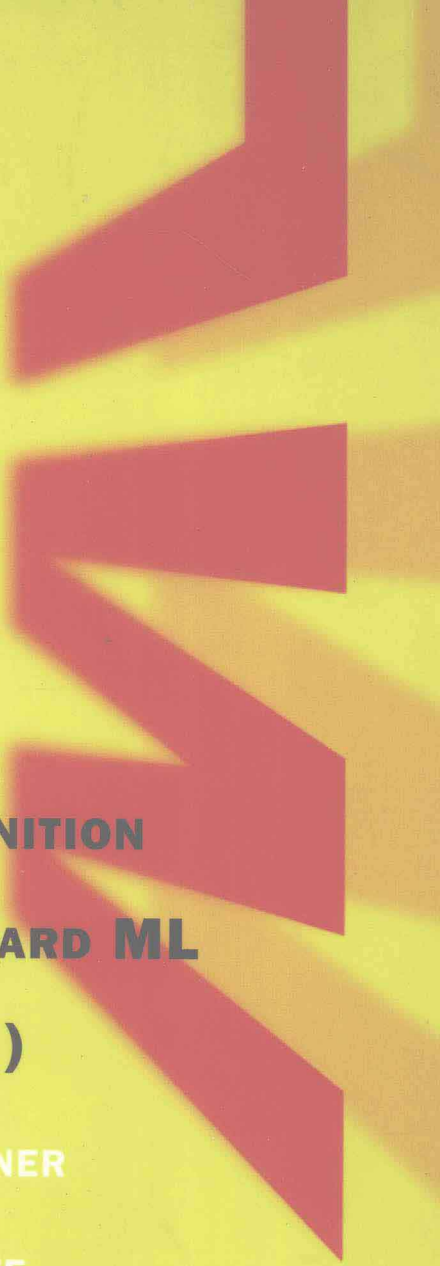

The Definition of Standard ML (Revised)

Robin Milner

Mads Tofte

Robert Harper

David MacQueen

The Definition of Standard ML (Revised)

Robin Milner, Mads Tofte, Robert Harper and David MacQueen

The MIT Press
Cambridge, Massachusetts
London, England

Printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

The definition of standard ML: revised / Robin Milner ... et al.
p. cm.
Includes bibliographical references and index.
ISBN 0-262-63181-4 (alk. paper)
1. ML (Computer program language) I. Milner, R. (Robin), 1934-
QA76.73.M6D44 1997
005.13'3—dc21

97-59
CIP

The Definition of Standard ML

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, particularly 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. We also briefly describe the recent revisions to the 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

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 $P_1, \ldots, P_n$.

We have written the Definition in a form suggested by the previous remarks. That is, we have defined our semantic objects in mathematical notation which is completely independent of Standard ML, and we have developed just enough of their theory to give sense to our rules of evaluation.

Following another suggestion above, we have factored our task by describing *abstract* evaluation – the inference and checking of types and signatures (which can be done at compile-time) – completely separately from *concrete* evaluation. It really is a factorisation, because a *full* value in all its glory – you can think of it as a concrete object with a type attached – never has to be presented.

The Revision of Standard ML

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 changes that have been made to the 1990 Definition are enumerated in Appendix G.

The resulting document is, we hope, valuable as the essential point of reference for Standard ML. If it is to play this role well, it must be supplemented by other literature. Many expository books have already been written, and this Definition will be useful as a background reference for their readers. We became convinced, while writing the 1990 Definition, that we could not discuss many questions without making it far too long. Such questions are: Why were certain design choices made? What are their implications for programming? Was there a good alternative meaning for some constructs, or was our hand forced? What different forms of phrase are equivalent? What is the proof of certain claims? Many of these questions are not answered by pedagogic texts either. We therefore wrote a Commentary on the 1990 Definition to assist people in reading it, and to serve as a bridge between the Definition and other texts. Though in part outdated by the present revision, the Commentary still largely fulfils its purpose.

There exist several textbooks on programming with Standard ML[45,44,56,50]. The second edition of Paulson's book[45] conforms with the present revision.

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

Robin Milner Mads Tofte Robert Harper David MacQueen

November 1996

Contents

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	
Static Semantics	Section 4	Section 5	Section 8
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 \cdots f$\	This sequence is ignored, where $f \cdots f$ 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` ⋯, 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.