9/85

The Standard ML Core Language (Revised)

Robin Milner

University of Edinburgh

1. Introduction

1.1 How this proposal evolved

ML is a strongly typed functional programming language, which has been used by a number of people for serious work during the last few years [1]. At the same time HOPE, designed by Rod Burstall and his group, has been similarly used [2]. The original DEC-10 ML was incomplete in some ways, redundant in others. Some of these inadequacies were remedied by Cardelli in his VAX version; others could be put right by importing ideas from HOPE.

In April '83, prompted by Bernard Sufrin, I wrote a tentative proposal to consolidate ML, and while doing so became convinced that this consolidation was possible while still keeping its character. The main strengthening came from generalising the "varstructs" of ML - the patterns of formal parameters - to the patterns of HOPE, which are extendible by the declaration of new data types. Many people immediately discussed the initial proposal. It was extremely lucky that we managed to have several separate discussions, in large and small groups, in the few succeeding months; we could not have chosen a better time to do the job. Also, Luca Cardelli very generously offered to freeze his detailed draft ML manual [3] until this proposal was worked out.

The proposal went through a second draft, on which there were further discussions. The results of these discussions were of two kinds. First, it became clear that two areas were still unsettled: Input/Output, and Modules for separate compilation. Second, many points were brought up about the remaining core of the language, and these were almost all questions of fine detail. The conclusion was rather clear; it was obviously better to present at first a definition of a Core language without the two unsettled areas. This course was further justified by the fact that the Core language appeared to be almost completely unaffected by the choice of Input/Output primitives and of separate compilation constructs. Also, there were already strong proposals, from Cardelli and MacQueen respectively, for these two vital facilities.

A third draft [4] of the Core language was discussed in detail in a design meeting at Edinburgh in June '84, attended by nine of the people mentioned below; several points were ironed out, and the outcome was reported in [5]. The meeting also looked in detail at the MacQueen Modules proposal and the Cardelli Input/Output proposal, and agreed on their essentials.

During the ensuing year, having an increasingly firm design of MacQueen's Modules, we were able to assess the language as a whole. The Modules proposal, which is the most adventurous part of the language, reached a state of precise definition. At a final design meeting, which was held in Edinburgh in May 1985 and attended by fifteen people (including twelve named below), the Modules design was discussed and warmly accepted; it appears as [6]. We also took advantage of the meeting to tidy up the Core language, and to settle finally the primitives for Input/Output. The final Core language design is presented in this document; the Input/Output facilities are detailed in [7].

The main contributors to Standard ML, through their work on ML and on HOPE, are:

Rod Burstall, Luca Cardelli, Michael Gordon, David MacQueen,
Robin Milner, Lockwood Morris, Malcolm Newey, Christopher Wadsworth.

The language also owes much to criticisms and suggestions from many other people: Guy Cousineau, Bob Harper, Jim Hook, Gerard Huet, Dave Matthews, Robert

Milne, Kevin Mitchell, Brian Monahan, Peter Mosses, Alan Mycroft, Larry Paulson, David Park, David Rydeheard, Don Sannella, David Schmidt, John Scott, Stefan Sokolowski, Bernard Sufrin, Philip Wadler. Most of them have expressed strong support for the design; any inadequacies which remain in the Core Language are my fault, but I have tried to represent the consensus.

1.2 Design principles

The proposed ML is not intended to be the functional language. There are too many degrees of freedom for such a thing to exist: lazy or eager evaluation, presence or absence of references and assignment, whether and how to handle exceptions, types-as-parameters or polymorphic type-checking, and so on. Nor is the language or its implementation meant to be a commercial product. It aims to be a means for propagating the craft of functional programming and a vehicle for further research into the design of functional languages.

The over-riding design principle is to restrict the Core language to ideas which are simple and well-understood, and also well-tried - either in previous versions of ML or in other functional languages (the main other source being HOPE, mainly for its argument-matching constructs). One effect of this principle has been the omission of polymorphic references and assignment. There is indeed an elegant and sound scheme for polymorphic assignment worked out by Luis Damas, and described in his Edinburgh PhD thesis; however, it may be susceptible to improvement with further study. Meanwhile there is the advantage of simplicity in keeping to the well-understood polymorphic type-checking discipline which derives from Curry's Combinatory Logic via Hindley.

A second design principle is to generalise well-tried ideas where the generalisation is apparently natural. This has been applied in generalising ML "varstructs" to HOPE patterns, in broadening the structure of declarations (following Cardelli's declaration connectives which go back to Robert Milne's Ph.D. Thesis) and in allowing exceptions which carry values of arbitrary polymorphic type. It should be pointed out here that a difficult decision had to be made concerning HOPE's treatment of data types - present only in embryonic form in the original ML - and the labelled records and variants which Cardelli introduced in his VAX version. Each treatment has advantages which the other lacks; each is well-rounded in its own terms. Though a combination of these features was seen to be possible, it seemed at first (to me, but some disagreed!) to entail too rich a language. Thus the HOPE treatment alone was adopted in [5]. However, at the design meeting of June '84 it was agreed to experiment with at least two different ways of adding labelled records to the Core as a smooth extension. The outcome - decided at the May '85 meeting - is the inclusion of a form of labelled records (but not variants) nearly identical to Cardelli's, and its marriage with the HOPE constructions now appears harmonious.

A third principle is to specify the language completely, so that programs will port between correct implementations with minimum fuss. This entails, first, precise concrete syntax (abstract syntax is in some senses more important - but we do not all have structure editors yet, and humans still communicate among themselves in concrete syntax!); second, it entails exact evaluation rules (e.g. we must specify the order of evaluation of two expressions, one applied to the other, because of side-effects and the exception mechanism). At a level which is not fully formal, this document and its sister reports on Modules and on Input/Output constitute a complete description; however, we intend to augment them both with a formal definition and with tutorial material.

1.3 An example

The following declaration illustrates some constructs of the Core language. A longer expository paper should contain many more examples; here, we hope only to draw attention to some of the less familiar ideas.

The example sets up the abstract type 'a dictionary , in which each entry associates an item (of arbitrary type 'a) with a key (an integer). Besides the null dictionary, the operations provided are for looking up a key, and for adding a new entry which overrides any old entry with the same key. A natural representation is by a list of key-item pairs, ordered by key.

```
abstype 'a dictionary =
   dict of (int * 'a)list                          (* dict is the datatype    *)
                                                   (*  constructor, available *)
with                                               (*  only in the with part. *)
   val nulldict = dict nil
                                                   (* The function lookup may *)
   exception lookup : unit                         (*    raise an exception.  *)

   fun lookup (key:int)                            (* 'a is the result type.  *)
              (dict entrylist :'a dictionary) :'a  =
      let fun search nil = raise lookup            (* An auxiliary clausal    *)
            | search ((k,item)::entries) =         (*  function declaration.  *)
                if key=k then item
                else if key<k then raise lookup
                else search entries
      in search entrylist
      end

   fun enter (newentry as (key, item:'a))          (* A layered pattern.      *)
             (dict entrylist) :'a dictionary  =
      let fun update nil = [ newentry ]            (* A singleton list.       *)
            | update ((entry as (k,_))::entries) =
                if key=k then newentry::entries
                else if key<k then newentry::entry::entries
                else entry::update entries
      in dict(update entrylist)
      end
end                                                (* end of dictionary *)
```

After the declaration is evaluated, five identifier bindings are reported, and recorded in the top-level environment. They are the type binding of dictionary, the exception binding of lookup, and three typed value bindings:

```
nulldict : 'a dictionary
lookup : int -> 'a dictionary -> 'a
enter : int * 'a -> 'a dictionary -> 'a dictionary
```

The layered pattern construct "as" was first introduced in HOPE, and yields both brevity and efficiency. The discerning reader may be able to find one furtner use for it in the declaration.

2. The bare language

2.1 Discussion

It is convenient to present the language first in a bare form, containing enough on which to base the semantic description given in Section 3. Things omitted from the bare language description are:

(1) Derived syntactic forms, whose meaning derives from their equivalent forms in the bare language (Section 6);

(2) Directives for introducing infix identifier status (Section 4);

(3) Standard bindings (Section 5);

(4) References and equality (Section 7);

(5) Type-checking (Section 9).

The principal syntactic objects are expressions and declarations. The composite expression forms are application, record formation, raising and handling exceptions, local declaration (using let) and function abstraction.

Another important syntactic class is the class of patterns; these are essentially expressions containing only variables and value constructors, and are used to create value bindings. Declarations may declare value variables (using value bindings), types with associated constructors or operations (using type and datatype bindings), and exceptions (using exception bindings). Apart from this, one declaration may be local to another (using local), and a sequence of declarations is allowed as a single declaration.

An ML program is a series of declarations, called top-level declarations,

dec1 ; ... decn ;

each terminated by a semicolon (where each deci is not itself of the form "dec ; dec'"). In evaluating a program, the bindings created by dec1 are reported before dec2 is evaluated, and so on. In the complete language, an expression occurring in place of any deci is an abbreviated form (see Section 6.2) for a declaration binding the expression value to the variable "it"; such expressions are called top-level expressions.

The bare syntax is in Section 2.8 below; first we consider lexical matters.

2.2 Reserved words

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

abstype and andalso as case do datatype
else end exception fn fun handle if in
infix infixr let local nonfix of op open
orelse raise rec then type val with while

( ) [ ] { } , : ; ... | || = => _ ?

2.3 Special constants

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

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

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

\n	A single character interpreted by the system as end-of-line.
\t	Tab.
\^c	The control character c, for any appropriate c.
\ddd	The single character with ASCII code ddd (3 decimal digits).
\"	"
\\	\
\f__f\	This sequence is ignored, where f__f stands for a sequence of one or more formatting characters (a subset of the non-printable characters including at least space, tab, newline, formfeed). This allows one to write long strings on more than one line, by writing \ at the end of one line and at the start of the next.

2.4 Identifiers

Identifiers are used to stand for six different syntax classes which, if we had a large enough character set, would be disjoint:

value variables	(var)	type variables	(tyvar)
value constructors	(con)	type constructors	(tycon)
exception names	(exn)	record labels	(lab)

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

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

In either case, however, reserved words are excluded. This means that for example ? and | are not identifiers, but ?? and |=| are identifiers. The only exception to this rule is that the symbol =, which is a reserved word, is also allowed as an identifier to stand for the equality predicate (see Section 7.2). The identifier = may not be rebound; this precludes any syntactic ambiguity.

A type variable (tyvar) may be any alphanumeric identifier starting with a prime. The other five classes (var, con, exn, tycon, lab) are represented by identifiers not starting with a prime; the class lab is also extended to include the numeric labels #1, #2, #3, ___ .

Type variables are therefore disjoint from the other five classes. Otherwise, the syntax class of an occurrence of identifier id is determined thus:

(1) At the start of a component in a record type, record pattern or record expression, id is a record label.
(2) Elsewhere in types id is a type constructor, and must be within the scope of the type binding or datatype binding which introduced it.
(3) Following exception, raise or handle, or in the context "exception exn = id", id is an exception name.
(4) Elsewhere, id is a value constructor if it occurs in the scope of a datatype binding which introduced it as such, otherwise it is a value variable.

It follows from (4) that no declaration must make a hole in the scope of a value constructor by introducing the same identifier as a variable; this is because, in the scope of the declaration which introduces id as a value constructor, any occurrence of id in a pattern is interpreted as the constructor and not as the binding occurrence of a new variable.

The syntax-classes var, con, tycon and exn all depend on which bindings are in force, but only the classes var and con are necessarily disjoint. The context determines (as described above) to which class each identifier occurrence belongs.

In the Core language, an identifier may be given infix status by the infix or infixr directive; this status only pertains to its use as a var or a con. If id has infix status, then "exp1 id exp2" (resp. "pat1 id pat2") may occur wherever the application "id(exp1,exp2)" (resp. "id(pat1,pat2)") would otherwise occur. On the other hand, non-infixed occurrences of id must be prefixed by the keyword "op". Infix status is cancelled by the nonfix directive. (Note: the tuple expression "(exp1,exp2)" is a derived form of the numerically labelled record expression "{#1=exp1,#2=exp2}", and a similar derived form exists for numerically labelled record patterns. See Section 6.1.)

2.5 Comments

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

2.6 Lexical analysis

Each item of lexical analysis is either a reserved word, a numeric label, a special constant or an identifier; comments and formatting characters separate items (except within string constants; see Section 2.3) and are otherwise ignored. At each stage the longest next item is taken.

As a consequence of this simple approach, spaces or parentheses are needed sometimes to separate identifiers and reserved words. Two examples are

```
    a:= !b      or      a:=(!b)      but not     a:=!b
                                     (assigning contents of b to a)
~ :int->int     or   (~):int->int    but not     ~:int->int
                                     (unary minus qualified by its type)
```

Rules which allow omission of spaces in such examples would also forbid certain symbol sequences as identifiers; moreover, such rules are hard to remember. It seems better to keep a simple scheme and tolerate a few extra spaces or parentheses.

2.7 Delimiters

Not all constructs have a terminating reserved word; this would be verbose. But a compromise has been adopted; end terminates any construct which declares bindings with local scope. This involves only the let, local and abstype constructs.

2.8 The bare syntax

The syntax of the bare language is presented in the adjacent table. The following metasyntactic conventions are adopted:

Conventions

(1) The brackets "<< >>" enclose optional phrases.

(2) Repetition of iterated phrases is represented by "__"; this must not be confused with "...", a reserved word used in flexible record patterns.

(3) For any syntax class s, we define the syntax class s_seq as follows:

```
s_seq ::=  s
           (s1, __ ,sn)   (n≥1)
```

(4) Alternatives are in order of decreasing precedence.

(5) L (resp. R) means left (resp. right) association.

The syntax of types binds more tightly that that of expressions, so type constraints should be parenthesized if not followed by a reserved word.

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

THE SYNTAX OF THE BARE LANGUAGE

EXPRESSIONS exp

```
aexp ::=
  var                          (variable)
  con                          (constructor)
  { lab1=exp1, __ ,            (record,n≥0)
    labn=expn }
  ( exp )

exp ::=
  aexp                         (atomic)
  exp aexp                    L(application)
  exp : ty                    L(constraint)
  exp handle handler          R(handle exc'ns)
  raise exn with exp           (raise exc'n)
  let dec in exp end           (local dec'n)
  fn match                     (function)

match ::=
  rule1 | __ | rulen           (n≥1)

rule ::=
  pat => exp

handler ::=
  hrule1 || __ || hrulen       (n≥1)

hrule ::=
  exn with match
  ? => exp
```

DECLARATIONS dec

```
dec ::=
  val vb                       (values)
  type tb                      (types)
  datatype db                  (datatypes)
  abstype db                   (abstract
      with dec end               datatypes)
  exception eb                 (exceptions)
  local dec in dec' end        (local dec'n)
  dec1<<;>> __ decn<<;>>       (sequence,n≥0)
```

PROGRAMS : dec1 ; __ decn ;

PATTERNS pat

```
apat ::=
  _                            (wildcard)
  var                          (variable)
  con                          (constant)
  { lab1=pat1, __ ,            (record,n≥0)**
    labn=patn <<, ...>>}
  ( pat )

pat ::=
  apat                         (atomic)
  con apat                    L(construction)
  pat : ty                    L(constraint)
  var<<:ty>> as pat           R(layered)
```

VALUE BINDINGS vb

```
vb ::=
  pat = exp                    (simple)
  vb1 and __ and vbn           (multiple,n≥2)
  rec vb                       (recursive)
```

TYPE BINDINGS tb

```
tb ::=
  <<tyvar_seq>>tycon
      = ty                     (simple)
  tb1 and __ and tbn           (multiple,n≥2)
```

DATATYPE BINDINGS db

```
db ::=
  <<tyvar_seq>>tycon
      = constrs                (simple)
  db1 and __ and dbn           (multiple,n≥2)

constrs ::=
  con1<<of ty1>>| __ | conn<<of tyn>>
```

EXCEPTION BINDINGS eb

```
eb ::=
  exn<<:ty>><< =exn'>>(simple)
  eb1 and __ and ebn           (multiple,n≥2)
```

TYPES ty

```
ty ::=
  tyvar                        (type variable)
  <<ty_seq>>tycon              (type constr'n)
  { lab1:ty1, __ ,
    labn:tyn }                 (record type,n≥0)
  ty -> ty'                   R(function type)
  ( ty )
```

** The reserved word "..." is called the record wildcard. If it is absent, then the pattern will match any record with exactly those components which are specified; if it is present, then the matched record may also contain further components. If it occurs when n=0, then the preceding comma is omitted; "{...}" is a pattern which matches any record whatever.

3. Evaluation

3.1 Environments and Values

Evaluation of phrases takes place in the presence of an ENVIRONMENT and a STORE. An ENVIRONMENT E has two components: a value environment VE associating values to variables and to value constructors, and an exception environment EE associating exceptions to exception names. A STORE S associates values to references, which are themselves values. (A third component of an environment, a type environment TE, is ignored here since it is relevant only to type-checking and compilation, not to evaluation.)

A value v is either a constant (a nullary constructor), a construction (a constructor with a value), a record, a reference, or a function value. A record value is a set of label-value pairs, written "{lab1=v1, __ ,labn=vn}", in which the labels are distinct; note that the order of components is immaterial. The labels labi in a record value must be either all identifiers, or else they must be the numeric labels #1, #2, __ , #n; the two kinds of label may not be mixed. A function value f is a partial function which, given a value, may return a value or a packet; it may also change the store as a side-effect.

An exception e, associated to an exception name exn in any exception environment, is an object drawn from an infinite set (the nature of e is immaterial, but see Section 3.8). A packet p=(e,v) is an exception e paired with a value v, called the excepted value. Neitner exceptions nor packets are values.

Besides possibly changing S (by assignment), evaluation of a phrase returns a result as follows:

Phrase	Result
Expression	v or p
Value binding	VE or p
Type or datatype binding	VE
Exception binding	EE
Declaration	E or p

For every phrase except a handle expression, whenever its evaluation demands the evaluation of an immediate subphrase which returns a packet p as result, no further evaluation of subphrases occurs and p is also the result of the phrase. This rule should be remembered while reading the evaluation rules below. In presenting the rules, explicit type constraints (:ty) have been ignored since they have no effect upon evaluation.

3.2 Environment manipulation

We may write <(id1,v1) __ (idn,vn)> for a value environment VE (the idi being distinct). Then VE(idi) denotes vi, <> is the empty value environment, and VE+VE' means the value environment in which the associations of VE' supersede those of VE. Similarly for exception environments. If E=(VE,EE) and E'=(VE',EE'), then E+E' means (VE+VE',EE+EE'), E+VE' means E+(VE',<>), etc. This implies that an identifier may be associated both in VE and in EE without conflict.

3.3 Matching patterns

The matching of a pattern pat to a value v either fails or yields a value environment. Failure is distinct from returning a packet, but a packet will be returned when all patterns fail in applying a match to a value (see Section 3.4). In the following rules, if any component pattern fails to match then the whole pattern fails to match.

The following is the effect of matching a pattern pat to a value v, in each of the cases for pat (with failure if any condition is not satisfied):

_ : the empty value environment <> is returned.

var : the value environment <(var,v)> is returned.

con<<pat>> : if v = con<<v'>> then pat is matched to v', else failure.

var as pat : pat is matched to v returning VE; then <(var,v)>+VE is returned.

{ lab1=pat1, __ , labn=patn, <<, ...>>} :
if v={ lab1=v1, __ ,labm=vm } , where m≥n if "..." is present and m=n otherwise, then pati is matched to vi returning VEi, for each i; then VE1+ __ +VEn is returned.

3.4 Applying a match

Assume environment E. Applying a match "pat1=>exp1| __ |patn=>expn" to value v returns a value or packet as follows. Each pati is matched to v in turn, from left to right, until one succeeds returning VEi; then expi is evaluated in E+VEi. If none succeeds, then the packet (ematch,()) is returned, where ematch is the standard exception bound by predeclaration to the exception name "match". But matches which may fail are to be detected by the compiler and flagged with a warning; see Section 10(2).

Thus, for each E, a match denotes a function value.

3.5 Evaluation of expressions

Assume environment E=(VE,EE). Evaluating an expression exp returns a value or packet as follows, in each of the cases for exp:

var : the value VE(var) is returned.

con : the value VE(con) is returned.

exp aexp : exp is evaluated, returning function value f; then aexp is evaluated, returning value v; then f(v) is returned.

{ lab1=exp1, __ , labn=expn } :
the expi are evaluated in sequence, from left to right, returning vi respectively; then the record { lab1=v1, __ , labn=vn } is returned.

raise exn with exp : exp is evaluated, returning value v; then the

packet (e,v) is returned, where e = EE(exn).

exp handle handler : exp is evaluated; if exp returns a value v, then v is returned; if it returns a packet p = (e,v) then the handling rules of the handler are scanned from left to right until a rule is found which satisfies one of two conditions:
(1) it is of form "exn with match" and e=EE(exn), in which case match is applied to v;
(2) it is of form "? => exp'", in which case exp' is evaluated.
If no such hrule is found, then p is returned.

let dec in exp end : dec is evaluated, returning E'; then exp is evaluated in E+E'.

fn match : f is returned, where f is the function of v gained by applying match to v in environment E.

3.6 Evaluation of value bindings

Assume environment E = (VE,EE). Evaluating a value binding vb returns a value environment VE' or a packet as follows, by cases of vb:

pat = exp : exp is evaluated in E, returning value v; then pat is matched to v; if this returns VE', then VE' is returned, and if it fails then the packet (ebind,()) is returned, where ebind is the standard exception bound by predeclaration to the exception name "bind".

vb1 and __ and vbn: vb1,__,vbn are evaluated in E from left to right, returning VE1, __ ,VEn; then VE1+ __ +VEn is returned.

rec vb : vb is evaluated in E', returning VE', where E' = (VE+VE',EE). Because the values bound by "rec vb" must be function values (see 10(4)), E' is well defined by "tying knots" (Landin).

3.7 Evaluation of type and datatype bindings

The components VE and EE of the current environment do not affect the evaluation of type bindings or datatype bindings (TE affects their type-checking and compilation). Evaluation of a type binding just returns the empty value environment <>; the purpose of type bindings in the Core language is merely to provide an abbreviation for a compound type. Evaluation of a datatype binding db returns a value environment VE' (it cannot return a packet) as follows, by cases of db:

<<tyvar_seq>>tycon = con1<<of ty1>> | __ | conn<<of tyn>> :
the value environment VE' = <(con1,v1), __ ,(conn,vn)> is returned, where vi is either the constant value coni (if "of tyi" is absent) or else the function which maps v to coni(v). Other effects of this datatype binding are dealt with by the compiler or type-checker, not by evaluation.

db1 and __ and dbn : db1, __ , dbn are evaluated from left to right, returning VE1, __ ,VEn; then VE' = VE1+ __ +VEn is returned.

3.8 Evaluation of exception bindings

Assume environment E = (VE,EE). The evaluation of an exception binding eb returns an exception environment EE' as follows, by cases of eb:

exn << =exn'>> : EE' = <(exn,e)> is returned, where
(1) if exn' is present then e = EE(exn'); this is a non-generative exception binding since it merely re-binds an existing exception to exn;
(2) otherwise e is a previously unused exception; this is a generative exception binding.

eb1 and __ and ebn : eb1, __ ,ebn are evaluated in E from left to right, returning EE1,__,EEn; then EE' = EE1+__+EEn is returned.

3.9 Evaluation of declarations

Assume environment E = (VE,EE). Evaluating a declaration dec returns an environment E' or a packet as follows, by cases of dec:

val vb : vb is evaluated, returning VE'; then E' = (VE',<>) is returned.

type tb : E' = (<>,<>) is returned.

datatype db : db is evaluated, returning VE'; then E' = (VE',<>) is returned.

abstype db with dec end :
db is evaluated, returning VE'; then dec is evaluated in E+VE', returning E'; then E' is returned.

exception eb : eb is evaluated, returning EE'; then E' = (<>,EE') is returned.

local dec1 in dec2 end :
dec1 is evaluated, returning E1, then dec2 is evaluated in E+E1, returning E2; then E' = E2 is returned.

dec1<<;>> __ decn<<;>> :
each deci is evaluated in E+E1+ __ +E(i-1), returning Ei, for i = 1,2, __ ,n; then E' = (<>,<>)+E1+ __ +En is returned. Thus when n=0 the empty environment is returned.

Each declaration is defined to return only the new environment which it makes, but the effect of a declaration sequence is to accumulate environments.

3.10 Evaluation of programs

The evaluation of a program "dec1 ; __ decn ;" takes place in the initial presence of the standard top-level environment ENV0 containing all the standard bindings (see Section 5). For i>0 the top-level environment ENVi, present after the evaluation of deci in the program, is defined recursively as follows: deci is evaluated in ENV(i-1) returning environment Ei, and then ENVi = ENV(i-1)+Ei.

4. Directives

Directives are included in ML as (syntactically) a subclass of declarations. They possess scope, as do all declarations.

There is only one kind of directive in the standard language, namely those concerning the infix status of value variables and constructors. Others, perhaps also concerned with syntactic conventions, may be included in extensions of the language. The directives concerning infix status are:

infix<<r>> <<d>> id1 __ idn
nonfix id1 __ idn

where d is a digit. The infix and infixr directives introduce infix status for each idi (as a value variable or constructor), and the nonfix directive cancels it. The digit d (default 0) determines the precedence, and an infixed identifier associates to the left if introduced by infix, to the right if by infixr. Different infixed identifiers of equal precedence associate to the left. As indicated in Appendix 1, the precedence of infixed application is just weaker than that of application.

While id has infix status, each occurrence of it (as a value variable or constructor) must be infixed or else preceded by op. Note that this includes occurrences of the identifier within patterns, even binding occurrences of variables.

Several standard functions and constructors have infix status (see Appendix 3) with precedence; these are all left associative except "::".

It may be thought better that the infix status of a variable or constructor should be established in some way within its binding occurrence, rather than by a separate directive. However, the use of directives avoids problems in parsing.

The use of local directives (introduced by let or local) imposes on the parser the burden of determining their textual scope. A quite superficial analysis is enough for this purpose, due to the use of end to delimit local scopes.

5. Standard bindings

The bindings of this section form the standard top-level environment ENV0.

5.1 Standard type constructors

The bare language provides the record type "{lab1:ty1, __ , labn:tyn}" for each n≥0, and the infixed function-type constructor "->". Otherwise, type constructors are postfixed. The following are standard:

```
Type constants (nullary constructors)  : unit,bool,int,real,string
Unary type constructors                : list,ref
```

None of the identifiers ->, *, unit, bool, int, real, string, list, ref may be redeclared as type constructors. ("*" is used in the type of n-tuples, a derived form of record type.)

The constructors unit, bool and list are fully defined by the following assumed declaration

```
infixr 5 ::
type unit = {}
datatype bool =  true | false
  and 'a list =   nil | op :: of {#1:'a, #2:'a list}
```

The word "unit" is chosen since the type contains just one value "{}", the empty record. This is why it is preferred to the word "void" of ALGOL 68.

The type constants int, real and string are equipped with special constants as described in Section 2.3. The type constructor ref is for constructing reference types; see Section 7.

5.2 Standard functions and constants

All standard functions and constants are listed in Appendix 3. There is not a lavish number; we envisage function libraries provided by each implementation, together with the equivalent ML declaration of each function (though the implementation may be more efficient). In time, some such library functions may accrue to the standard; a likely candidate for this is a group of array-handling functions, grouped in a standard declaration of the unary type constructor "array".

Most of the standard functions and constants are familiar, so we need mention only a few critical points:

(1) explode yields a list of strings of size 1; implode is iterated string concatenation (^). ord yields the Ascii code number of the first character of a string; chr yields the Ascii character (as a string of size 1) corresponding to an integer. The ordering relations <, >, <= and >= on strings use the lexicographic order; for this purpose, the newline character "\n" is identified with linefeed.

(2) ref is a monomorphic function, but in patterns it may be used polymorphically, with type 'a ->'a ref .

(3) The character functions ord and chr, the arithmetic operators *, /, div, mod, + and - , and the standard functions floor, sqrt, exp and ln may raise standard exceptions (see Section 5.3) whose name in each case is the same as that of the function. This occurs for ord when the string is empty; for chr when the the integer is not an Ascii code; and for the others when the result is undefined or out of range.

(4) The values r = a mod d and q = a div d are determined by the condition d*q + r = a , where either $0 \leq r < d$ or $d < r \leq 0$. Thus the remainder takes the same sign as the divisor, and has lesser magnitude. The result of arctan lies between $\pm pi/2$, and ln (the inverse of exp) is the natural logarithm. The value floor(x) is the largest integer $\leq x$; thus rounding may be done by floor(x+0.5) .

(5) Two multi-typed functions are included as quick debugging aids. The function print :ty->ty is an identity function, which as a side-effect prints its argument exactly as it would be printed at top-level. The printing caused by "print(exp)" will depend upon the type ascribed to this <u>particular</u> occurrence of exp ; thus print is not a normal polymorphic function. The function makestring :ty->string is similar, but instead of printing it returns as a string what print would produce on the screen. Since top-level printing is not fully specified, programs using these two functions should not be ported between implementations.

5.3 <u>Standard exceptions</u>

All predeclared exception names are of type unit. There are three special ones: match, bind and interrupt. These exceptions are raised, respectively, by failures of matching and binding as explained in Sections 3.4 and 3.6, and by an interrupt generated (often by the user) outside the program. Note, however, that match and bind exceptions cannot occur unless the compiler has given a warning, as detailed in Section 10(2),(3), except in the case of a top-level declaration as indicated in 10(3).

The only other predeclared exception names are

ord chr * / div mod + - floor sqrt exp ln

Each name identifies the corresponding standard function, which is ill-defined or out of range for certain arguments, as detailed in Section 5.2. For example, using the derived <u>handle</u> form explained in Section 8.2, the expression

3 div x <u>handle</u> div => 10000

will return 10000 when x = 0.

6. Standard Derived Forms

6.1 Expressions and Patterns

DERIVED FORM	EQUIVALENT FORM	
Types :		
`ty1 * __ * tyn`	`{ #1:ty1, __ , #n:tyn }`	
Expressions :		
`()`	`{ }`	(no space in "()")
`( exp1, __ , expn )`	`{ #1=exp1, __ , #n=expn }`	(n≥2)
`raise exn`	`raise exn with ()`	
`case exp of match`	`(fn match) (exp)`	
`if exp then exp1 else exp2`	`case exp of true=>exp1 | false=>exp2`	
`exp orelse exp'`	`if exp then true else exp'`	
`exp andalso exp'`	`if exp then exp' else false`	

`( exp1; __ ; expn; exp )`	`case exp1 of (_) =>`
	`  __ __`
	`  __ __`
	`  case expn of (_) => exp`
		(n≥1)

`let dec in` `  exp1; __ ; expn end`	`let dec in` `  (exp1; __ ; expn) end`	

`while exp do exp'`	`let val rec f = fn () =>`
	`  if exp then (exp'; f()) else ()`
	`  in f() end`	

`[ exp1 , __ , expn ]`	`exp1:: __ ::expn::nil`	(n≥0)
Handling rules :		
`exn => exp`	`exn with (_) => exp`	
Patterns :		
`()`	`{ }`	(no space in "()")
`( pat1, __ , patn )`	`{ #1=pat1, __ , #n=patn }`	(n≥2)
`[ pat1, __ , patn ]`	`pat1:: __ ::patn::nil`	(n≥0)
`{ __, id<<:ty>><<as pat>>, __ }`	`{ __, id=id<<:ty>><<as pat>>, __ }`	

Each derived form may be implemented more efficiently than its equivalent form, but must be precisely equivalent to it semantically. The type-checking of each derived form is also defined by that of its equivalent form. The precedence among all bare and derived forms is shown in Appendix 1.

The derived type "ty1 * __ * tyn" is called an (n-)tuple type, and the values of this type are called (n-)tuples. The associated derived forms of expressions and patterns give exactly the treatment of tuples in the previous ML proposal [5].

The shortened raise form is only admissible with exceptions of type unit. The shortened form of handling rule is appropriate whenever the excepted value is immaterial, and is therefore (in the full form) matched to the wildcard pattern.

The final derived pattern allows a label and its associated variable to be elided in a record pattern, when they are the same identifier.

6.2 Bindings and Declarations

A new syntax class fb, of function bindings, is introduced. Function bindings are a convenient form of value binding for function declarations. The equivalent form of each function binding is an ordinary value binding. These new function bindings must be declared by "fun", not by "val"; however, the bare form of value binding may still be used to declare functions, using val or val rec.

DERIVED FORM	EQUIVALENT FORM
Function bindings fb :	
var apat11 __ apat1n<<:ty>>= exp1 | __ __ | var apatm1 __ apatmn<<:ty>>= expm	var = fn x1 => __ fn xn => case (x1, __ , xn) of (apat11, __ , apat1n) => exp1<<:ty>> | __ __ | (apatm1, __ , apatmn) => expm<<:ty>> (where the xi are new, and m,n≥1)
fb1 and __ and fbn	vb1 and __ and vbn (where vbi is the equivalent of fbi)
Declarations :	
fun fb	val rec vb (where vb is the equivalent of fb)
exp	val it = exp

The last derived declaration (using "it") is only allowed at top-level, for treating top-level expressions as degenerate declarations; "it" is just a normal value variable.

7. References and equality

7.1 References and assignment

Following Cardelli, references are provided by the type constructor "ref". Since we are sticking to monomorphic references, there are two overloaded functions available at all monotypes mty:

(1) ref : mty -> mty ref, which associates (in the store) a new reference with its argument value. "ref" is a constructor, and may be used polymorphically in patterns, with type 'a -> 'a ref .

(2) op := : mty ref * mty -> unit , which associates its first (reference) argument with its second (value) argument in the store, and returns () as result.

The polymorphic contents function "!" is provided, and is equivalent to the declaration "fun !(ref x) = x".

7.2 Equality

The overloaded equality function op = : ety * ety -> bool is available at all types ety which admit equality, according to the definition below. The effect of this definition is that equality will only be applied to values which are built up from references (to arbitrary values) by value constructors, including of course constant values. On references, equality means identity; on objects of other types ety, it is defined recursively in the natural way.

The types which admit equality are as follows, assuming that abbreviations introduced by type bindings have first been expanded out:

(1) A type ty admits equality iff it is built from arbitrary reference types by the record type construction and by type constructors which admit equality.

(2) The standard type constructors bool, int, real, string and list all admit equality.

Thus for example, the type (int * 'a ref)list admits equality, but (int * 'a)list and (int -> bool)list do not.

A user-defined type constructor tycon, declared by a datatype binding db whose form is

<<tyvar_seq>>tycon = con1<<of ty1>> | __ | conn<<of tyn>>

admits equality within its scope (but, if declared by abstype, only within the with part of its declaration) iff it satisfies the following condition:

(3) Each construction type tyi in this binding is built from arbitrary reference types and type variables, either by type constructors which already admit equality or by tycon or any other type constructor declared by simultaneously with tycon, provided these other type constructors also satisfy the present condition.

The first paragraph of this section should be enough for an intuitive understanding of the types which admit equality, but the precise definition is given in a form which is readily incorporated in the type-checking mechanism.

8. Exceptions

8.1 Discussion

Some discussion of the exception mechanism is needed, as it goes a little beyond what exists in other functional languages. It was proposed by Alan Mycroft, as a means to gain the convenience of dynamic exception trapping without risking violation of the type discipline (and indeed still allowing polymorphic exception-raising expressions). Brian Monahan put forward a similar idea. Don Sannella also contributed, particularly to the nature of the derived forms (Section 8.2); these forms give a pleasant way of treating standard exceptions, as explained in Section 5.3.

The rough and ready rule for understanding how exceptions are handled is as follows. If an exception is raised by a raise expression

raise exn with exp

which lies in the textual scope of a declaration of the exception name exn, then it may be handled by a handling rule

exn with match

in a handler, but only if this handler is in the textual scope of the same declaration. Otherwise it may only be caught by the universal handling rule

? => exp' .

This rule is perfectly adequate for exceptions declared at top level; some examples in Section 8.4 below illustrate what may occur in other cases.

8.2 Derived forms

A handler discriminates among exception packets in two ways. First, it handles just those packets (e,v) for which e is the exception bound to the exception name in one of its handling rules; second, the match in this rule may discriminate upon v, the excepted value. Note however that, if a universal handling rule "? => exp'" is activated, then all packets are handled without discrimination. Thus "?" may be considered as a wildcard, matching any packet. It should be used with some care, bearing in mind that it will even handle interrupts.

A case which is likely to be frequent is when discrimination is required upon the exception, but not upon the excepted value; in this case, the derived handling rule

exn => exp'

is appropriate for handling. Further, exceptions of type unit may be raised by the shortened form

raise exn

since the only possible excepted value is ().

8.3 An example

To illustrate the generality of exception handling, suppose that we have declared some exceptions as follows:

```
exception oddlist :int list and oddstring :string
```

and that a certain expression exp:int may raise either of these exceptions and also runs the risk of dividing by zero. The handler in the following handle expression would deal with these exceptions:

```
exp handle oddlist   with  []      => 0
                           | [x]     => 2*x
                           | x::y::_ => x div y
        || oddstring with "" => 0
                           |  s => size(s)-1
        || div  =>  10000
```

Note that the whole expression is well-typed because in each handling rule the type of each match-pattern is the same as the exception type, and because the result type of each match is int , the same as the type of exp. The last handling rule is the shortened form appropriate for exceptions of type unit .

Note also that the last handling rule will handle div exceptions raised by exp , but will not handle the div exception which may be raised by "x div y" within the first handling rule. Finally, note that a universal handling rule

```
        ||  ?  =>  50000
```

at the end would deal with all other exceptions raised by exp .

8.4 Some pathological examples

We now consider some possible misuses of exception handling, which may arise from the fact that exception declarations have scope, and that each evaluation of a generative exception binding creates a distinct exception. Consider a simple example:

```
exception exn : bool;
fun f(x) =
     let exception exn:int in
          if x > 100 then raise exn with x else x+1
     end;
f(200) handle exn with true=>500 | false=>1000;
```

The program is well-typed, but useless. The exception bound to the outer exn is distinct from that bound to the inner exn; thus the exception raised by f(200), with excepted value 200, could only be handled by a handler within the scope of the inner exception declaration - it will not be handled by the handler in the program, which expects a boolean value. So this exception will be reported at top level. This would apply even if the outer exception declaration were also of type int; the two exceptions bound to exn would still be distinct.

On the other hand, if the last line of the program is changed to

```
f(200) handle ? => 500 ;
```

then the exception will be caught, and the value 500 returned. A universal handling rule (i.e. containing "?") catches any exception packet, even one exported from the scope of the declaration of the associated exception name, but cannot examine the excepted value in the packet, since the type of this value cannot be statically determined.

Even a single textual exception binding - if for example it is declared within a recursively defined function - may bind distinct exceptions to the same identifier. Consider another useless program:

```
fun f(x) =
     let exception exn in
          if p(x) then a(x) else
          if q(x) then f(b(x)) handle exn => c(x)
                  else raise exn with d(x)
     end;
f(v);
```

Now if p(v) is false but q(v) is true, the recursive call will evaluate f(b(v)). Then, if both p(b(v)) and q(b(v)) are false, this evaluation will raise an exn exception with excepted value d(b(v)). But this packet will not be handled, since the exception of the packet is that which is bound to exn by the inner - not outer - evaluation of the exception declaration.

These pathological examples should not leave the impression that exceptions are hard to use or to understand. The rough and ready rule of Section 8.1 will almost always give the correct understanding.

9. Type-checking

The type discipline is exactly as in original ML, and here only a few points about type-checking will be discussed.

In a match "pat1=>exp1 | __ | patn=>expn", the types of all pati must be the same (ty say), and if variable var occurs in pati then all free occurrences of var in expi must have the same type as its occurrence in pati. In addition, the types of all the expi must be the same (ty' say). Then ty->ty' is the type of the match. The type of "fn match" is the type of the match.

The type of a handler rule "exn with match" is ty', where exn has type ty and match has type ty->ty'. The type of a universal handling rule "? => exp" is the type of exp . The type of a handler is the type of all its handling rules (which must therefore be the same), and the type of "exp handle handler" is that of both exp and handler. The type of "raise exn with exp" is arbitrary, but exp and exn must have the same type. Exceptions may be polymorphic; any exn must have the same type at all occurrences within the scope of its declaration.

A type variable is only explicitly bound (in the sense of variable-binding in lambda-calculus) by its occurrence in the tyvar_seq on the left hand side of a simple type or datatype binding "<<tyvar_seq>>tycon = __ ", and then its scope is the right hand side. (This means for example that bound uses of 'a in both tb1 and tb2 in the type binding "tb1 and tb2" bear no relation to each other.) Otherwise, repeated occurrences of a type variable may serve to link explicit type constraints. The scope of such a type variable is determined by its first occurrence (ignoring all occurrences which lie within scopes already thus determined). If this first occurrence is in an exception declaration, then it

has the same scope as the declared exception(s); otherwise, its scope is the smallest val (or fun) declaration in which it lies. For example, consider

```
fun G(f:'a->'b)(x:'a) = let val y:'b = f(x)
                            and Id = (fn x:'c => x)
                        in (Id(x):'a, Id(y):'b) end
```

Here the scope of both 'a and 'b is the whole fun declaration, while the scope of 'c is just the val declaration. Note that this allows "Id" to be used polymorphically after its declaration. Moreover, type-checking must not further constrain a type variable within its scope. Thus for example the declaration "fun Apply(f:'a->'b)(x:'a):'b = x" - in which "x" has been written in error in place of "f(x)" - will be faulted since it requires 'a and 'b to be equated.

A simple datatype binding "<<tyvar_seq>>tycon = __ " is generative, since a new unique type constructor (denoted by tycon) is created by each textual occurrence of such a binding. A simple type binding "<<tyvar_seq>>tycon = ty", on the other hand, is non-generative; to take an example, the type binding " 'a couple = 'a * 'a " merely allows the type expression "ty couple" to abbreviate "ty * ty" (for any ty) within its scope. There is no semantic significance in abbreviation; in the Core language it is purely for brevity, though in Modules non-generative type-bindings are essential in matching Signatures. However, the type-checker should take some advantage of non-local type abbreviations in reporting types at top-level; in doing this, it may need to choose sensibly between different possible abbreviations for the same type.

Some standard function symbols (e.g. =,+) stand for functions of more than one type; in these cases the type-checker should complain if it cannot determine from the context which is intended (an explicit type constraint may be needed). Note that there is no implicit coercion in ML, in particular from int to real; the conversion function real:int->real must be used explicitly.

10. Syntactic restrictions

(1) No pattern may contain the same variable twice. No binding may bind the same identifier twice. No record type, record expression or record pattern may use the same label twice. In a record type or expression, either all labels must be identifiers or they must be the numeric labels #1, __ , #n for some n. The same applies to record patterns, except that some numeric labels may be absent if "..." is present.

(2) In a match "pat1=>exp1 | __ | patn=>expn", the pattern sequence pat1, __ , patn should be irredundant and exhaustive. That is, each patj must match some value (of the right type) which is not matched by pati for any i<j, and every value (of the right type) must be matched by some pati. The compiler must give warning on violation of this restriction, but should still compile the match. Thus the "match" exception (see Section 3.4) will only be raised for a match which has been flagged by the compiler. The restriction is inherited by derived forms; in particular, this means that in the function binding "var apat1 __ apatn<<:ty>> = exp" (consisting of one clause only), each separate apati should be exhaustive by itself.

(3) For each value binding "pat = exp" the compiler must issue a report (but still compile) if either pat is not exhaustive or pat contains no variable. This will (on both counts) detect a mistaken declaration like "val nil = exp" in which the user expects to declare a new variable nil (whereas the language

dictates that nil is here a constant pattern, so no variable gets declared). However, these warnings should not be given when the binding is a component of a top-level declaration val vb ; e.g. "val x::l = exp1 and y = exp2" is not faulted by the compiler at top level, but may of course generate a "bind" exception (see Section 3.6).

(4) For each value binding "pat = exp" within rec, exp must be of the form "fn match". The derived form of value binding given in Section 6.2 necessarily obeys this restriction.

(5) In the left hand side "<<tyvar_seq>>tycon" of a simple type or datatype binding, the tyvar_seq must contain no type variable more than once. The right hand side may contain only the type variables mentioned on the left. Within the scope of the declaration of tycon, any occurrence of tycon must be accompanied by as many type arguments as indicated by the <<tyvar_seq>> in the declaration.

(6) Assume temporarily that locally declared datatype constructors have been renamed so that no two textually distinct datatype bindings bind identically-named datatype constructors. Then, if the typechecker ascribes type ty to a program phrase p , every datatype constructor in ty must be declared with scope containing p . For example, if ty is ascribed to exp in "let dec in exp end" then ty must contain no datatype constructor declared by dec , since ty is also the type ascribed to the whole let expression.

(7) Every global exception binding - that is, not localised either by let or by local - must be explicitly constrained by a monotype.

(8) If, within the scope of a type constructor tycon, a type binding tb or datatype binding db binds (simultaneously) one or more type constructors tycon1, __ , tyconn then: (a) if the identifiers tyconi are all distinct from tycon, then their value constructors (if any) must also have identifiers distinct from those (if any) of tycon; (b) if any tyconi is the same identifier as tycon, then any value constructor of tycon may be re-bound as a value constructor for one of tycon1, __ , tyconn, but is otherwise considered unbound (as a variable or value constructor) within the scope of tb or db , unless it is bound again therein. This constraint ensures that the scope of a type constructor is identical with the scopes of its associated value constructors, except that in an abstype declaration the scope of the value constructors is restricted to the with part.

11. Relation between the Core language and Modules.

The sister report [7] on ML Modules describes how ML declarations are grouped together into Structures which can be compiled separately. Structures, and the Functors which generate them, may not be declared locally within ML programs, but only at top-level or local to other Structures and Functors; this means that the Core language is largely unaffected by their nature.

However, Structures and their components (types, values, exceptions and other Structures) may be accessed from ML programs via qualified names of the form

$$id1.\ \_\_\ .idn.id \qquad (n \geq 1)$$

where id1, __ , idn are Structure names, each idi is the name of a component

structure of id(i-1) for $1<i\leq n$, and id is either a type constructor, a value constructor, a value variable, an exception name or a Structure name declared as a component of Structure idn. Thus the syntax classes tycon, con, var and exn are extended to include qualified names. Further, the declaration

open id1. __ .idn ($n\geq 1$)

(where id1, __ , idn are as above) allows the components of the Structure id1. __ .idn to be named without qualification in the scope of the declaration.

Each Structure is equipped with a Signature, which determines the nature and type of each component, and this permits static analysis and type-checking for programs which use the Structure.

12. Conclusion

This design has been under discussion for over two years. In the conclusion (Section 11) of [5] we predicted that a few infelicities of design would emerge during the last year, and this has happened. But they are satisfyingly few. Use of the language by a wider community will probably raise further suggestions for change, but against this we must set the advantage of maintaining complete stability in the language. We shall adopt a policy of minimum change.

At the same time, extensions to ML - ones which preserve the validity of all existing programs - may be suggested either by practical need or by increased theoretical understanding. Examples of the latter may be the introduction of polymorphic assignment, or the extension of the equality predicate to a wider class of types. We hope that these extensions will be made when appropriate.

REFERENCES:

[1] M.Gordon, R.Milner and C.Wadsworth (1979) Edinburgh LCF. Springer-Verlag, Lecture Notes in Computer Science, Vol 78.

[2] R.Burstall, D.MacQueen and D.Sannella (1980) HOPE: An Experimental Applicative Language. Report CSR-62-80, Computer Science Dept, Edinburgh University.

[3] L.Cardelli (1982) ML under UNIX. Bell Laboratories, Murray Hill, New Jersey.

[4] R.Milner (1983) A Proposal for Standard ML. Report CSR-157-83, Computer Science Dept, Edinburgh University.

[5] R.Milner (1984) The Standard ML Core Language. Report CSR-168-84, Computer Science Dept, Edinburgh University.

[6] R.Harper (1985) Standard ML Input/Output. Computer Science Department, Edinburgh University.

[7] D.MacQueen (1985) Modules for Standard ML. AT&T Bell Laboratories, Murray Hill, New Jersey.

APPENDIX 1. SYNTAX : EXPRESSIONS and PATTERNS

(See Section 2.8 for conventions and remarks)

```
aexp ::=
       <<op>>var                                  (variable)
       <<op>>con                                  (constructor)
       { lab1=exp1, __ , labn=expn }              (record, n≥0)
       ()                                         (0-tuple)
       ( exp1 , __ , expn )                       (n-tuple, n≥2)
       [ exp1 , __ , expn ]                       (list, n≥0)
       ( exp1 ; __ ; expn )                       (sequence, n≥1)

exp ::=
       aexp                                       (atomic)
       exp aexp                                  L(application)
       exp id exp'                                (infixed application)
       exp : ty                                  L(constraint)
       exp andalso exp'                           (conjunction)
       exp orelse exp'                            (disjunction)
       exp handle handler                        R(handle exception)
       raise exn <<with exp>>                     (raise exception)
       if exp then exp1 else exp2                 (conditional)
       while exp do exp'                          (iteration)
       let dec in exp1 ; __ ; expn end            (local declaration, n≥1)
       case exp of match                          (case expression)
       fn match                                   (function)

match ::=                              handler ::=
       rule1 | __ | rulen   (n≥1)             hrule1 || __ || hrulen    (n≥1)

rule ::=                               hrule ::=
       pat => exp                             exn with match
                                              exn => exp
                                              ? => exp

apat ::=
       _                                          (wildcard)
       <<op>>var                                  (variable)
       con                                        (constant)
       { lab1=pat1, __ , labn=patn <<, ...>>}     (record,n≥0) **
       ()                                         (0-tuple)
       ( pat1 , __ , patn )                       (tuple, n≥2)
       [ pat1 , __ , patn ]                       (list, n≥0)
       ( pat )

pat ::=
       apat                                       (atomic)
       <<op>>con apat                            L(construction)
       pat con pat'                               (infixed construction)
       pat : ty                                  L(constraint)
       <<op>>var<<:ty>> as pat                   R(layered)
```

** If n=0 then omit the comma; "{...}" is the pattern which matches any record. If a component of a record pattern has the form "id=id<<:ty>><<as pat>>", then it may be written in the elided form "id<<:ty>><<as pat>>".

APPENDIX 2. SYNTAX : TYPES, BINDINGS, DECLARATIONS

(See Section 2.8 for conventions)

```
ty ::=
       tyvar                                          (type variable)
       <<ty_seq>>tycon                                (type construction)
       { lab1:ty1, __ , labn:tyn }                    (record type, n≥0)
       ty1 * __ * tyn                                 (tuple type, n≥2)
       ty1 -> ty2                                    R(function type)
       ( ty )

vb ::=
       pat = exp                                      (simple)
       vb1 and __ and vbn                             (multiple, n≥2)
       rec vb                                         (recursive)

fb ::=
       <<op>>var apat11 __ apat1n<<:ty>> = exp1       (clausal function,
         |  __   __                                                m,n≥1) **
            __   __
         | <<op>>var apatm1 __ apatmn<<:ty>> = expm
       fb1 and __ and fbn                             (multiple,n≥2)

tb ::=
       <<tyvar_seq>>tycon = ty                        (simple)
       tb1 and __ and tbn                             (multiple,n≥2)

db ::=
       <<tyvar_seq>>tycon = constrs                   (simple)
       db1 and __ and dbn                             (multiple,n≥2)

constrs ::=
       <<op>>con1<<of ty1>> | __
         | <<op>>conn<<of tyn>>                       (n≥1)

eb ::=
       exn<<:ty>><< =exn'>>                           (simple)
       eb1 and __ and ebn                             (multiple, n≥2)

dec ::=
       val vb                                         (value declaration)
       fun fb                                         (function declaration)
       type tb                                        (type declaration)
       datatype db                                    (datatype declaration)
       abstype db with dec end                        (abstract type declaration)
       exception eb                                   (exception declaration)
       local dec in dec' end                          (local declaration)
       exp                                            (top-level only)
       dir                                            (directive)
       dec1<<;>> __ decn<<;>>                         (declaration sequence,n≥0)

dir ::=
       infix<<r>> <<d>> id1 __ idn                    (declare infix,0≤d≤9)
       nonfix id1 __ idn                              (cancel infix)
```

** If var has infix status then op is required in this form; alternatively var may be infixed in any clause. Thus, at the start of any clause, "op var (apat,apat') __ " may be written "(apat var apat') __ "; the parentheses may also be dropped if ":ty" or "=" follows immediately.

APPENDIX 3. PREDECLARED VARIABLES and CONSTRUCTORS

In the types of these bindings, "num" stands for either int or real, and "nums" stands for integer, real or string (the same in each type). Similarly "ty" stands for an arbitrary type, "mty" stands for any monotype, and "ety" (see Section 7.2) stands for any type admitting equality.

nonfix

```
nil             : 'a list
map             : ('a->'b) -> 'a list
                        -> 'b list
rev             : 'a list -> 'a list

true,false      : bool
not             : bool -> bool

~               : num  -> num
abs             : num  -> num
floor           : real -> int
real            : int  -> real
sqrt            : real -> real
sin,cos,arctan  : real -> real
exp,ln          : real -> real

size            : string -> int
chr             : int -> string
ord             : string -> int
explode         : string -> string list
implode         : string list -> string

ref             : mty -> mty ref
!               : 'a ref -> 'a

print           : ty -> ty
makestring      : ty -> string
```

infix

```
Precedence 7 :
     /  : real * real -> real
   div  : int  * int  -> int
   mod  :  "      "       "
    *   : num  * num  -> num

Precedence 6 :
     +  :  "      "       "
     -  :  "      "       "
     ^  : string * string -> string

Precedence 5 :
     :: : 'a * 'a list -> 'a list
     @  : 'a list * 'a list
                     -> 'a list
Precedence 4 :
     =  : ety  * ety  -> bool
     <> :  "      "       "
     <  : nums * nums -> bool
     >  :  "      "       "
     <= :  "      "       "
     >= :  "      "       "

Precedence 3 :
     o  : ('b->'c) * ('a->'b)
                      -> ('a->'c)
     := : mty ref * mty -> unit
```

Special constants: as in Section 2.3.

Notes:

(1) The following are constructors, and thus may appear in patterns:

nil true false ref :: and all special constants.

(2) Infixes of higher precedence bind tighter. "::" associates to the right; otherwise infixes of equal precedence associate to the left.

(3) The meanings of these predeclared bindings are discussed in Section 5.2.

Standard ML Input/Output

Robert W. Harper

June 6, 1985

1 Introduction

This document describes the Standard ML [1] character stream input/output system. The basic primitives defined below are intended as a simple basis that may be compatibly superseded by a more comprehensive I/O system that provides for streams of arbitrary type or a richer repertoire of I/O operations. The I/O primitives are organized into two modules, one for the basic I/O primitives that are required to be provided by all implementations, and one for extensions to the basic set. An implementation may support any, all, or none of the functions in the extended I/O module, and may extend this module with new primitives. If an implementation does not implement a primitive from the set of extensions, then it must leave it undefined so that unsupported features are recognized at compile time.

The fundamental notion in the SML I/O system if the (finite or infinite) character stream. There are two types of stream, `instream` for input streams, and `outstream` for output streams. These types are provided by the implementation of the basic I/O module. Interaction with the outside world is accomplished by associating a stream with a *producer* (for input streams) or a *consumer* (for output streams). The notion of a producer and a consumer is purely metaphorical. Their realization is left to each implementation; the SML programmer need be aware of their existence only insofar as it is necessary to imagine the source (or sink) of characters in a stream. For instance, ordinary disk files, terminals, and processes are all acceptable as producers or consumers. A given implementation may support a range producers and consumers; all implementations must allow disk files to be associated with input and output streams.

Streams in SML may be finite or infinite; finite streams may or may not have a definite end. A natural use of an infinite stream is the connection of an `instream` to a process that generates an infinite sequence, say of prime numbers represented as numerals. Most often streams are finite, though not always terminated. Ordinary disk files are a good example of producers of finite streams of characters.

Processes as producers give rise to the notion of an unterminated finite stream — a process may at any time refuse to supply an more characters to a stream, a condition which is, of course, undetectable. All subsequent input requests will therefore wait forever. Primitives are provided for detecting the end of an input stream and for terminating an output stream.

The stream types provided by the basic I/O module are abstract, and as such have no visible structure. However, it is helpful to imagine that each stream has associated with it a buffer that mediates the interaction between the ML system and the producer or consumer associated with that stream, and a control object, which is used for device–specific mode–setting and control. A typical example of the use of the control object is to modify the character processing performed by a terminal device driver.

In the spirit of simplicity and generality, this proposal does not treat such implementation–dependent details as the resolution of multiple file access (both within and between processes), and the names of files and processes. The window between the SML system and the operating system is limited to two primitives, each of which takes a string parameter whose interpretation is implementation–specific. One convention must be enforced by all implementations — end of line is represented by the single newline character, \n, regardless of how it is represented by the host system. However, since end of file is a condition, as opposed to a character, the means by which this condition is indicated on a terminal is left to the implementation.

2 Basic I/O Primitives

The fundamental I/O primitives are packaged into a structure with signature `BasicIO` (see Figure 1). A structure matching this signature (and having the semantics defined below) must be provided by every SML implementation. It is implicitly open'd by the standard prelude so that these identifiers may be used without the qualifier `BasicIO`.

The type `instream` is the type of input streams and the type `outstream` is the type of output streams. The exception `io_failure` is used to represent all of the errors that may arise in the course of performing I/O. The value associated with this exception is a string representing the type of failure. In general, any I/O operation may fail if, for any reason, the host system is unable to perform the requested task. The value associated with the exception should describe the type of failure, insofar as this is possible.

The standard prelude binds `std_in` to an instream and binds `std_out` to an outstream. For interactive ML processes, these are expected to be associated with the user's terminal. However, an implementation that supports the connection of

```
signature BasicIO = sig
  (* Types and exceptions *)
  type instream
  type outstream
  exception io_failure: string

  (* Standard input and output streams *)
  val std_in: instream
  val std_out: outstream

  (* Stream creation *)
  val open_in: string -> instream
  val open_out: string -> outstream

  (* Operations on input streams *)
  val input: instream * int -> string
  val lookahead: instream -> string
  val close_in: instream -> unit
  val end_of_stream: instream -> bool

  (* Operations on output streams *)
  val output: outstream * string -> unit
  val close_out: outstream -> unit
end
```

Figure 1: Basic I/O Primitives

processes to streams may associate one process's `std_in` with another's `std_out`.

The `open_in` and `open_out` primitives are used to associate a disk file with a stream. The expression `open_in(s)` creates a new `instream` whose producer is the file named `s` and returns that stream as value. If the file named by `s` does not exist, the exception `io_failure` is raised with value `"Cannot open "^s`. Similarly, `open_out(s)` creates a new `outstream` whose consumer is the file `s`, and returns that stream.

The `input` primitive is used to read characters from a stream. Evaluation of `input(s,n)` causes the removal of `n` characters from the input stream `s`. If fewer than `n` characters are currently available, then the ML system will block until they become available from the producer associated with `s`.[1] If the end of stream

[1]The exact definition of "available" is implementation–dependent. For instance, operating sys-

```
signature ExtendedIO = sig
  val execute: string -> instream * outstream
  val flush_out: outstream -> unit
  val can_input: instream * int -> bool
  val input_line: instream -> string
  val open_append: string -> outstream
  val is_term_in: instream -> bool
  val is_term_out: outstream -> bool
end
```

Figure 2: Extended I/O Primitives

is reached while processing an input, fewer than n characters may be returned. In particular, input from a closed stream returns the null string. The function `lookahead(s)` returns the next character on `instream` `s` without removing it from the stream. Input streams are terminated by the `close_in` operation. This primitive is provided primarily for symmetry and to support the reuse of unused streams on resource–limited systems. The end of an input stream is detected by `end_of_stream`, a derived form that is defined as follows:

```
val end_of_stream(s) = (lookahead(s)="")
```

Characters are written to an `outstream` with the output primitive. The string argument consists of the characters to be written to the given `outstream`. The function `close_out` is used to terminate an output stream. Any further attempts to output to a closed stream cause `io_failure` to be raised with value `"Output stream is closed"`.

3 Extended I/O Primitives

In addition to the basic I/O primitives, provision is made for a some extensions that are likely to be provided by many implementations. The signature ExtendedIO (see Figure 2) describes a set of operations that are commonly used but are either too complex to be considered primitive or to be implementable on all hosts.

The function `execute` is used to create a pair of streams, one an `instream` and one an `outstream`, and associate them with a process. The string argument to

tems typically buffer terminal input on a line–by–line basis so that no characters are available until an entire line has been typed.

execute is the (implementation–dependent) name of the process to be executed. In the case that the process is an SML program, the `instream` created by execute is connected to the `std_out` stream of the process, and the `outstream` returned is connected to the process's `std_in`.

The function `flush_out` ensures that the consumer associated with an `outstream` has received all of the characters that have been written to that stream. It is provided primarily to allow the ML user to circumvent undesirable buffering characteristics that may arise in connection with terminals and other processes. All output streams are flushed when they are closed, and in many implementations an output stream is flushed whenever a newline is encountered if that stream is connected to a terminal.

The function `can_input` takes an `instream` and a number and determines whether or not that many characters may be read from that stream without blocking. For instance, a command processor may wish to test whether or not a user has typed ahead in order to avoid an unnecessary prompt. The exact definition of "currently available" is implementation–specific, perhaps depending on such things as the processing mode of a terminal.

The `input_line` primitive returns a string consisting of the characters from an `instream` up through, and including, the next end of line character. If the end of stream is reached without reaching an end of line character, all remaining characters from the stream (*without* an end of line character) are returned.

Files may be open for output while preserving their contents by using the open_append primitive. Subsequent output to the `outstream` returned by this primitive is appended to the contents of the specified file.

Basic support for the complexities of terminal I/O are also provided. The pair of functions `is_term_in` and `is_term_out` test whether or not a stream is associated with a terminal. These functions are especially useful in association with `std_in` and `std_out` because they are opened as part of the standard prelude. A terminal may be designated as the producer or consumer of a stream using the ordinary open_in and open_out functions; an implementation supporting this capability must specify a naming convention for designating terminals. Terminal I/O is, in general, more complex than ordinary file I/O. In most cases the `ExtendedIO` module provided by an implementation will have additional operations to support mode control. Since the details of such control operations are highly host–dependent, the functions that may be provided are left unspecified.

Acknowledgements

The Standard ML I/O system is based on Luca Cardelli's proposal [2], and on a simplified form of it proposed by Kevin Mitchell and Robin Milner. The final

version was prepared in conjunction with Dave MacQueen, Dave Matthews, Robin Milner, Kevin Mitchell, and Larry Paulson.

References

[1] Robin Milner, *The Standard ML Core Language*, Edinburgh University.

[2] Luca Cardelli, *Stream Input/Output*, AT&T Bell Laboratories.