\mapsto$	$(\forall\mathtt{'a}.\mathtt{'a} * \mathtt{'a\ list} \to \mathtt{'a\ list},\ \mathtt{c})$
`true`	$\mapsto$	$(\mathtt{bool}, \mathtt{c})$	Precedence 4, left associative :		
`false`	$\mapsto$	$(\mathtt{bool}, \mathtt{c})$	`=`	$\mapsto$	$(\forall\mathtt{''a}.\ \mathtt{''a}\ *\ \mathtt{''a} \to \mathtt{bool}, \mathtt{v})$
`Match`	$\mapsto$	$(\mathtt{exn}, \mathtt{e})$	Precedence 3, left associative :		
`Bind`	$\mapsto$	$(\mathtt{exn}, \mathtt{e})$	`:=`	$\mapsto$	$(\forall\mathtt{'a}.\ \mathtt{'a\ ref}\ *\ \mathtt{'a} \to \{\}, \mathtt{v})$

Note: In type schemes we have taken the liberty of writing $ty_1 * ty_2$ in place of $\{\mathtt{1} \mapsto ty_1, \mathtt{2} \mapsto ty_2\}$.

Figure 25: Static VE_0

D Appendix: The Initial Dynamic Basis

We shall indicate components of the initial basis by the subscript 0. The initial dynamic basis is $B_0 = F_0, G_0, E_0$, where $F_0 = \{\}$, $G_0 = \{\}$ and $E_0 = (SE_0, TE_0, VE_0)$, where $SE_0 = \{\}$, TE_0 is shown in Figure 26 and

$$\begin{aligned} VE_0 = \{&\texttt{=} \mapsto (\texttt{=}, \texttt{v}),\ \texttt{:=} \mapsto (\texttt{:=}, \texttt{v}),\ \texttt{Match} \mapsto (\texttt{Match}, \texttt{e}),\ \texttt{Bind} \mapsto (\texttt{Bind}, \texttt{e}),\\ &\texttt{true} \mapsto (\texttt{true}, \texttt{c}),\ \texttt{false} \mapsto (\texttt{false}, \texttt{c}),\\ &\texttt{nil} \mapsto (\texttt{nil}, \texttt{c}),\ \texttt{::} \mapsto (\texttt{::}, \texttt{c}),\ \texttt{ref} \mapsto (\texttt{ref}, \texttt{c})\}. \end{aligned}$$

tycon	$\mapsto$	$\{vid_1 \mapsto (v_1, is_1), \ldots, vid_n \mapsto (v_n, is_n)\}\quad (n \geq 0)$
unit	$\mapsto$	$\{\}$
bool	$\mapsto$	$\{\texttt{true} \mapsto (\texttt{true}, \texttt{c}),\ \texttt{false} \mapsto (\texttt{false}, \texttt{c})\}$
int	$\mapsto$	$\{\}$
word	$\mapsto$	$\{\}$
real	$\mapsto$	$\{\}$
string	$\mapsto$	$\{\}$
char	$\mapsto$	$\{\}$
list	$\mapsto$	$\{\texttt{nil} \mapsto (\texttt{nil}, \texttt{c}), :: \mapsto (::, \texttt{c})\}$
ref	$\mapsto$	$\{\texttt{ref} \mapsto (\texttt{ref}, \texttt{c})\}$
exn	$\mapsto$	$\{\}$

Figure 26: Dynamic TE_0

E Overloading

Two forms of overloading are available:

- Certain special constants are overloaded. For example, `0w5` may have type `word` or some other type, depending on the surrounding program text;
- Certain operators are overloaded. For example, `+` may have type $\texttt{int} * \texttt{int} \rightarrow \texttt{int}$ or $\texttt{real} * \texttt{real} \rightarrow \texttt{real}$, depending on the surrounding program text;

Programmers cannot define their own overloaded constants or operators.

Although a formal treatment of overloading is outside the scope of this document, we do give a complete list of the overloaded operators and of types with overloaded special constants. This list is consistent with the Basis Library[18].

Every overloaded constant and value identifier has among its types a *default type*, which is assigned to it, when the surrounding text does not resolve the overloading. For this purpose, the surrounding text is no larger than the smallest enclosing structure-level declaration; an implementation may require that a smaller context determines the type.

E.1 Overloaded special constants

Libraries may extend the set T_0 of Appendix C with additional type names. Thereafter, certain subsets of T_0 have a special significance; they are called *overloading classes* and they are:

$$\begin{array}{lcl}
\text{Int} & \supseteq & \{\texttt{int}\} \\
\text{Real} & \supseteq & \{\texttt{real}\} \\
\text{Word} & \supseteq & \{\texttt{word}\} \\
\text{String} & \supseteq & \{\texttt{string}\} \\
\text{Char} & \supseteq & \{\texttt{char}\} \\
\text{WordInt} & = & \text{Word} \cup \text{Int} \\
\text{RealInt} & = & \text{Real} \cup \text{Int} \\
\text{Num} & = & \text{Word} \cup \text{Real} \cup \text{Int} \\
\text{NumTxt} & = & \text{Word} \cup \text{Real} \cup \text{Int} \cup \text{String} \cup \text{Char}
\end{array}$$

Among these, the five first (Int, Real, Word, String and Char) are said to be *basic*; the remaining are said to be *composite*. The reason that the basic classes are specified using $\supseteq$ rather than $=$ is that libraries may extend each of the basic overloading classes with further type names. Special constants are overloaded within each of the basic overloading classes. However, the basic overloading classes must be arranged so that every special constant can be assigned types from at most one of the basic overloading classes. For example, to `0w5` may be assigned type `word`, or some other member of Word, depending on the surrounding text. If the surrounding text does not determine the type of the constant, a default type is used. The default types for the five sets are `int`, `real`, `word`, `string` and `char` respectively.

NONFIX	INFIX
var $\mapsto$ set of monotypes	*var* $\mapsto$ set of monotypes
`abs` $\mapsto$ `realint` $\to$ `realint`	Precedence 7, left associative :
`~` $\mapsto$ `realint` $\to$ `realint`	`div` $\mapsto$ `wordint` $*$ `wordint` $\to$ `wordint`
	`mod` $\mapsto$ `wordint` $*$ `wordint` $\to$ `wordint`
	`*` $\mapsto$ `num` $*$ `num` $\to$ `num`
	`/` $\mapsto$ `Real` $*$ `Real` $\to$ `Real`
	Precedence 6, left associative :
	`+` $\mapsto$ `num` $*$ `num` $\to$ `num`
	`-` $\mapsto$ `num` $*$ `num` $\to$ `num`
	Precedence 4, left associative :
	`<` $\mapsto$ `numtxt` $*$ `numtxt` $\to$ `numtxt`
	`>` $\mapsto$ `numtxt` $*$ `numtxt` $\to$ `numtxt`
	`<=` $\mapsto$ `numtxt` $*$ `numtxt` $\to$ `numtxt`
	`>=` $\mapsto$ `numtxt` $*$ `numtxt` $\to$ `numtxt`

Figure 27: Overloaded identifiers

Once overloading resolution has determined the type of a special constant, it is a compile-time error if the constant does not make sense or does not denote a value within the machine representation chosen for the type. For example, an escape sequence of the form `\u`$xxxx$ in a string constant of 8-bit characters only makes sense if $xxxx$ denotes a number in the range $[0, 255]$.

E.2 Overloaded value identifiers

Overloaded identifiers all have identifier status `v`. An overloaded identifier may be rebound with any status (`v`, `c` and `e`) but then it is not overloaded within the scope of the binding.

The overloaded identifiers are given in Figure 27. For example, the entry

$$\texttt{abs} \mapsto \texttt{realint} \to \texttt{realint}$$

states that `abs` may assume one of the types $\{t \to t \mid t \in \mathrm{RealInt}\}$. In general, the same type name must be chosen throughout the entire type of the overloaded operator; thus `abs` does not have type `real` $\to$ `int`.

The operator `/` is overloaded on all members of Real, with default type `real`$*$`real` $\to$ `real`. The default type of any other identifier is that one of its types which contains the type name `int`. For example, the program `fun double(x) = x + x;` declares a function of type `int` $*$ `int` $\to$ `int`, while `fun double(x:real) = x + x;` declares a function of type `real` $*$ `real` $\to$ `real`.

The dynamic semantics of the overloaded operators is defined in [18].

G Appendix: What is New?

This appendix gives an overview of how the present Definition differs from the 1990 Definition of Standard ML[42]. For the purpose of this appendix, we write SML '90 for the language defined by the 1990 Definition and SML '96 for the present language. For each major change, we give its rationale and an overview of its practical implications. Also, the index (page 94 ff.) may be used for locating changes.

G.1 Type Abbreviations in Signatures

There are cases of type sharing which cannot be expressed in SML '90 signatures although they arise in structures. For example, there is no SML '90 signature which precisely describes the relationship between `s` and `t` in

```
structure a =
 struct
   datatype s = C
   type t = s * s
 end
```

In SML '96, one can write *type abbreviations* in signatures, e.g.,

```
signature A =
 sig
   type s
   type t = s * s
 end
```

The need for type abbreviations in signatures was clear when SML '90 was defined. However, type abbreviations were not included since, in the presence of both structure sharing and type abbreviations, principal signatures do not exist[41] – and the SML '90 Definition depended strongly upon the notion of principal signature. Subsequently, Harper's and Lillibridge's work on translucent sums[22] and Leroy's work on modules[30] showed that, in the absence of structure sharing and certain other features of the SML '90 signatures, type abbreviations in signatures are possible. Type abbreviations in signatures were implemented by David MacQueen in *SML/NJ* 0.93 and by Xavier Leroy in *Caml Special Light*[31].

In SML '96, structure sharing has been removed (see Section G.3 below). Type abbreviations are not included directly, but they arise as a derived form, as follows. First, a new form of signature expression is allowed:

$$sigexp\ \texttt{where type}\ tyvarseq\ longtycon\ \texttt{=}\ ty$$

Here *longtycon* has to be specified by *sigexp*. The type expression *ty* may refer to type constructors which are present in the basis in which the whole signature expression is elaborated, but not to type constructors specified in *sigexp*.

The effect of the `where type` is, roughly speaking, to instantiate *longtycon* to *ty*. For example, the following sequence of declarations is legal:

```
signature SIG1 = sig type t; val x: t end;
signature SIG2 = SIG1 where type t = int*int;
structure S1: SIG1 = struct type t = real; val x = 1.0 end;
structure S2: SIG2 = struct type t = int*int; val x = (5, 7) end;
```

Next, a type abbreviation is a derived form. For example, `type u = t*t` is equivalent to `include sig type u end where type u = t*t` . In SML '96 it is allowed to `include` an arbitrary signature expression, not just a signature identifier.

G.2 Opaque Signature Matching

In imposing a signature on a structure, one often wants the types of the resulting structure to be "abstract" in order to hide their implementation. (Signature matching in SML '90 hides components, but does not hide type sharing.) MacQueen originally suggested and implemented an `abstraction` declaration for this purpose[32]. In the Commentary[41] it was pointed out that the issue is the semantics of matching. SML '96 provides two kinds of matching, as new forms of structure expression:

$$\mathit{strexp}\ \ \texttt{:}\ \ \ \mathit{sigexp}$$

$$\mathit{strexp}\ \ \texttt{:>}\ \mathit{sigexp}$$

The first (`:`) is the SML '90 signature matching; the second (`:>`) is *opaque matching*. Opaque matching can be applied to the result structure of a functor; thus it is more general than MacQueen's `abstraction` declaration. In CAML Special Light, all signature matching is opaque.

With opaque matching, types in the resulting structure will be abstract, to precisely the degree expressed in *sigexp*. Thus

```
signature Sig =
   sig
     type t = int
     val x: t
     type u
     val y: u
   end;
structure S1:> Sig =
 struct type t = int
        val x = 3
        type u = real
        val y = 3.0
 end
val r = S1.x + 1
```

is legal, but a subsequent declaration `val s = S1.y + 1.5` will fail to elaborate. Similarly, consider the functor declaration:

```
functor Dict(type t; val leq: t*t->int):>
            sig type u = t*t
                type 'a dict
            end =
  struct
    type u = t*t
    type 'a dict = (t * 'a) list
  end
```

When applied, `Dict` will propagate the identity of the type `t` from argument to result, but it will produce a fresh `dict` type upon each application.

Types which are specified as "abstract" in a opaque functor result signature give rise to generation of fresh type names upon each application of the functor, even if the functor body is a constant structure. For example, after the elaboration of

```
structure A = struct type t = int end
functor f():> sig type t end = A
structure B = f()
      and C = f();
```

the two types `B.t` and `C.t` are different.

G.3 Sharing

Structure sharing is a key idea in MacQueen's original Modules design[32]. The theoretical aspects of structure sharing have been the subject of considerable research attention[24,53, 1,55,35]. However, judging from experience, structure sharing is not often used in its full generality, namely to ensure identity of values. Furthermore, experience from teaching suggests that the structure sharing concept is somewhat hard to grasp. Finally, the semantic accounts of structure sharing that have been proposed are rather complicated.

The static semantics of SML '96 has no notion of structure sharing. However, SML '96 does provide a weaker form of structure sharing constraints, in which structure sharing is regarded as a derived form, equivalent to a collection of type sharing constraints.

G.3.1 Type Sharing

In SML '90, a type sharing constraint `sharing type` $longtycon_1$ `=` $\cdots$ `=` $longtycon_n$ was an admissible form of specification. In SML '96 such a constraint does not stand by itself as a specification, but may be used to qualify a specification. Thus there is a new form of specification, which we shall call a *qualified* specification:

$$spec\ \texttt{sharing type}\ longtycon_1\ \texttt{=}\ \cdots\ \texttt{=}\ longtycon_n$$

Here the long type constructors have to be specified by *spec*. The type constructors may have been specified by `type`, `eqtype` or `datatype` specifications, or indirectly through signature identifiers and `include`. In order for the specification to be legal, all the type constructors must denote flexible type names. More precisely, let B be the basis in which the qualified specification is elaborated. Let us say that a type name t is *rigid (in B)* if $t \in T$ of B and that t is *flexible (in B)* otherwise. For example `int` is rigid in the initial basis and every datatype declaration introduces additional rigid type names into the basis. For the qualified specification to elaborate in basis B, it is required that each $longtycon_i$ denotes a type name which is flexible in B. In particular, no $longtycon_i$ may denote a type function which is not also a type name (e.g., a *longtycon* must not denote $\Lambda().\mathtt{s} * \mathtt{s}$).

For example, the two signature expressions

```
sig                          sig
  type s                       type s
  type t                       datatype t = C
  sharing type s = t           sharing type s = t
end                          end
```

are both legal. By contrast, the signature expressions

```
sig                          sig
  type s                       type s = int
  type t = s*s                 datatype t = C
  sharing type s = t           sharing type s = t
end                          end
```

are both illegal.

G.3.2 The equality attribute of specified types

If *spec* `sharing type` $longtycon_1$ `=` $\cdots$ `=` $longtycon_n$ elaborates successfully, then all n type constructors will thereafter denote the same type name. This type name will admit equality, if *spec* associates an equality type name with one of the type constructors. Thus

```
eqtype t
type u
sharing type t = u
```

is legal and both `t` and `u` are equality types after the sharing qualification. The mechanism for inferring equality attributes for datatype specifications is the same as for inferring equality attributes for datatype declarations. Thus the specification

```
datatype answer = YES | NO
datatype 'a option = Some of 'a | None
```

specifies two equality types. Every specification of the form `datatype` *datdesc* introduces one type name for each type constructor described by *datdesc*. The equality attribute of such a type name is determined at the point where the specification occurs. Thus, in

```
type s
datatype t  = C of s
```

the type name associated with `t` will not admit equality, even if `s` later is instantiated to an equality type. Type names associated with datatype specifications can be instantiated to other type names by subsequent `type sharing` or `where type` qualifications. In this case, no effort is made to ban type environments that do not respect equality. For example,

```
sig
  eqtype s
  datatype t = C of int -> int
  sharing type s = t
end
```

is legal in SML '96, even though it cannot be matched by any real structure.

G.3.3 Structure Sharing

For convenience, structure sharing constraints are provided, but only as a shorthand for type sharing constraints. There is a derived form of specification

$$\mathit{spec}\ \texttt{sharing}\ \mathit{longstrid}_1\ \texttt{=}\ \cdots\ \texttt{=}\ \mathit{longstrid}_k \qquad (k \geq 2)$$

Here *spec* must specify $\mathit{longstrid}_1, \ldots, \mathit{longstrid}_k$. The equivalent form consists of *spec* qualified by all the type sharing constraints

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

$(1 \leq i < j \leq k)$ such that both $\mathit{longstrid}_i.\mathit{longtycon}$ and $\mathit{longstrid}_j.\mathit{longtycon}$ are specified by *spec*.

In SML '90, structure sharing constraints are transitive, but in SML '96 they are not. For example,

```
structure A: sig type t end
structure B: sig end
structure C: sig type t end
sharing A=B=C
```

induces type sharing on `t`, whereas

```
structure A: sig type t end
structure B: sig end
structure C: sig type t end
sharing A=B sharing B=C
```

induces no type sharing. Thus a structure sharing constraint in some cases induces less sharing in SML '96 than in SML '90.

Next, SML '96 does not allow structure sharing equations which refer to "external" structures. For example, the program

```
structure A= struct end;
signature SIG = sig structure B : sig end
                    sharing A = B
                end;
```

is not legal in SML '96, because the sharing constraint now only qualifies the specification `structure B: sig end`, which does not specify `A`. Thus not all legal SML '90 signatures are legal in SML '96.

The removal of structure sharing has a dramatic simplifying effect on the semantics. Most importantly, the elaboration rules can be made monogenic (i.e., "deterministic"), up to renaming of new type names. The need for the notion of principal signature (and even equality-principal signature) disappears. The notions of structure name, structure consistency and well-formed signature are no longer required. The notion of cover can be deleted. Only one kind of realisation, namely type realisation, remains. The notion of type-explication has been removed, since it can be proved that signatures automatically are type-explicit in the revised language.

G.4 Value Polymorphism

The imperative types of SML '90 were somewhat subtle, and they propagated into signatures in an unpleasant way. Experiments on existing code suggest that the power of imperative types is rarely used fully and that *value polymorphism*, which can in fact be seen as a restriction of the imperative type discipline, usually suffices[57]. With value polymorphism, there is only one kind of type variable. The definition of non-expansive expressions (see G.13 below) is relaxed to admit more expressions. In a declaration

`val` x `=` *exp*

the variable x will only be given a non-trivial polymorphic type scheme (i.e., a type scheme which is not also a type) if *exp* is non-expansive. This applies even if there is no application of `ref` in the entire program.

Example: in the declaration `val x = [] @ []`, x can be assigned type `'a list`, but not the type scheme $\forall$`'a.'a list` (since `[] @ []` is an expansive expression). Consequently, `(1::x, true::x)` will not elaborate in the scope of the declaration. Also, if the declaration appears at top level, the compiler may refuse elaboration due to a top-level free type variable (see G.8). Thus the top-level phrase `[] @ []` may fail, since it abbreviates `val it = [] @ []`. But of course it will not fail if a monotype is explicitly ascribed, e.g. `[] @ []:int list`.

On the other hand, in `fun f() = [] @ []` (or `val f = fn () => [] @ []`), `f` can be assigned type scheme $\forall$`'a.unit` $\to$ `'a list` so that, for example, `(1::f(),true::f())`

elaborates. This transformation (η-conversion) often gives the desired polymorphism. But beware that η-conversion can change the meaning of the program, if *exp* does not terminate or has side-effects.

G.5 Identifier Status

The 1990 Definition treated identifier status informally (in Section 2.4); a fuller definition was given in the Commentary[41, Appendix B]. However, some problems with the handling of exception constructors remained[27, Sect. 10.3].

In the present document, we have collapsed the three identifier classes Var, ExCon and Con into a single class, VId, of *value identifiers*. The semantic objects VE previously called variable environments are replaced by *value environments*. A value environment maps value identifiers to pairs of the form (o, is), where o is some semantic object and is is an *identifier status* ($is \in \{\mathtt{v}, \mathtt{c}, \mathtt{e}\}$) indicating whether the identifier should be regarded as a value variable (`v`), a value constructor (`c`) or an exception constructor (`e`). These changes have been carried out both in the static and in the dynamic semantics, for both Core and Modules. Thus the assignment of identifier status is incorporated formally in the present Definition.

The definition of enrichment has been modified to allow an identifier that has been specified as a value to be matched by a value constructor or an exception constructor. However, a specification of a value or exception constructor must be matched by a value or exception constructor, respectively.

Thus, the status descriptor says more than just what the lexical status of the identifier is — it is a statement about the value in the corresponding dynamic environment: if the status of *id* in the static environment is `c`, then the value in a matching dynamic environment must be a value constructor. Similarly, if the status of *id* in the static environment is `e`, then the value in a matching dynamic environment must be an exception name. If the status of *id* is just `v`, however, the corresponding value in the dynamic environment can be any kind of value (of the appropriate type), including a value constructor and an exception name.

The exception environment (EE) has been deleted from the semantics, since it is no longer required for the definition of enrichment. Also, the constructor environment CE in the static semantics has been replaced by a value environment in which every identifier has status `c`.

The new handling of identifier status admits some `val rec` declarations that were illegal in SML '90 (see the comment to Rule 26).

G.6 Replication of Datatypes

SML '96 allows *datatype replication*, i.e. declarations and specifications of the form

`datatype` *tycon* `=` `datatype` *longtycon*

When elaborated, this binds type constructor *tycon* to the entire type structure (value constructors included) to which *longtycon* is bound in the context. Datatype replication does not generate a new datatype: the original and the replicated datatype share.

Here is an example of a use of the new construct:

```
signature MYBOOL =
sig
  type bool
  val xor: bool * bool -> bool
end;
structure MyBool: MYBOOL =
struct
  datatype bool = datatype bool (* from the initial basis *)
  fun xor(true, false) = true
    | xor(false, true) = true
    | xor _ = false
end;
val x = MyBool.xor(true, false);
```

Here `MyBool.xor(true, false)` evaluates to `true`. Note the use of transparent signature matching; had opaque matching been used instead, the declaration of `x` would not have elaborated.

A datatype replication implicitly introduces the value constructors of *longtycon* into the current scope. This is significant for signature matching. For example, the following program is legal:

```
datatype t0 = C;
structure A : sig type t val C: t end =
  struct
    datatype t = datatype t0
  end;
```

Note that `C` is specified as a value in the signature; the datatype replication copies the value environment of `t0` into the structure and that is why the structure contains the required `C` value.

To make it possible for datatype replication to copy value environments associated with type constructors, the dynamic semantics has been modified so that environments now contain a *TE* component (see Figure 13, page 38). Further, in the dynamic semantics of modules, the $\downarrow$ operation, which is used for cutting down structures when they are matched against signatures, has been extended to cover the *TE* component (see page 48). In the above example, the value environment assigned to `A.t` will be empty, signifying that the type has no value constructors. Had the signature instead been

```
sig datatype t val C: t end
```

then the signature matching would have assigned `A.t` a value environment with domain {`C`}, indicating that `A.t` has value constructor `C`.

When the datatype replication is used as a specification, *longtycon* can refer to a datatype which has been introduced either by declaration or by specification. Here is an example of the former:

```
datatype t = C | D;
signature SIG =
sig
  datatype t = datatype t (* replication is not recursive! *)
  val f: t -> t
end
```

G.7 Local Datatypes

This change is concerned with expressions of the form `let` *dec* `in` *exp* `end` in which *dec* contains a `datatype` declaration. Let us refer to such a datatype declaration as a *local* datatype declaration. There are two reasons why changes to the handling of local datatype declarations are necessary.

The first is that the rule given for elaboration of `let`-expressions in the 1990 Definition is unsound[27]; the problem has to do with the ability to export type names of locally declared datatypes out of scope.

The second is that the static semantics relies on the following invariant about all contexts, C, which arise in elaboration from the initial basis:

$$\mathrm{tynames}\, C \subseteq T \text{ of } C$$

This invariant is used, for example, in the rule for elaborating datatype declarations, where type names are picked "fresh" with respect to T of C. As pointed out by Kahrs, the second premise of rule 16 in the 1990 Definition violates the above invariant.

To solve the first problem, the rule for elaborating `let`-expressions (rule 4 in the present document) has been provided with a side-condition which prevents the type of *exp* from containing type names generated by *dec*. For example,

```
let datatype t = C in C end
```

was legal SML '90 but is not legal SML '96.

To solve the second problem, a side-condition has been added in the rule for matches and the rule for `val rec` (rules 14 and 26 of the present document). As a consequence, again fewer programs elaborate. For example, the expression

```
fn x => let datatype t = C
            val _ = if true then x else C
        in 5
        end
```

is not legal SML '96, although it was legal SML '90.

G.8 Principal Environments

In SML '90, the elaboration rule which allows any *dec* to appear as a *strdec* is

$$\frac{C \text{ of } B \vdash dec \Rightarrow E \quad E \text{ principal for } dec \text{ in } (C \text{ of } B)}{B \vdash dec \Rightarrow E}$$

The side-condition forces the type scheme in E to be as general as possible. However, this side-condition would be undesirably restrictive in SML '96, since the new definition of the Clos operation admits less polymorphism than the one used in SML '90. For example, neither

```
val f = (fn x => x)(fn x => x)
structure A  = struct end
val y = f 7
```

(where the presence of the `structure` declaration forces each `val` declaration to be parsed as a *strdec*), nor

```
structure A: sig val f: int -> int end =
  struct
     val f = (fn x => x)(fn x => x)
  end
```

would be legal in SML '96, if the side-condition were enforced. (A type-checker may at first infer the type `'a` $\rightarrow$ `'a` from the declaration of `f`, but since `(fn x => x)(fn x => x)` is expansive, the generalisation to $\forall$`'a.'a` $\rightarrow$ `'a` is not allowed.) By dropping the side-condition, it becomes possible to have the textual context of a structure-level declaration constrain free type variables to monotypes. Thus both the above examples can be elaborated.

Rather than lifting the notion of principal environments to the modules level, we have chosen to drop the requirement of principality. Since the notion of principal environments is no longer used in the rules, even the definition of principal environments has been removed. In practice, however, type checkers still have to infer types that are as general as possible, since implementations should not reject programs for which successful elaboration is possible.

In order to avoid reporting free type variables to users, rules 87 and 89 require that the environment to which a *topdec* elaborates must not contain free type variables. It is possible to satisfy this side-condition by replacing such type variables by arbitrary monotypes; however, implementers may instead choose to refuse elaboration in such situations.

G.9 Consistency and Admissibility

The primary purpose of consistency in SML '90 was to allow a very simple elaboration rule for structure sharing. A secondary purpose was to ban any signature which, because it specifies a datatype in inconsistent ways (e.g. with different constructors), can never be matched. With the removal of structure sharing, the primary purpose of consistency has gone away. In our experience, the secondary purpose has turned out not to be very significant in practice. Textual copying of datatype specifications in different signatures is best avoided, since changes in the datatype will have to be done several places. In practice, it is better to specify a datatype in one signature and then access it elsewhere using structure specifications or `include`. In SML '90 one could specify sharing between a datatype specification and an external (i.e., declared) datatype, and a consistency check was useful in this case. But in SML '96 this form of sharing is not allowed, so there remains no strong reason for preserving consistency; therefore it has been dropped.

In SML '90, admissibility was imposed partly to ensure the existence of principal signatures (which are no longer needed) and partly to ban certain unmatchable signatures. In SML '90, admissibility was the conjunction of well-formedness, cycle-freedom and consistency. Cycle-freedom is no longer relevant, since there is no structure sharing. We have already discussed consistency. Well-formedness of signatures is no longer relevant, but the notion of well-formed type structures is still relevant. It turns out that well-formedness only needs to be checked in one place (in rule 64). Otherwise, well-formedness is preserved by the rules (in a sense which can be made precise). Thus one can avoid a global well-formedness requirement and dispense with admissibility. This we have done.

G.10 Special Constants

The class of special constants has been extended with `word` and `char` constants and with hexadecimal notation. Also, there are additional escape sequences in strings and support for UNICODE characters. See Section 2.2.

G.11 Comments

A clarification concerning unmatched comment brackets was presented in the Commentary; subsequently, Stefan Kahrs discovered a problem with demanding that an unmatched `*)` be reported by the compiler. In SML '96, we therefore simply demand that an unmatched `(*` must be reported by the compiler.

G.12 Infixed Operators

The rules for associativity of infix operators at the same level of precedence have been modified, to avoid confusion between right- and left-associative operators with the same binding precedence (see Section 2.6).

G.13 Non-expansive Expressions

The class of non-expansive expressions (Section 4.7) has been extended, to compensate for the loss of polymorphism which value polymorphism entails.

G.14 Rebinding of built-in identifiers

In SML '96, no *datbind*, *valbind* or *exbind* may bind `true`, `false`, `nil`, `::` or `ref` and no *datbind* or *exbind* may bind `it` (Section 2.9). Similarly, no *datdesc*, *valdesc* or *exdesc* may describe `true`, `false`, `nil`, `::` or `ref` and no *datdesc* or *exdesc* may describe `it` (Section 3.5). These changes are made in order to fix the meaning of derived forms and to avoid ambiguity in the handling of `ref` in the dynamic semantics of the Core.

G.15 Grammar for Modules

There are several new derived forms for modules, see Appendix A (Figures 18 and 19). The grammar for *topdec* has been modified, so that there is no longer any need to put semicolons at the end of signature and functor declarations. Empty and sequential signature and functor declarations have been removed, as they no longer serve any purpose. SML '96 has neither functor signature expressions nor functor specifications, since they could not occur in programs and did not gain wide acceptance.

G.16 Closure Restrictions

Section 3.6 of the 1990 Definition has been deleted.

G.17 Specifications

`open` and `local` specifications have been criticised on the grounds of programming methodology[4]. Also, they are no longer needed for defining the derived forms for functors and they conflict with a desire to have all signatures be type-explicit.

SML '96 therefore admits neither `open` nor `local` in specifications. Moreover, sequential specifications must not specify the same identifier twice. As a consequence, the definition of type-explication has been removed: type-explication is automatically preserved by elaboration (if one starts in the initial basis) so there is no need to impose type-explicitness explicitly.

G.18 Scope of Explicit Type Variables

A binding construct for explicit type variables has been introduced at `val` and `fun` (see Figure 21). For example, one can declare the polymorphic identity function by

```
fun 'a id(x:'a) = x
```

There is no requirement that all explicit type variables be bound by this binding construct. For those that are not, the scope rules of the 1990 Definition apply. The explicit binding construct has no impact on the dynamic semantics. In particular, there are no explicit type abstractions or applications in the dynamic semantics.

G.19 The Initial Basis

To achieve a clean interface to the new Standard ML Basis Library[18], the initial basis (Appendices C and D) has been cut down to a bare minimum. The present Definition only provides what is necessary in order to define the derived forms and special constants of type int, real, word, char and string. The following identifiers are no longer defined in the initial basis: <>, ^, !, @, Abs, arctan, chr, Chr, close_in, close_out, cos, Diff, Div, end_of_stream, exp, Exp, explode, floor, Floor, implode, input, instream, Interrupt, Io, ln, Ln, lookahead, map, Mod, Neg, not, real (the coercion function), rev, sin, size, sqrt, Sqrt, std_in, std_out, Sum, output, outstream, Prod, Quot. The corresponding basic values have also been deleted.

G.20 Overloading

The Standard ML Basis Library[18] rests on an overloading scheme for special constants and pre-defined identifiers. We have adopted this scheme (see Appendix E).

G.21 Reals

real is no longer an equality type and real constants are no longer allowed in patterns. The Basis Library provides IEEE equality operations on reals.