Volume I, Number 3 December, 1983

Polymorphism

The ML/LCF/Hope Newsletter

Contents

Letter from the Editors

There has been an unusually long gap between this issue of Polymorphism and the last, but that is certainly not because of lack of activity in the Polymorphism community. Last April Robin Milner decided that the time was ripe to consider revising ML in the light of several years of experience with both ML and Hope, and he wrote a first draft of a proposal for the new version. Intensive discussions of the draft ensued, leading to several revised drafts which ultimately converged to the Proposal included in this issue.

Although the new version of ML is called "Standard ML", it is far more than a tidying-up of an old version. Neither is it simply an amalgamation of the original LCF/ML version with Luca Cardelli's variant. It is a carefully done reconsideration of the language starting from basic principles. It is also an attempt to unify divergent developments, so that the ML user community can avoid fragmentation and enjoy the advantages of a common version of the language.

Early in the discussions, it was decided that more progress could be made by separating the treatment of the core language from peripheral though important topics such as I/O and modules, and this resulted in two subsidiary reports. The first, by Luca Cardelli, presents a stream input/output system, which has been partially implemented as part of his Unix based ML system. The second is Dave MacQueen's module proposal, which explores relatively new territory and is exposed here for the first time. It is therefore rather more expository in style than the main proposal. A limited precursor of this proposal was also implemented as part of Unix ML, and though the proposal does not discuss implementation issues, most of the machinery required to support it was in fact developed for this prototype version.

We hope that the appearance of these three reports will set off a wide and fruitful discussion. Questions, criticisms and alternative proposals are invited, and we will publish the responses in our next issue. It is important that as many viewpoints as possible be heard, but it is also important to stabilize the design soon so that full implementations can be produced and put to use. We hope to see a final specification by next summer, with complete implementations available shortly thereafter.

Regarding the implementation of Standard ML, the "ML under Unix" manual describes an ML implementation which is in the process of being converted to Standard ML. The manual is supposed to reflect the true state of the implementation, and it is therefore not yet a full Standard ML manual. The major differences concern the failure mechanisms; other minor differences are described in an Appendix. Moreover the manual describes the prototype I/O and module systems mentioned above, which are not part of the basic standard.

It is now time to unveil the mystery of the cover picture: it is a detail from the picture shown in the next page (drawn by Luca Cardelli on a Blit terminal, using Rob Pike's "twid" program). The picture shows a "real programmer" walking a tight rope over a "bit jungle" full of dangers. The woodpecker represents violations of the type system, the balance bar is weak typechecking, the vultures are debuggers and the flies are bugs. Falling (crashing) into the bit jungle is a terrible experience: snakes and alligators are there ready to chew you to bits.

Instead, the ML programmer flies a hang glider in a blue sky, climbing higher and higher towards clouds of higher types. The bit jungle is invisible; the green forest of abstraction hides it from view, and guarantees a soft landing in case the programmer had to drift down.

Luca Cardelli
David MacQueen

A Proposal for Standard ML

Robin Milner, November 1983

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 can 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 contentious: input/output and facilities 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 the core language without the two contentious areas. This course is further justified by the fact that the core language appears to be almost completely unaffected by the choice of input/output primitives and of separate compilation constructs. Also, there are already strong and carefully considered proposals, from Cardelli and MacQueen respectively, on how to design these two vital facilities. These proposals will appear very soon, and together with this document they will form a complete language definition which can be adopted in its entirety, while still leaving open the possibility of adopting only parts of it. But the strong hope is that the whole will be very widely accepted.

The main contributors to the proposed language, through their design work on ML and on HOPE, are:
Rod Burstall, Luca Cardelli, Michael Gordon, David MacQueen,
Robin Milner, Lockwood Morris, Malcolm Newey, Christopher Wadsworth.
The final proposal also owes much to criticisms and suggestions from many other people: Guy Cousineau, Gerard Huet, Robert Milne, Kevin Mitchell, Brian Monahan, Peter Mosses, Alan Mycroft, Larry Paulson, David Rydeheard, Don Sannella, David Schmidt, John Scott, Stefan Sokolowski, Bernard Sufrin, Philip Wadler. Most of them have expressed strong support for most of the design; any inadequacies which remain are my fault, but I have tried to represent the consensus.

[1] M.Gordon, R.Milner and C.Wadsworth, Edinburgh LCF; Springer-Verlag, Lecture Notes in Computer Science, Vol 78, 1979.
[2] R.Burstall, D.MacQueen and D.Sannella, HOPE: An Experimental Applicative Language; Report CSR-62-80, Computer Science Dept, Ediburgh University, 1980.
[3] L.Cardelli, ML under UNIX; Bell Laboratories, Murray Hill, New Jersey, 1982.

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; unfortunately it is not yet documented, and we will do better to wait for a clear exposition either from Damas or - as promised - from David MacQueen. In the proposed language much can be done to get the polymorphic effect by passing assignment functions as parameters; it is worthwhile experimenting with this method, and there is further advantage in keeping to the simple 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. The latter have definite advantages which the former lack; on the other hand, the HOPE treatment is well-rounded in its own terms. Though a combination of these features is possible, it seemed (at least to me, but some disagreed!) to entail too rich a language for the present proposal. Thus the HOPE treatment is fully adopted here.

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, just because of the exception mechanism). Third, the principle implies that the present document is not a full language definition; it only becomes so when the proposals for input/output and for separate compilation are added.

1.3 An example

The following declaration illustrates some constructs of the proposed 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 abstraction}
with                                                     {      constructor.      }
  val nulldict = dict nil
                                                         {The function lookup may}
  exception lookup : unit                                {    raise an exception.}

  val lookup (key:int)                                   {'a is the result type.  }
          (dict entrylist) :'a  =
    let val rec search nil = escape lookup                {An auxiliary clausal   }
              | search ((k, item)::entries) =             {  function declaration.}
                  if key=k then item
                  else if key<k then escape lookup
                  else search entries
    in search entrylist
    end

  val enter (newentry as (key,item))                     {A layered varstruct.    }
          (dict entrylist) :'a dictionary  =
    let val rec 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   {of dictionary}
```

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

```
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 further 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 types (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, type constraint, tupling, raising and handling exceptions, local declaration (using let) and function abstraction.

An important subclass of expressions are the varstructs; they are essentially expressions containing only variables and value constructors, and are used as patterns to create value bindings. Declarations may declare value variables (using value bindings), types with associated constructors or operations (using type 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 complete 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 else end
escape exception fun handle if in infix
infixr let local nonfix of op orelse raise
rec then trap type val with while

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

2.3 Special constants

The unique object of type unit is denoted by the special constant ().

An integer constant is any non-empty sequence of digits, possibly preceded by ~ representing negation.

A string constant is a sequence of zero or more printable characters or spaces enclosed between quotes ("), but within which any quote symbol is preceded by the escape character \ . Use of \ in strings also has meaning as follows:

\1..\9	One to nine spaces	\E	Escape
\0	Ten spaces	\N	Null (Ascii 0)
\C	Carriage return	\D	Del (Ascii 127)
\L	Line feed	\^c	Ascii control character c
\T	Tabulation	\c	c (any other character)
\B	Backspace		

2.4 Identifiers

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

value variables	(var)
value constructors	(con)
type variables	(tyvar)
type constructors	(tycon)
exception identifiers	(exid)

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 four classes (var, con, tycon, exid) are represented by identifiers not starting with a prime. Thus type variables are disjoint from the other four classes. Otherwise, the syntax class of an occurrence of identifier id is determined thus:

(1) In types, id is a type constructor, and must be within the scope of the type binding which introduced it.

(2) Following exception, raise or handle id is an exception identifier.

(3) Elsewhere, id is a value constructor if it occurs in the scope of a type binding which introduced it as such, otherwise it is a value variable.

It follows from (3) that no value binding can make a hole in the scope of a value constructor by introducing the same identifier as a variable, since this identifier must stand for the constructor in any varstruct which lies in the scope of the type declaration by which this constructor was introduced. In fact, by means of a syntactic restriction (see Section 10(8)), we ensure that the scopes of a type constructor and of its associated value constructors are identical.

The syntax-classes var, con, tycon and exid 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 complete 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. "vs1 id vs2") may occur wherever the application "id(exp1,exp2)" (resp. "id(vs1,vs1)") 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.

2.5 Comments

A comment is any character sequence within curly brackets {} in which curly brackets are properly nested. An unmatched occurrence of } is faulted by the compiler.

2.6 Lexical analysis

Each item of lexical analysis is either a reserved word or a special constant or an identifier; comments and non-visible characters separate items (except spaces within string constants) 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, such as adopted by Cardelli in VAX ML, also forbid certain symbol sequences as identifiers and - more importantly - 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; and terminates any construct which declares bindings with local scope. This involves only the let, local and abstype constructs.

2.8 The bare syntax

Conventions: {..} means optional.
For any syntax class s, define s_seq ::= s
(s1,..,sn) (n≥1)

Alternatives are in order of decreasing precedence.
L (resp. R) means left (resp. right) association.
Parentheses may enclose phrases of any named syntax class defined below or in Appendices 1 and 2.

EXPRESSIONS exp

```
aexp ::=
  var                          (variable)
  con                          (constructor)
  ( exp )

exp ::=
  aexp                         (atomic)
  exp aexp                     L(application)
  exp : ty                     L(constraint) **
  exp1 , .. , expn             (tuple, n≥2)
  raise exid exp               (raise exc'n)
  let dec in exp and           (local dec'n)
  exp handle exid match        (handle exc'n) *
  exp1 ? exp2                  (handle exc'ns) *
  fun match                    (function)

match ::=                      (n≥1)
  vs1.exp1 | .. | vsn.expn
```

DECLARATIONS dec

```
dec ::=
  val vb                       (values)
  type tb                      (types)
  abstype tb
     with dec end              (abs. types)
  exception eb                 (exceptions)
  local dec1
     in dec2 end               (local dec'n)
  dec1 {;} .. decn {;}         (sequence, n≥0)
```

TYPES ty

```
ty ::=
  tyvar                        (type variable)
  {ty_seq} tycon               (type construction)
  ty1 * .. * tyn               (tuple type, n≥2)
  ty1 -> ty2                   R(function type)
```

VARSTRUCTS vs

```
avs ::=
  _                            (wildcard)
  var                          (variable)
  con                          (constant)
  ( vs )

vs ::=
  avs                          (atomic)
  con avs                      L(construction)
  vs : ty                      L(constraint) **
  var{:ty} as vs               (layered)
  vs1 , .. , vsn               (tuple, n≥2)
```

VALUE BINDINGS vb

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

TYPE BINDINGS tb

```
tb ::=
  {tyvar_seq} tycon
       = constrs               (simple)
  tb1 and .. and tbn           (multiple,n≥2)
  rec tb                       (recursive)

constrs ::=                    (n≥1)
  con1{of ty1} | .. | conn{of tyn}
```

EXCEPTION BINDINGS eb

```
eb ::=
  exid {: ty}                  (simple)
  eb1 and .. and ebn           (multiple,n≥2)
```

PROGRAMS: dec1 ; .. decn ;

** The syntax of types binds more tightly than that of expressions, so these forms should be parenthesized if not followed by a reserved word.
* These two forms are of equal precedence and left associative.

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 identifiers. 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 typechecking and compilation, not to evaluation.)

An exception e, associated to an exception identifier exid in any EE, is an object from which exid may be recovered. A packet p=(e,v) is an exception e paired with a value v, called the excepted value. Neither 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 binding	TE
Exception binding	EE
Declaration	E or p

For every phrase except a handle or "?" 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.

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. Every other value is either a constant (a nullary constructor), a construction (a constructor with a value), a tuple or a reference.

3.2 Environment manipulation

We may write <(id1,v1)...(idn,vn)> for a value environment, where the idi are distinct. Then <> is the empty VE, and VE1 + VE2 means the VE in which the associations of VE2 supersede those of VE1. 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 varstructs

The matching of a varstruct vs to a value v either fails or yields a VE. Failure is distinct from returning a packet, but will result in this when all varstructs fail in applying a match to a value (see Section 3.4). In the following rules, if any component varstruct fails to match then the whole varstruct fails to match.

The following is the effect of matching cases of varstruct vs to value v:

_	: the empty VE is returned.
var	: the VE <(var,v)> is returned.
con[vs]	: if v = con[v'] then vs is matched to v', else failure.
var[:ty] as vs	: vs is matched to v returning VE; then <(var,v)> + VE is returned.
vs1,..,vsn	: if v=(v1,...,vn) then vsi is matched to vi returning VEi, for each i; then VE1+...+VEn is returned.
vs:ty	: vs is matched to v.

3.4 Applying a match

Assume environment E. Applying match m = vs1.exp1|..|vsn.expn to value v returns a value or packet as follows:

Each vsi 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 identifier "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 m 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, by cases of exp:

var	: returns value VE(var).
con	: returns value VE(con).
exp aexp	: exp is evaluated, returning function value f; then aexp is evaluated, returning value v; then f(v) is returned.
exp1,...,expn	: the expi are evaluated in sequence, from left to right, returning vi respectively; then (v1,...,vn) is returned.
raise exid exp	: exp is evaluated, returning value v; then packet (e,v) is returned, where e = EE(exid).
exp handle exid match	: exp is evaluated; if exp returns a value v, then v is returned; if exp returns p = (e,v) then: (1) if e = EE(exid) then match is applied to v, (2) otherwise p is returned.

exp1 ? exp2 : exp1 is evaluated; if exp1 returns a value v, then v is returned; if exp1 returns any packet, then exp2 is evaluated.

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

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

exp:ty : exp is evaluated.

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:

vs = exp : exp is evaluated in E, returning value v; then vs 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 identifier "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 evaluating vb must be function values (Section 10(4)), E' is well defined by "tying knots" (Landin).

3.7 Evaluation of type bindings

The components VE and EE of the current environment do not affect the evaluation of type bindings (TE affects their type checking and compilation). Evaluating a type binding tb returns a value environment VE' (it cannot return a packet) as follows, by cases of tb:

(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). Note that all other effect of this type binding is handled by the compiler or type-checker, not by evaluation.

tb1 and..and tbn : tb1, .. ,tbn are evaluated from left to right, returning VE1, .. ,VEn; then VE' = VE1 + .. + VEn is returned.

rec tb : tb is evaluated. Note again that the recursion is handled by typechecking only.

3.8 Evaluation of exception bindings

The evaluation of an exception binding eb returns an exception environment EE' as follows, by cases of eb:

exid (:ty) : a new exception e is generated (an object from which the exception identifier exid may be retrieved), and EE' = <(exid,e)> is returned.

eb1 and..and ebn : eb1, .. ,ebn are evaluated 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 : tb is evaluated, returning VE'; then E' = (VE',<>) is returned.

abstype tb with dec end : tb 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 (<>,<>)+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 ENV containing all the standard bindings (see Section 5). The top-level environmen ENVi, present after the evaluation of deci in the program, is define 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| (p) id1 ... idn
nonfix id1 ... idn
```

where p is a non-negative integer. The infix and infixr directives introduce infix status for each idi (as a value variable or constructor), and the nonfix directive cancels it. The integer p (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.

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 such occurrences within varstructs, even within the varstructs of a match.

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 (introduce 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 and to delimit local scopes.

5. Standard bindings

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

5.1 Standard type constructors

The bare language provides the function-type constructor, ->, and for each n ≥ 2 a tuple-type constructor *n. Type constructors are in general postfixed in ML, but -> is infixed, and the n-ary tuple-type constructed from ty1, .., tyn is written "ty1 * .. * tyn". Besides these type constructors, the following are standard:

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

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

```
infixr 30 ::
type unit = () and bool = true | false
and rec 'a list = nil | op :: of 'a * 'a list
```

The word "unit" is chosen since the type contains just one value; this is why it is preferred to the word "void" of ALGOL 68. Note that it is also (up to isomorphism) a unit for type tupling, though we do not exploit this isomorphism by allowing a coercion between the types ty and ty * unit .

The type constant int is equipped with constants 0, 1, ~1, 2, .. . The type constant string is equipped with constants as described in Section 2.3. The type constructor ref is for constructing reference types; see Section 7.

Real numbers will be defined as a standard type in a later extension.

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.

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

(3) The list destructors hd and tl, the character functions ord and chr, and the five arithmetic operators *, div, mod, + and - , may generate standard exceptions (see Section 5.3) whose identifier in each case is the same as that of the function. This occurs for hd and tl when the argument is nil; for ord when the string is empty; for chr when the character is undefined; and for the others when the result is out of range or, for div and mod, when the second argument is zero.

(4) div and mod are defined as in PASCAL.

5.3 Standard exceptions

All predeclared exception identifiers are of type unit. There are two special ones, match and bind; these exceptions are raised on failure of matching or binding, as explained in Sections 3.4 and 3.6 respectively. Note, however, that these 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 identifiers are

```
hd  tl  ord  chr  *  div  mod  +  -
```

These are the identifiers of standard functions which are ill-defined for certain arguments, as detailed in Section 5.2. For example, using the derived form trap explained in Section 8.2, the expression

```
3 div x trap div 10000
```

will return 10000 when x = 0. Also the function hd, together with its exception identifier, could have been defined in ML as follows (using some derived forms introduced in Section 6):

```
exception hd : unit
val hd(nil) = escape hd
  | hd(x::l) = x
```


6. Standard Derived Forms

6.1 Expressions and Varstructs

DERIVED FORM	EQUIVALENT FORM
Expressions :-	
escape exid	raise exid ()
exp1 trap exid exp2	exp1 handle exid _.exp2
case exp of match	(fun match) exp
if exp then exp1 else exp2	case exp of true.exp1 | false.exp2
exp1 orelse exp2	if exp1 then true else exp2
exp1 andalso exp2	if exp1 then exp2 else false
exp1 ; exp2	case exp1 of _.exp2
while exp1 do exp2	let val rec f = fun (). if exp1 then (exp2;f()) else () in f() end
[exp1 ; .. ; expn]	exp1:: .. ::expn::nil (n≥0)
Varstructs :	
[vs1 ; .. ; vsn]	vs1:: .. ::vsn::nil (n≥0)

The 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 binding power of all bare and derived forms is shown in Appendix 1. Note particularly that a semicolon, whether used in declaration sequencing, in expression sequencing or in the derived list form, always has weakest binding power.

The escape form is only admissible with exceptions of type unit, and the trap form is useful both in this case and when the excepted value is immaterial.

6.2 Bindings and Declarations

DERIVED FORM	EQUIVALENT FORM
Value bindings :	

`var avs11 .. avs1n {:ty} = exp1`
`  | ..`
`  | var avsm1 .. avsmn {:ty} = expm`
	`var = fun x1. .. fun xn.`
	`  case (x1, .., xn) of`
	`  ( avs11, .., avs1n . exp1 {:ty}`
	`  | ..`
	`  | avsm1, .., avsmn . expm {:ty} )`
	`.. where the xi are new, and m,n≥1.`

Declarations :	
`exp`	`val it = exp`
`type tb`	`type tb'  dec'`

`abstype tb with dec end`	`abstype tb' with`
	`  local dec' in dec end`
	`end`
	`.. where tb',dec' are described below.`

The derived value binding allows function definitions, possibly Curried, with several clauses. The first derived declaration is only allowed at top-level, for treating top-level expressions as degenerate declarations; "it" is just a normal value variable.

The last two derived declarations are concerned with a generalisation of type bindings. In place of the bare syntax for constructions

```
constrs ::= con1 {of ty1} | .. | conn {of tyn}
```

a more general form is allowed in the full language, as follows:

```
constrs ::= con1 {of compt_seq1} | .. | conn {of compt_seqn}
compt ::= {var :} ty
```

The effect is to allow selector functions to be declared in the same phrase as constructor functions. (They can always be declared later, as illustrated for the selector hd in Section 5.3, but the present derived form makes the declaration simpler to write and easier to implement efficiently.) Before seeing the derivation of the general form from the bare language, it is helpful to look at the simple example of defining the type constructor list complete with its selectors hd and tl, which together form the inverse of :: . The derived declaration is

```
type rec 'a list = nil | op :: of (hd:'a, tl:'a list)
```

This is equivalent, by the general rule below, to the following declaration sequence:


```
type rec 'a list = nil | op :: of 'a * 'a list
exception hd:unit   val hd(x::l) = x | hd(_) = escape hd
exception tl:unit   val tl(x::l) = l | tl(_) = escape tl
```

Unlike other derived forms, the derived type binding is not by itself equivalent to a phrase of the bare language, but any declaration containing the derived binding is equivalent to an expanded declaration in the bare language. To obtain this we first define, for a derived type binding tb, a bare type binding tb' and an auxiliary declaration sequence dec'. Consider any construction .

```
con of ({var1:}ty1, .. ,{varn:}tyn)
```

occurring within the right side of any simple binding within tb (the parentheses could be absent if n=1). Then in the bare type binding tb' the corresponding construction is

```
con of ty1 * .. * tyn
```

Also, for each vari present in the construction, the auxiliary declaration dec' will contain the subsequence

```
exception vari:unit
val vari(con(x1,..,xn)) = xi | vari(_) = escape vari
```

Note that the selectors thus introduced may raise exceptions identified by their own names. However, these exceptions cannot occur when con is the unique constructor of a simple binding; in this case the exception declaration and the second clause of the val declaration (which would be redundant) are omitted from dec'.

Having defined tb' and dec' for any derived type binding tb, we can now expand any type or abstype declaration of tb into a bare language declaration, as shown in the table above.

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 varstructs, with type 'a -> 'a ref .

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

The polymorphic contents function "!" is provided, and is equivalent to the declaration "val !(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 value constants. On references, equality means identity; on objects of other types ety, it is defined recursively in the natural way.

The types ety which admit equality are therefore defined as follows:

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

(2) The standard type constructors *n, unit, int, bool, 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 type binding tb whose bare 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 (if tb is within a rec) by tycon or any other type constructor declared by mutual recursion 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 also put forward a similar idea.

The rough and ready rule for understanding it is as follows. If an exception is raised by a raise expression

raise exid exp

which lies in the textual scope of a declaration of the exception identifier exid, then it may be handled by a handle expression

handle exid match

only if this expression is in the textual scope of the same declaration. If it is not so handled, then it may only be caught by the universal handler "?".

This rule is perfectly adequate for exceptions declared at top level; some examples in 8.3 below illustrate what may occur in other cases.

8.2 Derived forms

A handle expression is a double filter for exception packets. First, it handles only those packets (e,v) for which e is the exception bound to its exception identifier; second, its match may discriminate upon v, the excepted value.

A case which is likely to be frequent is when no discrimination on v is required; in this case, the derived expression

exp1 trap exid exp2

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

escape exid

since the only possible excepted value is ().

These two derived forms differ from those in earlier drafts of this proposal, which rested upon the assumptions of a single predeclared exception identifier of type string. Don Sannella convinced me that the present forms are more elegant and useful; they also harmonise well with the predeclared exceptions of Section 5.3.

8.3 Some examples

The evaluation rules for raise and handle expressions are designed to ensure (with the help of type-checking) that any exception packet which is handled by the expression

handle exid match

will contain an excepted value of the type expected by the match. Consider a simple example:

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

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

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

```
f(200) ? 500 ;
```

then the exception will be caught, and the value 500 returned. The universal exception handler "?" catches any exception packet, even one exported from the scope of the declaration of the associated exception identifier, 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:

```
val rec f(x) =
    let exception exid in
        if p(x) then a(x) else
        if q(x) then f(b(x)) trap exid c(x) else
        raise exid 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 exid 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 exid 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-checking discipline is exactly as in original ML, and therefore need only be described with respect to new phrases.

In a match "vs1.exp1 | .. | vsn.expn", the types of all vsi must be the same (ty say), and if variable var occurs in vsi then all free occurrences of var in expi must have the same type as its occurrence in vsi. There is one relaxation of this rule -- the nase expression; see below. 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 "fun match" is the type of the match. The type of "exp handle exid match" is ty', where exp has type ty', match has type ty->ty', and exid has type ty. The type of "raise exid exp" is arbitrary, but exp and exid must have the same type. Thus the type of an exception may be polymorphic; exid is only required to have the same type at all occurrences within the scope of its declaration (and this must be an instance of any type qualifying the declaration).

A type variable is only explicitly bound (in the sense of variable-binding in lambda-calculus) by its occurrence in the tyvar_seq in the type binding "{tyvar_seq} tycon = constrs", and then its scope is "constrs". 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 (free) type variable may serve to link explicit type constraints. The scope of such a type variable is the top-level declaration or expression in which it occurs.

The type-checker refers to the type environment (TE) component of the environment, and records its findings there. Details of TE are not given in this report; they are compatible with what is done in current ML implementations, except that value constructors (and their types) are associated with the type constructors to which they belong.

10. Syntactic restrictions

(1) No varstruct may contain two occurrences of the same variable.

(2) In a match "vs1.exp1 | .. | vsn.expn", the varstruct sequence vs1, .., vsn should be irredundant and exhaustive. That is, each vsj must match some value (of the right type) which is not matched by vsi for any i<j, and every value (of the right type) must be matched by some vsi. The compiler must give warning on violation of this restriction, but should still compile the match. The restriction is inherited by derived forms; in particular, this means that in the Curried function binding "var avs1 .. avsn [:ty] = exp" (consisting of one clause only), each separate avsi should be exhaustive by itself.

(3) For each value binding "vs = exp" the compiler must issue a report (but still compile) if either vs is not exhaustive or vs 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 varstruct, so no variable gets declared). Cardelli points out this danger.

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 an escape with exception identifier "match" (see Section 3.4).

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

(5) In every instance of "{tyvar_seq} tycon" the tyvar_seq must contain no type variable more than once. The right hand side of a simple type binding may contain only the type variables mentioned on the left.

(6) In "let dec in exp end" and "local dec in dec' end" no type constructor exported by dec may occur in the type of exp or in the type of any variable or value constructor exported by dec'.

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

(8) If a type constructor tycon' is declared within the scope of a type constructor tycon, then (a) if tycon and tycon' are distinct identifiers, then their value constructors must be disjoint; (b) if tycon and tycon' are the same identifier, then the value constructors of the outer declaration are not accessible in the scope of the inner declaration, whether or not the inner and outer declarations employ the same identifier(s) as a value constructor(s). These constraints ensure that the scope of a type constructor is identical with the scopes of its associated value constructors.

11. Conclusion

Many points in this design were ardently debated, and it might be thought worthwhile to end with a detailed discussion of the reasons for each particular choice. But it seems rather better - and certainly simpler - to let the result speak for itself, and to allow infelicities (which presumably exist) to emerge during the next year or so in the course of experience with implementation and use.

It would be reasonable after such a period to collect the reactions and to publish a list of corrections - just those which can be agreed among several seriously concerned implementers and users.

Besides these corrections there will clearly be extensions - design ideas which use the present language as a platform. It will be important to keep these two developments separate as far as possible. Corrections should be few and preferably done at most once; extensions may be many, but need not impair the identity of the present language.

APPENDIX 1

SYNTAX : EXPRESSIONS and VARSTRUCTS
(See Section 2.8 for conventions)

```
aexp ::=
      {op} var                                  (variable)
      {op} con                                  (constructor)
      [ exp1 ; .. ; expn ]                      (list)
      ( exp )

exp ::=
      aexp                                      (atomic)
      exp aexp                                  L(application)
      exp : ty                                  L(constraint) **
      exp1 id exp2                               (infixed application)
      exp1 andalso exp2                          (conjunction)
      exp1 orelse exp2                           (disjunction)
      exp1 , .. , expn                           (tuple, n≥2)
      raise exid exp                             (raise exception)
      escape exid                                (raise unit exception)
      if exp then exp1 else exp2                 (conditional)
      while exp1 do exp2                         (iteration)
      let dec in exp end                         (local declaration)
      case exp of match                          (case expression)
      exp handle exid match                      (handle exception) *
      exp1 trap exid exp2                        (handle exception,discarding value) *
      exp1 ? exp2                                (handle all exceptions) *
      fun match                                  (function)
      exp1 ; exp2                                (sequence)

match ::=                                        (n≥1)
      vs1.exp1 | .. | vsn.expn

vs ::=
      _                                          (wildcard)
      {op} var                                   (variable)
      con                                        (constant)
      [ vs1 ; .. ; vsn ]                         (list, n≥0)
      ( vs )

vs ::=
      avs                                        (atomic)
      {op} con avs                               L(construction)
      vs : ty                                    L(constraint) **
      vs1 con vs2                                (infixed construction)
      var {: ty} as vs                           (layered)
      vs1 , .. , vsn                             (tuple, n≥0)
```

* These three forms are of equal precedence and left associative.
** The syntax of types binds more tightly than that of expressions, so these forms should be parenthesized if not followed by a reserved word.

APPENDIX 2

SYNTAX : TYPES, BINDINGS, DECLARATIONS and PROGRAMS
(See Section 2.8 for conventions)

```
ty ::=
      tyvar                                      (type variable)
      {ty_seq} tycon                             (type construction)
      ty1 * .. * tyn                             (tuple type, n≥2)
      ty1 -> ty2                                 R(function type)

vb ::=
      vs = exp                                   (simple)
      {op} var avs11...avs1n {:ty} = exp1        (clausal function) *
        | ..
        | {op} var avsm1 .. avsmn {:ty} = expn   (m,n≥1)
      vb1 and .. and vbn                         (multiple, n≥2)
      rec vb                                     (recursive)

tb ::=
      {tyvar_seq} tycon = constrs                (simple)
      tb1 and .. and tbn                         (multiple, n≥2)
      rec tb                                     (recursive)

constrs ::=                                      (n≥1)
      con1 {of compt_seq1} | .. | conn {of compt_seqn}

compt ::=
      {var :} ty

eb ::=
      exid {:ty}                                 (simple)
      eb1 and .. and ebn                         (multiple, n≥2)

dec ::=
      val vb                                     (value declaration)
      type tb                                    (type declaration)
      abstype tb with dec end                    (abstract type declaration)
      exception eb                               (exception declaration)
      local dec1 in dec2 end                     (local declaration)
      exp                                        (top-level only)
      dir                                        (directive)
      dec1 {;} .. decn {;}                       (declaration sequence, n≥0)

dir ::=
      infix{r} {p} id1 .. idn                    (declare infix status, p≥0)
      nonfix id1 .. idn                          (cancel infix status)
```

PROGRAMS: dec1 ; .. decn ;

* 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 (avs,avs') may be written : (avs var avs')

where the parentheses may also be dropped if "=" follows immediately.

APPENDIX 3

PREDECLARED VARIABLES and CONSTRUCTORS

nonfix

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

true        : bool
false       : bool
not         : bool -> bool
~           : int -> int

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

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

Special constants:

```
()          : unit
0,1,~1,2,.. : int
"___"       : string
```

infix

Precedence 50:

```
*   : int * int -> int
div :  "     "     "
mod :  "     "     "
```

Precedence 40:

```
+   :  "     "     "
-   :  "     "     "
^   : string * string -> string
```

Precedence 30:

```
::  : 'a * 'a list -> 'a list
@   : 'a list * 'a list
                  -> 'a list
```

Precedence 20:

```
=   : ety * ety -> bool
<>  :  "     "     "
<   : int * int -> bool
>   :  "     "     "
<=  :  "     "     "
>=  :  "     "     "
```

Precedence 10:

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

Notes:

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

nil true false ref ::
.. and all special constants.

(2) mty denotes any monotype, and ety (as explained in Section 7.2) denotes any type admitting equality.

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

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

ML under Unix

Luca Cardelli

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

ML is a statically-scoped **functional** programming language. Functions are first class objects which can be passed as parameters, returned as values and embedded in data structures. Higher-order functions (i.e. functions receiving or producing other functions) are used extensively.

ML is an **interactive** language. An ML session is a dialogue of questions and answers with the ML system. Interaction is achieved by an incremental compiler, which has some of the advantages of interpreted languages (fast turnaround dialogues) and of compiled languages (high execution speed).

ML is a **strongly typed** language. Every ML expression has a type, which is determined statically. The type of an expression is usually automatically inferred by the system, without need of type definitions. The ML type system guarantees that any expression that can be typed will not generate type errors at run time. Static typechecking traps at compile-time a large proportion of bugs in programs.

ML has a **polymorphic type system** which confers on the language much of the flexibility of type-free languages, without paying the conceptual cost of run-time type errors or the computational cost of run-time typechecking.

ML has a rich collection of data types. Moreover the user can define new **abstract data types**, which are indistinguishable from the system predefined types. Abstract types are used extensively in large programs for modularity and abstraction.

ML has an **exception-trap** mechanism, which allows programs to handle uniformly system and user generated exceptions. Exceptions can be selectively trapped, and exception handlers can be specified.

ML programs can be grouped into **modules**, which can be separately compiled. Dependencies among modules can be easily expressed, and the sharing of common submodules is automatically guaranteed. The system keeps track of module versions to detect compiled modules which are out of date.

This manual describes an implementation of ML running on VAX under the Unix operating system.

Unix is a trade mark of Bell Laboratories.
VAX is a trade mark of Digital Equipment Corporation.

Contents

1. Introduction

ML is a *functional* programming language. Functions are first class objects which can be passed as parameters, returned as values and embedded in data structures. Higher-order functions (i.e. functions receiving or producing other functions) are used extensively. Function application is the most important control construct, and it is extremely uniform: all functions take exactly one argument and return exactly one result. Arguments and results can however be arbitrary structures, thereby achieving the effect of passing many arguments and producing many results. Arguments to functions are evaluated before the function calls.

ML is an *interactive* language. An ML session is a dialogue of questions and answers with the ML system. Interaction is achieved by an incremental compiler, which has some of the advantages of interpreted languages (fast turnaround dialogues) and of compiled languages (high execution speed).

ML is *statically scoped*. All the variables are associated with values according to where they occur in the program text, and not depending on run-time execution paths. This avoids name conflicts in large programs, because variable names can be hidden in local scopes, and prevents accidental damage to preexisting programs. Static scoping greatly improves the security and, incidentally, the efficiency of an interactive language, and it is necessary for implementing higher-order functions with their conventional mathematical meaning.

ML is a *strongly typed* language. Every ML expression has a type, which is determined statically. The type of an expression is usually automatically inferred by the system, without need for type definitions. This *type inference* property is very useful in interactive use, when it would be distracting having to provide all the type information. However, it is always possible to specify the type of any expression or function, e.g. as good documentation practice in large programs. The ML type system guarantees that any expression that can be typed will not generate type errors at run time (for some adequate definition of what a type error is). Static typechecking traps at compile-time a large proportion of bugs in programs that make extensive use of the ML data structuring capabilities (typechecking does not help in numerical programs!). Usually, only truly "logical" bugs are left after compilation.

ML has a *polymorphic type system*. Type expressions may contain type variables, indicating, for example, that a function can work uniformly on a class of arguments of different (but structurally related) types. For example the length function which computes the length of a list has type α list -> int (which is automatically inferred from the obvious untyped recursive definition). Length can work on lists of any type (lists of integers, lists of functions, lists of lists, etc.), essentially because it disregards the elements of the list. The polymorphic type system confers on ML much of the flexibility of type-free languages, without paying the conceptual cost of run-time type errors or the computational cost of run-time typechecking.

ML has a rich collection of data types. Moreover the user can define new *abstract data types*, which are indistinguishable from the system predefined types. Abstract types are used extensively in large programs for modularity and abstraction. They put a barrier between the implementor of a package and its users, so that changes in the implementation of an abstract type will not influence the users, as long as the external interface is preserved. Abstract types also provide a clean extension mechanism for the language. If a new data type is needed which cannot be effectively implemented with the existing ML primitives (e.g. bitmaps for graphics), this can be still specified and prototyped in ML as a new abstract type, and then efficiently implemented and added to the basic system. This path has already been followed for arrays, which are not part of the language definition and can be considered as a predefined abstract type which just happens to be more efficient than its ML specification would lead one to believe.

ML has an *exception-trap* mechanism, which allows programs to uniformly handle system and user generated exceptions. Exceptions can be selectively trapped, and handlers can be specified.

ML programs can be grouped into *modules*, which can be separately compiled. Depedencies among modules can be easily expressed, and the sharing of common submodules is automatically guaranteed. The system keeps track of module versions to detect compiled modules which are out

of date.

1.1. Expressions

ML is an expression-based language; all the standard programming constructs (conditionals, declarations, procedures, etc.) are packaged into expressions yielding values. Strictly speaking there are no statements: even side-effecting operations return values.

It is always meaningful to supply arbitrary expressions as arguments to functions (when the type constraints are satisfied), or to combine them to form larger expressions in the same way that simple constants can be combined.

Arithmetic expressions have a fairly conventional appearance; the result of evaluating an expression is presented as a value and its type separated by a colon:

```
expression:     (3 + 5) * 2;

result:             16 : int
```

Sequences of characters enclosed in quotes '"' are called *strings*:

```
expression:     "this is it";

result:             "this is it" : string
```

Tuples of values are built by an infix operator ',' (comma); the type of a tuple is given by the infix type operator *, which denotes cartesian products.

```
expression:     3,4;

result:             3,4 : int * int

expression:     3,4,5;

result:             3,4,5 : int * int * int
```

Lists are enclosed in square brackets and their elements are separated by semicolons. The list type operator is a suffix: int list means list of integers.

```
expression:     [1; 2; 3; 4];

result:             [1; 2; 3; 4] : int list

expression:     [3,4; 5,6];

result:             [(3,4); (5,6)] : (int * int) list
```

Conditional expressions have the ordinary if-then-else syntax (but else cannot be omitted):

```
expression:     if true then 3 else 4;

result:             3 : int

expression:     if (if 3 = 4 then false else true)
                then false
```



```
                else true;

result:             false : bool
```

The if part must be a boolean expression; two predefined constants true and false denote the basic boolean values; and the boolean operators are not, & (and) and or.

1.2. Declarations

Variables can be bound to values by *declarations*. Declarations can appear at the 'top-level', in which case their scope is *global*, or in scope blocks, in which case they are have a limited *local* scope spanning a single expression. Global declarations are introduced in this section, and local declarations in the next one.

Declarations are not expressions: they establish bindings instead of returning values. Value bindings are introduced by the keyword val, additional value bindings are prefixed by and:

```
declaration:    val a = 3
                and b = 5
                and c = 2;

bindings:           val a = 3 : int
                    val b = 5 : int
                    val c = 2 : int

expression:     (a + b) div c;

result:             4 : int
```

In this case we have defined the variables a, b and c at the top level; they will always be accessible from now on, unless redefined. Value bindings printed by the system are always prefixed by val, to distinguish them from type bindings and module bindings which we shall encounter later.

You may notice that all the variables must be initialized when introduced. Their initial values determine their types, which do not need to be given explicitly. The type of an expression is generally inferred automatically by the system and printed after the result.

Value declarations are also used to define functions, with the following intuitive syntax:

```
declaration:    val f x = x + 1;

binding:            val f : int -> int

declaration:    val g(a,b) = (a + b) div 2;

binding:            val g : (int * int) -> int

expression:     f 3, g(8,4);

result:             4,6 : int * int
```

(the arrow -> denotes the function-space type operator.)

The function f has one argument x. The result of a function is just the value of its body, in this case x+1. Arguments to functions do not need to be parenthesized in general (both in definitions and applications): the simple juxtaposition of two expressions is interpreted as a function application. Function application is treated as an invisible infix operator, and it is the strongest-binding

infix operator; expressions like f 3 + 4 are parsed like (f 3) + 4 and not like f (3 + 4).

The function g above has a pair of arguments, a and b; in this case parentheses are needed for, otherwise g 8,4 is interpreted as (g 8), 4.

The identifiers f and g are plain variables which denote functions. Function variables do not need to be applied to arguments:

```
expression:     f, f 3;

result:             fun,4 : (int -> int) * int

declaration:    val h = g;

binding:            val h : (int * int) -> int
```

In the first example above, f is returned as a function and is paired with a number. Functional values are always printed fun, without showing their internal structure. In the second example, g is bound to h as a whole function. Instead of val h = g we could also have written val h(a,b) = g(a,b).

Variables are statically scoped, and their types cannot change. When new variables are defined, they may 'hide' previously defined variables having the same name, but they never 'affect' them. For example:

```
declaration:    val f x = a + x;

binding:            val f : int -> int

declaration:    val a = [1;2;3];

binding:            val a = [1;2;3] : int list

expression:     f 1;

result:             4 : int
```

here the function f uses the top-level variable a, which was bound to 3 in a previous example. Hence f is a function from integers to integers which returns its argument plus 3. Then a is redefined at the top level to be a list of three integers; any subsequent reference to a will yield that list (unless a is redefined again). But f is not effected at all: the old value of a was 'frozen' in f at the moment of its definition, and f keeps adding 3 to its arguments.

This kind of behavior is called *lexical* or *static* scoping of variables. It is quite common in block-structured programming languages, but it is rarely used in languages which, like ML, are interactive. The use of static scoping at the top-level may sometimes be counterintuitive. For example, if a function f calls a previously defined function g, then redefining g (e.g. to correct a bug) will not change f, which will keep calling the old version of g.

1.3. Local scopes

Declarations can be made local by embedding them between the keywords let and in in a scope-block construct. Following the keyword in there is an expression, called the *body* of the scope-block, terminated by the keyword end. The scope of the declaration is limited to this body.

```
expression:     let   val a = 3
                      and b = 5
                in (a + b) div 2 end;

result:             4 : int
```


here the identifiers a and b are bound to the values 3 and 5 respectively for the extent of the expression (a + b) div 2. No top-level binding is introduced; the whole let construct is an expression whose value is the value of its body.

Variables can be locally redefined, hiding the previous definitions for the extent of the local scope. Outside the local scope, the old definitions are not changed:

```
declaration:    val a = 3
                and b = 5;

bindings:           val a = 3 : int
                    val b = 5 : int

expression:     (let val a = 8 in a + b end), a;

result:             13,3 : int * int
```

The body of a scope-block can access all the variables declared in the surrounding environment (like b), unless they are redefined (like a).

The and keyword is used to define sets of independent bindings: none of them can use the variables defined by the other bindings. However, a declaration often needs variables introduced by previous declarations. At the top-level this is done by introducing those declarations sequentially (here two declarations are typed on the same line):

```
declarations:   val a = 3;  val b = 2 * a;

bindings:           val a = 3 : int
                    val b = 6 : int
```

Similarly, declarations can be composed sequentially in local scopes by separating them by ';':

```
expression:     let   val a = 3;
                      val b = 2 * a
                in a,b end;

result:             3,6 : int * int
```

In simple cases, the effect of ';' can be achieved by nested scope blocks; here is an equivalent formulation of the previous example:

```
expression:     let val a = 3
                in  let val b = 2 * a
                    in a,b end
                end;

result:             3,6 : int * int
```

1.4. Lists

Lists are *homogeneous*, i.e. all their elements must have the same type. It is possible to create lists of any type, e.g. lists of strings, lists of lists of integers, lists of functions (of some given type), etc. Many functions dealing with lists can work on lists of any kind (e.g. computing the length), and they do not have to be rewritten every time a new kind of list is needed. Other list functions are more specialized, like taking the sum of all the elements in a list. Even specialized functions can often be quickly defined from general list utilities; for example, summation can be defined by distributing the integer sum operation by a general fold function which folds lists under binary operations.

The fundamental list operations are nil, the empty list, and the infix :: (cons), which appends an element (its left argument) to the head of a list (its right argument). The square-brackets notation for lists (e.g. [1;2;3]), is an abbreviation for a sequence of cons operations terminated by nil; hence [] is another way of writing nil. The system always uses the square-brackets notation when printing lists.

```
nil                 is      []
1 :: [2; 3]         is      [1; 2; 3]
1 :: 2 :: 3 :: nil  is      [1; 2; 3]
```

Other predefined operations on lists include:

- null, which returns true if its argument is nil, and false on any other list.
- hd, which returns the first element of a non-empty list.
- tl, which strips the first element from the head of a non-empty list.
- @ (append), which concatenates lists.

```
null []             is      true
null [1; 2; 3]      is      false
hd [1; 2; 3]        is      1
tl [1; 2; 3]        is      [2; 3]
[1; 2] @ []         is      [1; 2]
[] @ [3; 4]         is      [3; 4]
[1; 2] @ [3; 4]     is      [1; 2; 3; 4]
```

1.5. Functions

All functions take exactly one argument and deliver exactly one result. However arguments and results can be arbitrary structures, thereby achieving the effect of passing multiple arguments and returning multiple results. For example:

```
declaration:    val plus(a,b) = a + b;
```

the function plus above takes a *single* argument (a,b), which is a pair of values. This could seem to be a pointless distinction, but consider the two following ways of using plus:

```
expression:     plus(3,4);

expression:     let val x = 3,4
                in plus x end;
```

The first use of plus is the standard one: two arguments 3 and 4 seem to be supplied to it. However we have seen that ',' is a pair-building operator, and it constructs the pair 3,4 *before* applying plus to it†. This is clearer in the second use of plus, where plus receives a single argument, which

† This is true conceptually; in practice a compiler can optimize away the extra pair constructions in most situations.

is actually a pair of values.

As every function actually has a single argument, the standard function definition looks like val f x = .. . We can put the definition of plus in this form as follows:

```
declaration:    val plus x =
                    let val a,b = x
                    in a + b end;
```

What we did in the previous definitions of plus was to use a *pattern* (a,b) for x, which implicitly associated a and b respectively with the first and second components of x . These patterns come in many forms. For example a pattern [a;b;c] matches a list of exactly three elements, which are bound to a, b and c; a pattern first::rest matches a non-empty list whose first element is associated with first, and the rest to rest; similarly first::second::rest matches a list of at least two elements; etc. The most common patterns are of course tuples, like (a,b,c), but more complicated patterns can be constructed by nesting, like ([a;b],c,(d,e)::_). The special pattern _ matches any value without establishing any binding. Patterns can conveniently replace the use of selector functions for unpacking data.

Patterns do not only appear in function parameters: they can also be used in simple variable declarations:

```
declaration:    val f[a;b;c] = a,b,c;

declaration:    val a,b,c = f[1;2;3];

bindings:           val a = 1 : int
                    val b = 2 : int
                    val c = 3 : int
```

In the above example we have a function f returning three values, and the pattern a,b,c is used to unpack the result of f[1;2;3] into its components.

Recursive functions are defined by placing the keyword rec in a declaration, just before the function definition:

```
declaration:    val rec factorial n =
                    if n = 0
                    then 1
                    else n * factorial(n-1);

binding:            val factorial : int -> int
```

The keyword rec is needed to identify the recursive occurrence of the identifier factorial with its defining occurrence. If rec is omitted, the identifier factorial is searched for in the environment outside the definition; in this case there is no previous definition of factorial, and an error would be reported.

The effect of rec propagates to whole groups of definitions; this is how mutually recursive functions are defined:

```
val rec f a = .. g ..
and g b = .. f .. ;
```

Functions can also be defined by *case analysis*, as a sequence of pattern-action pairs separated by vertical bars:

```
declaration:    val rec summation nil = 0 |
                    summation (head :: tail) = head + summation tail;

binding:            val summation : int list -> int
```

The patterns are evaluated sequentially from left to right. When a pattern matches the argument, the variables in the pattern are bound to the respective parts of the argument, and the corresponding action is executed. It there are several patterns matching some argument, only the first one is activated. If all patterns fail to match some argument, a run-time exception occurs.

1.6. Polymorphism

A function is said to be *polymorphic* when it can work uniformly over a class of arguments of different data types. For example consider the following function, computing the length of a list:

```
declaration:    val rec length nil = 0 |
                    length (_ :: tail) = 1 + length tail;

binding:            val length : 'a list -> int

expression:     length [1; 2; 3], length ["a"; "b"; "c"; "d"];

result:             3,4 : int * int
```

The type of length contains a *type variable* ('a), indicating that any kind of list can be used; e.g. an integer list or a string list. Any identifier or number prefixed by a prime ', and possibly containing more primes, is a type variable.

A type is called *polymorphic*, or a *polytype*, if it contains type variables, otherwise it is called *monomorphic*, or a *monotype*. A type variable can be replaced by any ML type to form an *instance* of that type, for example, int list is a monomorphic instance of 'a list. The instance types can contain more type variables, for example ('b * 'c) list is a polymorphic instance of 'a list.

Several type variables can be used in a type, and each variable can appear several times, expressing contextual relationships between components of a type; for example, 'a * 'a is the type of all pairs having components of the same type. Contextual constraints can also be expressed between arguments and results of functions, like in the identity function, which has type 'a -> 'a, or the following function which swaps pairs:

```
declaration:    val swap(x,y) = y,x;

binding:            val swap : ('a * 'b) -> ('b * 'a)

expression:     swap([],"abc");

result:             "abc",[] : string * ('a list)
```

Incidentally, you may notice that the empty list [] is a polymorphic object of type 'a list, in the sense that it can be considered an empty integer list, or an empty string list, etc..

In printing out polymorphic types, the ML system uses the type variables 'a, 'b, etc. in succession (they are pronounced 'alpha', 'beta', etc.), starting again from 'a at every new top-level declaration. After 'z there are ''a, .. ''z, '''a, .. '''z, etc., but these are rarely necessary. Type variables are really anonymous objects, and it is not important how they are expressed as long as the contextual relations are clear.

Several primitive functions are polymorphic. For example, we have already encountered the list

operations, whose types are:

```
nil     : 'a list
::      : ('a * 'a list) -> 'a list
null    : 'a list -> bool
hd      : 'a list -> 'a
tl      : 'a list -> 'a list
@       : ('a list * 'a list) -> 'a list
```

Note that if these operators were not polymorphic, we would need different primitive operators for all possible types of list elements! The 'a shared by the two arguments of :: (cons) prevents any attempt to build lists containing objects of different types.

One can always determine the type of any ML function or object just by typing its name at the top level; the object is evaluated and, as usual, its type is printed after its value.

```
expression:     [];

result:             [] : 'a list

expression:     hd;

result:             fun : ('a list) -> 'a
```

1.7. Higher-order functions

ML supports higher-order functions, i.e. functions which take other functions as arguments or deliver functions as results. A good example of use of higher-order functions is in *partial application*:

```
declaration:    val times a b = a * b;

binding:            val times : int -> (int -> int)

expression:     times 3 4;

result:             12 : int

declaration:    val twice = times 2;

binding:            val twice : int -> int

expression:     twice 4;

result:             8 : int
```

Note the type of times, and how it is applied to arguments: times 3 4 is read as (times 3) 4. Times first takes an argument 3 and returns a function from integers to integers; this function is then applied to 4 to give the result 12. We may choose to give a single argument to times (this is what is meant by partial application) to define the function twice, and supply the second argument later.

The function composition function, defined below, is a good example of partial application (it also has an interesting polymorphic type). Note how patterns are used in its definition.

```
declaration:    val comp (f,g) x = f (g x);
```

```
binding:        val comp : (('a -> 'b) * ('c -> 'a))  -> ('c -> 'b)

declaration:    val fourtimes = comp(twice,twice);

binding:        val fourtimes : int -> int

expression:     fourtimes 5;

result:         20 : int
```

Function composition comp takes two functions f and g as arguments, and returns a function which when applied to an argument x yields f (g x). Composing twice with itself, by partially applying comp to the pair twice,twice, produces a function which multiplies numbers by four. Function composition is also a predefined operator in ML; it is called o (infix), so that the composition of f and g can be written f o g.

Suppose now that we need to partially apply a function f which, like plus, takes a pair of arguments. We could simply redefine f as val f a b = f(a,b): the new f can be partially applied, and uses the old f with the expected kind of arguments.

To make this operation more systematic, it is possible to write a function which transforms any function of type, ('a * 'b) -> 'c (which requires a pair of arguments) into a function of type 'a -> ('b -> 'c) (which can be partially applied); this process is usually called *currying* a function:

```
declaration:    val curry f a b = f(a,b);

binding:        val curry : (('a * 'b) -> 'c) -> ('a -> ('b -> 'c))

declaration:    val curryplus = curry plus;

binding:        val curryplus : int -> (int -> int)

declaration:    val successor = curryplus 1;

binding:        val successor : int -> int
```

The higher-order function curry takes any function f defined on pairs, and two arguments a and b, and applies f to the pair (a,b). If we now partially apply curry to plus, we obtain a function curryplus which works exactly like plus, but which can be partially applied.

1.8. Exceptions

Certain primitive functions may raise *exceptions* on some arguments. For example, division by zero interrupts the normal flow of execution and escapes at the top level with an error message informing us that division failed:

```
expression:     1 div 0;

exception:          Exception: div
```

Another common exception is raised when trying to extract the head of an empty list:

```
expression:     hd [];

exception:          Exception: hd
```


Every exception is associated with an *exception string*, which describes the reason of failure: in this case "hd" and in the previous example "div".

Exceptions can be *trapped* before they propagate all the way to the top level, and an alternative value can be returned:

```
expression:     hd [] ? 3;

result:             3 : int
```

The question mark operator traps all the exceptions which happen during the execution of the expression on its left, and returns the value of the expression on its right. If the left hand side does not raise exceptions, then its value is returned and the right hand side is ignored.

Exceptions can be trapped selectively by a double question mark, followed by a list of strings: only exceptions with strings appearing in that list are trapped, while the other exceptions are propagated outward.

```
expression:     hd [] ?? ["hd"; "tl"] 3;

result:             3 : int

expression:     1 div 0 ?? ["hd"; "tl"] 3;

exception:          Exception: div
```

The user can generate exceptions by the **escape** keyword followed by a string, which is the exception reason.

```
expression:     escape "fail";

exception:          Exception: fail

expression:     hd [] ? escape "emptylist";

exception:          Exception: emptylist
```

There is no real distinction between system and user generated exceptions: both can be trapped and are reported at the top level in the same way.

1.9. Types

Types describe the *structure* of objects, i.e. whether they are numbers, tuples, lists, strings etc. In the case of functions, they describe the structure of the function arguments and results, e.g. int to int functions, string list to bool functions, etc.

A type only gives information about attributes which can be computed at compile time, and does not help distinguishing among different classes of objects having the same structure. Hence the set of positive integers is not a type, nor is the set of lists of length 3.

On the other hand, ML types can express structural relations inside objects, for example that the right part of a pair must have the same type (whatever that type is) as the left part of the pair, or that a function must return an object of the same type as its argument (whatever that type may be).

Types like bool, int, and string, which do not show any internal structure, are called *basic*. Compound types are built by *type operators* like * (cartesian product), list (list) and -> (function space). Type operators are usually infix or suffix: int * int, int list and int -> int are the types of

integer pairs, lists and functions. Strictly speaking, basic types are type operators which take no arguments. Type operators can be arbitrarily nested: e.g. ((int -> int) list) list is the type of lists of lists of integer functions.

Type variables can be used to express polymorphic types. Polytypes are mostly useful as types of functions, although some non-functional objects, like [] : 'a list, are also polymorphic. A typical example of polymorphic function is hd : 'a list -> 'a, which extracts the first element (of type 'a) of a list (of type 'a list, in general). The type of hd tells us that hd can work on any list, and that the type of the result is the same as the type of the elements of the list.

Every type denotes a *domain*, which is a set of objects, all of the given type; for example int * int is (i.e. denotes) the domain of integer pairs, and int -> int is the domain of all integer functions. An object can have several types, i.e. it can belong to several domains. For example the identity function (fun x. x) has type int -> int as it maps any object (of type integer) to itself (of type integer), but it is also has the type bool -> bool for the same reason. The most general type for the identity function is 'a -> 'a because all the types of the identity are instances of it. 'a -> 'a gives more information than int -> int, for instance, because the former encompasses all the types that the identity function can have and thus expresses all the ways that the identity function can be used. Hence 'a -> 'a is a 'better' type for (fun x. x), although int -> int is not wrong. The ML typechecker always determines the 'best' type for an expression, i.e. the one which corresponds to the smallest domain, given the information contained in that expression.

A type constraint can be appended to an expression, in order to *specify* the type of the expression:

```
expression:     3 : int;

result:             3 : int

expression:     [3,4; (5,6) : int * int];

result:             [(3,4); (5,6)] : (int * int) list
```

In the above examples, the type specifications following ':' do not really have any effect. The types are independently inferred by the system and checked against the specified types. Any attempt to specify a type incorrectly will result in a type error:

```
expression:     3 : bool;

type error:         Type Clash in:    3 : bool
                    Looking for :     bool
                    I have found :    int
```

However, a type specification can restrict the types inferred by the system by constraining polymorphic objects or functions:

```
expression:     [] : int list;

result:             [] : int list

expression:     (fun x. x) : int -> int;

result:             fun : int -> int
```

note that the type normally inferred for [] is 'a list and for (fun x. x) is 'a -> 'a.

Type specifications can be used in bindings:


```
declaration:    val (a : int) = 3;

declaration:    val f (a : int, b : int) : int = a + b;

declaration:    val f ((a,b) : int * int) = (a + b) : int;
```

The last two examples are equivalent. A ': *type*' just before the = of a function definition refers to the result type of the function.

1.10. Concrete types

A *concrete type* is a type for which *constructors* are available, together with other operations. A constructor is an operation which creates elements of some concrete type, by assembling simpler objects of other types.

Constructors are invertible functions: whatever a cofistructor does, can be undone by disassembling the parts which had been put together by the constructor. This allows constructors (or actually, their inverses) to be used in patterns to destructure data. We have already seen examples of this dual usage concerning the tuple constructor ' .. , .. ' and the list constructors nil and :: (cons).

A concrete type and its constructors should be considered as a single conceptual unit. Whenever a new concrete type is defined, its constructors are defined at the same time. Wherever a concrete type is known, its constructors are also known.

New concrete types and their constructors can be defined by *type declarations*. A type declaration defines at the same time a new type name, the names of the constructors for that type and the structure of the new type. In general a new type can consist of several alternatives, separated by '|'; for each of them we have a separate constructor, followed by the keyword 'of' and the type of the argument of that constructor. The keyword 'of' and the succeding type can be omitted: in this case the constructor is also called a *constant* of the new type.

For example, money can be a coin of some value (in cents), a bill of some value (in dollars), a check of some bank for some amount (in cents), or the absence of money.

```
declaration:    type money = nomoney | coin of int | bill of int | check of string * int;

bindings:           type money = nomoney | coin of int | bill of int | check of string * int
                    con nomoney : money
                    con coin : int -> money
                    con bill : int -> money
                    con check : (string * int) -> money
```

Here nomoney, coin, bill and check are money constructors (nomoney is a money constant). Constructors can be used like ordinary functions in expressions:

```
declaration:    val nickel = coin 5
                and dime = coin 10
                and quarter = coin 25;
```

But they can also be used in patterns:

```
declaration:    val   amount nomoney  = 0 |
                      amount (coin cents) = cents |
                      amount (bill dollars) = 100 * dollars |
                      amount (check(bank,cents)) = cents;

binding:                  val amount : money -> int
```

Note that quarter is not a constructor, and we cannot have a clause 'amount quarter = 25' in the above definition (quarter would be interpreted as a normal variable, like x).

A type can be entirely made of constants, in which case we have something similar to Pascal enumeration types. A type can also have a single constructor; then the type definition can be considered as an *abbreviation* for the type following of.

```
declaration:    type color = red | blue | yellow;

declaration:    type point = point of int * int;
```

If the definition of a type involves type variables, the type is called *parametric* and all the type variables used on the right hand side of the = must be listed on the left hand side as type parameters: here are type operators with one and two type parameters:

```
declaration:    type 'a predicate = predicate of 'a -> bool
                and ('b,'c) leftprojection = leftprojection of ('b * 'c) -> 'b;

bindings:           type 'a predicate = 'a -> bool
                    con predicate : ('a -> bool) -> ('a predicate)
                    type ('b,'c) leftprojection = leftprojection of ('b * 'c) -> 'b
                    con leftprojection : (('b * 'c) -> 'b) -> ('b,'c) leftprojection
```

Recursive types are introduced by the keyword rec. Here is how the predefined list type is defined:

```
declaration:    type rec 'a list = nil | :: of 'a * 'a list;

bindings:           type 'a list = nil | :: of 'a * ('a list)
                    con nil : 'a list
                    con :: : ('a * ('a list)) -> ('a list)
```

1.11. Abstract types

An abstract data type is a type with a set of operations defined on it. The structure of an abstract type is hidden from the user of the type, together with the all the constructors of the underlying structure. The associated operations are the only way of creating and manipulating objects of that type. However, the hidden structure and constructors are available while defining the operations.

An abstract data type provides an interface between the use and the implementation of a set of operations. The structure of the type, and the implementation of the operations, can be changed while preserving the external meaning of the operations.

For example we can define an abstract type (additive-) color which is a combination of three primary colors which can each have an intensity in the range 0..15:

```
declaration:    abstype color = blend of int * int
                with val white  = blend(0,0,0)
                and red = blend(15,0,0)
                and blue = blend(0,15,0)
                and yellow = blend(0,0,15)
                and mix (parts: int, blend(red,blue,yellow): color,
                         parts': int, blend(red',blue',yellow'): color) : color =
                    if parts < 0 or parts' < 0 then escape "mix"
                    else  let val totalparts = parts + parts'
                          in blend(
```



```
                              (parts*red + parts'*red') div totalparts,
                              (parts*blue + parts'*blue') div totalparts,
                              (parts*yellow + parts'*yellow') div totalparts)
                    end
end;

bindings:       abstype color
                val white = - : color
                val red = - : color
                val blue = - : color
                val yellow = - : color
                val mix : (int * color * int * color) -> color
```

Composite colors can be obtained by mixing the primary colors in different proportions:

```
declaration:    val green = mix(2,yellow,1,blue)
                and black = mix(1,red,2,mix(1,blue,1,yellow))
                and pink  = mix(1,red,2,white);
```

The bindings determined by the abstract type declaration are: an abstract type color whose internal structure is hidden; four colors (white, red, blue and yellow, printed as -) which are abstract objects whose structure is hidden; and a function to mix colors. No other operation can be applied to colors (unless defined in terms of the primitives), not even equality: if needed, an equality on colors can be defined as part of the abstract type definition.

An operation like mix, which takes abstract objects and produces abstract objects, typically uses patterns to get the representations of its arguments, manipulates the representations, and then uses constructors to produce the result.

As we said, the structure of abstract types and objects is hidden; for example val redpart,_,_ = pink does not give the intensity of red in that color, but produces a type error:

```
expression:     val redpart,_,_ = pink;

type error:         Type Clash in:    ((redpart,_,_) = pink)
                    Looking for :     'a * 'b * 'c
                    I have found :    color
```

The correct thing to do is val blend(redpart,_,_) = pink, but only where blend is available.

This protection mechanism allows one to change the representation of an abstract type without affecting the programs which are already using that type. For example we could have three pigments "red", "blue" and "yellow", and let a color be a list of pigments, where each pigment can appear at most 15 times; white would be blend [], etc. If we define all the operations appropriately, nobody will be able, from the outside, to distinguish between the two implementations of colors.

1.12. Interacting with the ML system

ML is an interactive language. It is entered by typing ml as a Unix command, and it is exited by typing *Control-D*.

Once the system is loaded, the user types phrases (i.e. expressions, top level declarations or commands), and some result or acknowledgement is printed. This cycle is repeated over and over, until the system is exited.

Every top-level phrase is terminated by a semicolon. A phrase can be distributed on several lines by breaking it at any position where a blank character is accepted. Several phrases can also be written on the same line.

There is no provision in the ML system for editing or storing programs or data which are created during an ML session. Hence ML programs and definitions are usually prepared in text files and loaded interactively. The use of a multi-window editor like Emacs is strongly suggested: one window may contain the ML system, and other windows contain ML definitions which can be modified (without having to exit ML) and reloaded when needed, or they can even be directly pasted from the text windows into the ML window.

An ML source file looks exactly like a set of top-level phrases, terminated by semicolons. A file prog.ml containing ML source code can be loaded by the use top-level command:

```
use "prog.ml";
```

where the string surrounded by quotes can be any Unix file name or path. The files loaded by use may contain more use commands (up to a certain stacking depth), so that the loading of files can be cascaded. The file extension .ml is normally used for ML source files, but this is not compulsory.

When working interactively at the top level, the first two characters of every line are reserved for system prompts.

```
- val a = 3
  and f x = x + 1;
> val a = 3 : int
| val f : int -> int

- a + 1;
  4 : int
```

'- ' is the normal system prompt, indicating that a top-level phrase is expected, and ' ' is the continuation prompt indicating that the phrase on the previous line has not been completed. Again, all the top level expressions and declarations must be terminated by a semicolon.

In the example above, the system responded to the first definition by confirming the binding of the variables a and f; '> ' marks the confirmation of the a new binding, followed '| ' on the subsequent lines when many variables are defined simultaneously. In general, the prompts '> ' and '| ' are followed by a variable name, an = sign, the new value of the variable, a ':', and finally the type of the variable. If the value is a functional object, then the = sign and the value are omitted.

When an expression is evaluated, the system responds with a blank prompt ' ', followed by the value of the expression (just fun for functions), a ':' and the type of the expression.

Every phrase-answer pair is followed by a blank line.

The variable it is always bound to the value of last expression evaluated; it is undefined just after loading the system, and is not affected by declarations or by computations which for some reason fail to terminate.

```
- 1 + 1;
  2 : int

- it;
  2 : int

- it + 1;
  3 : int

- val a = 5;
> val a = 5 : int
```



```
- it;
  3 : int

- it + z;
Unbound Identifier: z

- it;
  3 : int

- 1 div 0;
Exception: div

- it,it;
  3,3 : int * int
```

The it variable is very useful in interaction, when used to access values which have 'fallen on the floor' because they have been computed as expressions but have not been bound to any variable. it is not a keyword or a command, but just a plain variable (you can even temporarily redefined it by typing val it = *exp*;). Every time that an expression *exp*; is entered at the top level, this is considered an abbreviation for val it = *exp*;. Hence it is statically scoped, so that old its and new its are totally independent and simply hide each other.

1.13. Errors

Syntax errors report the fragment of code which could not be parsed, and often suggest the reason for the error. The erroneous program fragment is at most one line long; ellipses appear at its left when the source code was longer than a line. It is important to remember that the error was found at the *extreme right* of the error message.

```
- if true then 3;
Syntax Error: if true then 3;
I was expecting an "else"
```

Identifiers, type identifiers and type variables can be undefined; here are the messages given in those situations (type variables must be present on the left of a type definition whenever they are used on the right):

```
- noway;
Unbound Identifier: noway

- 3 : noway;
Unbound Type Identifier: noway

- type t = 'noway
Unbound Type Variable: 'noway
```

Type errors show the context of occurrence of the error and the two types which could not be matched. The context might not correspond exactly to the source code because it is reconstructed from internal data structures.

```
- 3 + true;
Type Clash In:     (3 + true)
Looking for :      int
I have found :     bool
```

The example above is a common matching type error. A different kind of matching error can be generated by self-application, and in general by constructs leading to circularities in the type structures:

```
- fun x. x x;
Type Clash  in:   (x x)
Attempt to build a self-referential type
by equating var:  'a
to type expr:     'a -> 'b

- val rec f _ = f;
Type Clash  in:   f _ = f
Attempt to build a self-referential type
by equating var:  'a
to type expr:     'b -> 'a
```

Less common error messages are described in section "Error messages".

2. Lexical matters

The following lexical entities are recognized: identifiers, keywords, integers, strings, delimiters, type variables and comments. See appendix "Syntax of lexical entities" for a definition of their syntax.

An identifier is either (i) a sequence of letters, numbers, primes ' and underscores '_' not starting with a digit, or (ii) a sequence of special characters (like *, @, /, etc.).

Keywords are lexically like identifiers, but they are reserved and cannot be used for program variables. The keywords are listed in appendix "Keywords".

Integers are sequences of digits.

Strings are sequences of characters delimited by string quotes '"'. String quotes and non-printable characters can be introduced in strings by using the escape character '\'. The details of the escape mechanism are described in appendix "Escape sequences for strings".

Type variables are only accepted in type expressions. They are identifiers starting with the character '. For example: ', ', 'a 'tyvar, etc.

Delimiters are parentheses ((, [,) and]), and other one-character punctuation marks like '.' and ';'. Delimiters never stick to any other character, so that no space is ever needed around them.

Comments are enclosed in curly brackets '{' and '}', and can be nested.

3. Expressions

Expressions denote values, and have a type which is statically determined. Expressions can be constants, identifiers, data constructors, conditionals, fun-expressions, function applications, infix operator applications, scope blocks, while expressions, case expressions, exception and trap expressions, and type specifications. All these different kinds of expressions are described in this section.

3.1. Unit

The expression () (called *unity*) denotes the only object of type unit.

```
()          : unit          (constant)
```

The unit type is useful when defining functions with no argument or no values: they take a unit argument or produce a unit result. There are no primitive operations on unity.

The unit type is not strictly primitive; it could be introduced by the declaration:


```
declaration:    type unit = ();

bindings:       type unit = ()
                con () : unit
```

except that the above is not legal ML syntax: user defined constants must be identifiers, and () is treated specially.

3.2. Booleans

The predefined constants

```
true, false     : bool          (constants)
```

denote the respective boolean values. Boolean operators are:

```
not             : bool -> bool
&, or           : (bool * bool) -> bool     (infixes)
```

Both arguments of & and or are always evaluated. In some languages & and or evaluate only one argument when this is sufficient to determine the result; the same effect can be achieved in ML by using nested conditional expressions.

The boolean type is not primitive; it can be defined in ML as:

```
declaration:    type bool = false | true;

bindings:       type bool = false | true
                con false : bool
                con true : bool
```

All the boolean operations (including if-then-else) can then be defined or simulated by case analysis.

3.3. Integers

The integer operators are:

```
.. ~2, ~1, 0, 1, 2, ..   : int                  (constants)
~                        : int -> int
+, -, *, div, mod        : (int * int) -> int   (infixes)
>, >=, <=, <             : (int * int) -> bool  (infixes)
```

Negative integers are written ~3, where '~' is the complement function (while '-' is difference). Integers have unbounded precision; arithmetic exceptions can only be generated by integer division div and module mod, which escapes with string "div" when their second argument is zero.

The integer type could in principle be defined in ML as:

```
declaration:    type rec nat = one | succ of nat;

declaration:    type int =  neg of nat | zero | pos of nat;
```

3.4. Tuples

A *tuple* is a fixed-length heterogeneous sequence of values. The unity value (i.e. ()) might be considered as a zero-length tuple. There are no tuples of length one; they are identified with their unique component. Tuples of length two or more are written as sequences of values separated by commas:

syntax: $exp_1, .. , exp_n$ $n \geq 2$

Tupling is an n-ary operation; in order to build non-flat tuples of tuples, parentheses must be used. For example the following is not a 4-tuple: it is a triple, whose second component is a pair:

```
expression:   1, (2, 3), 4;

result:       1,(2,3),4 : int * (int * int) * int
```

The type of an n-ary tuple is an n-ary cartesian product, denoted by the symbol *. Parentheses can be used to express non-flat cartesian products.

There are no primitive functions on tuples, except the basic use of commas to build them (note that it is not possible to define a function which extracts the first element of a tuple for tuples of any length). Destructuring must be done by pattern matching:

```
declaration:  val a,b,c = 1,2,3;

result:       a = 1 : int
              b = 2 : int
              c = 3 : int
```

The type of a tuple is the cartesian product of the types of its components:

typing rule: if $e_1 : t_1$ and .. and $e_n : t_n$
then $(e_1 , .. , e_n) : t_1 * .. * t_n$

Tuples are one of the few really fundamental types in ML; they cannot be explained in terms of other constructs in the language.

3.5. Lists

A list of elements exp_i is written:

syntax: $[exp_1 ; .. ; exp_n]$ $n \geq 0$

with [] as the empty list. All the elements of a list must have the same type, otherwise a compile-time type error will occur.

The basic constructors for lists are:

```
nil     : 'a list                      (constant)
::      : ('a * 'a list) -> 'a list    (infix constructor)
```

- nil is the empty list (the same as []).
- :: (cons) adds an element to a list (e.g. 1::[2;3;4] is the same as [1;2;3;4]).

Some other frequent operations on lists are predefined in the system:


```
null            : 'a list -> bool
hd              : 'a list -> 'a
tl              : 'a list -> 'a list
length          : ('a list) -> int
@               : ('a list * 'a list) -> 'a list        (infix)
rev             : ('a list) -> ('a list)
map             : ('a -> 'b) -> (('a list) -> ('b list))
fold, revfold   : (('a * 'b) -> 'b) -> (('a list) -> ('b -> 'b))
```

- null tests whether a list is empty.
- hd (head) extracts the first element of a list; it escapes with string "hd" on empty lists.
- tl (tail) returns a list where the first element has been removed; it escapes with string "tl" on empty lists.
- length returns the length of a list.
- @ (append) appends two lists (e.g. [1;2]@[3;4] is the same as [1;2;3;4]).
- rev returns the reverse of a list.
- map takes a function and then a list, and applies the function to all the elements of the list, returning the list of the results (i.e. map f $[e_1 ; .. ; e_n]$ is $[f\ (e_1); .. ; f\ (e_n)]$).
- fold maps a binary function to an n-ary function over a list (e.g. it can map + to the function computing the sum of the elements of a list), it takes first a binary function, then the list to accumulate, then the value to return on empty lists (i.e. fold f $[e_1 ; .. ; e_n]$ e is $f\ (e_1, .. f\ (e_n,e)..)$).
- revfold is just like fold but accumulates in the opposite direction (i.e. revfold f $[e_1 ; .. ; e_n]$ e is $f\ (e_n, .. f\ (e_1,e)..)$).

The type of lists can be defined in ML as:

```
declaration:  infix :: ;

declaration:  type rec 'a list = nil | :: of int * (int list);

bindings:     type 'a list = nil | :: of int * (int list);
              con nil : 'a list
              con :: : ('a * ('a list)) -> ('a list)
```

All the list operations can then be defined by case analysis. The bracket notation (e.g. [1;2]) is just an abbreviation for the compositions of list constructors (e.g. 1::2::nil).

typing rule: if $e_1 : t$ and .. and $e_n : t$ then $[e_1 ; .. ; e_n] : t$ list

3.6. Strings

A string constant is a sequence of characters enclosed in quotes, e.g. "this is a string".

Operation on strings are:

```
" ... "         : string                        (constants)
size            : string -> int
extract         : (string * int * int) -> string
explode         : string -> string list
implode         : string list -> string
explodeascii    : string -> int list
implodeascii    : int list -> string
intofstring     : string -> int
stringofint     : int -> string
```

• The escape character for strings is \; the conventions used to introduce quotes and non-printable characters inside strings are described in appendix "Escape sequences for strings".
• size returns the length of a string.
• extract extracts a substring from a string: the first argument is the source string, the second argument is the starting position of the substring in the string (the first character in a string is at position 1), and the third argument is the length of the substring; it escapes with string "extract" if the numeric arguments are out of range.
• explode maps a string into the list of its characters, each one being a 1-character string.
• implode maps a list of strings into a string which is their concatenation.
• explodeascii is like explode, but produces a list of the Ascii representations of the characters contained in the string.
• implodeascii maps a list of numbers interpreted as the Ascii representation of characters into a string containing those characters; it escapes with string "implodeascii" if the integers are not valid Ascii codes.
• intofstring converts a numeric string to the corresponding integer number, negative numbers start with '~'; it may escape with string "intofstring" if the string is not numeric.
• stringofint converts an integer to a string representation of the necessary length; negative numbers start with '~'.

For Ascii characters we intend here full 8-bit codes in the range 0..255.

3.7. Updatable references

Assignment operations act on *reference* objects. A reference object is an updatable pointer to another object. References are, together with arrays, the only data objects which can be side effected; they can be inserted anywhere an update operation is needed in variables or data structures.

References are created by the operator ref, updated by := and dereferenced by !. The assignment operator := always returns unity. Reference objects have type t ref, where t is the type of the object contained in the reference.

```
ref     : 'a -> 'a ref              (constructor)
!       : 'a ref -> 'a
:=      : 'a ref * 'a -> unit       (infix)
```

Here is a simple example of the use of references. A reference to the number 3 is created and updated to contain 5, and its contents are then examined.

```
declaration:    val a = ref 3;

binding:            val a = ref 3 : int ref

expression:     a := 5;

result:             () : unit

expression:     ! a;

result:             5 : int
```

References can be embedded in data structures. Side effects on embedded references are reflected in all the structures which share them:

```
declaration:    type 'a refpair = refpair of 'a ref * 'a ref;
```



```
binding:        type 'a refpair = refpair of ('a ref) * ('a ref)
                con refpair : (('a ref) * ('a ref)) -> ('a refpair)

declaration:    val r = ref 3;
                val p = refpair(r,r);

bindings:           val r = ref 3 : int ref
                    val p = refpair(ref 3,ref 3) : int refpair

expression:     r := 5;

result:             () : unit

expression:     p;

result:             refpair(ref 5,ref 5) : int refpair
```

3.8. Arrays

Arrays are ordered collections of values with a constant access time retrieval operation. They have a lower bound and a size, which are integers, and can be indexed by integer numbers in the range lowerbound ≤ i ≤ lowerbound+size. Arrays can have any positive or null size, but once they are built they retain their initial size.

Arrays over values of type *t* have type *t* array. Arrays can be built over elements of any type; arrays of arrays account for multi-dimensional arrays.

The array primitives are:

```
array       : (int * int * 'a) -> 'a array
arrayoflist : (int * 'a list) -> 'a array
lowerbound  : 'a array -> int
arraysize   : 'a array -> int
sub         : ('a array * int) -> 'a                (infix)
update      : ('a array * (int * 'a)) -> unit       (infix)
arraytolist : 'a array -> 'a list
```

• array makes a constant array (all items equal to the third argument) of size n≥0 (second argument) from a lowerbound (first argument). It escapes with string "array" if the size is negative.
• arrayoflist makes an array out of a list, given a lower bound for indexing.
• lowerbound returns the lower bound of an array.
• arraysize returns the size of an array.
• sub extracts the i-th item of an array. It escapes with string "sub" if the index is not in range.
• update updates the i-th element of an array with a new value. It escapes with string "update" if the index is not in range.
• arraytolist converts an array into the list of its elements.

Arrays are not a primitive concept in ML: they can be defined as an abstract data type over lists of assignable references. This specification of arrays, which is given below, determines the semantics of arrays, but does not have a fast indexing operation. Arrays are actually implemented at a lower level as contiguous blocks of memory with constant time indexing. Here is an ML specification of arrays, semantically equivalent to their actual implementation (see sections "Lists", "Updatable references" and "Lexical declarations" to properly understand this program).

```
infix sub;
infix update;
```

```
export

    abstype array
    val array arrayoflist lowerbound arraysize sub update arraytolist

from

    type 'a array = arr of {lowerbound:} int * {size:} int * {table:} 'a ref list;

    val rec el (n: int, (head :: tail): 'a list) : 'a =
        if n=0 then head else el(n-1,tail);

    val array (lb: int, length: int, item: 'a) : 'a array =
        if length<0 then escape "array"
        else arr(lb,length,list length)
            where rec val list n =
                    if n=0 then [] else (ref item) :: list(n-1)
            end

    and arrayoflist (lb: int, list: 'a list) : 'a array =
        arr(lb,length list,map ref list)

    and lowerbound (arr(lb,_,_): 'a array) : int = lb

    and arraysize (arr(_,size,_): 'a array) : int = size

    and (arr(lb,size,table): 'a array) sub (n: int) : 'a =
        let val n' = n-lb
        in    if n'<0 or n'>=size then escape "sub"
              else !(el(n',table))
        end

    and (arr(lb,size,table): 'a array) update (n: int, value: 'a) : unit =
        let val n' = n-lb
        in    if n'<0 or n'>=size then escape "update"
              else el(n',table):=value
        end

    and arraytolist (arr(_,_,table): 'a array) : 'a list =
        map (op !) table

end;
```

Applicative programming fans should type the following declarations, or introduce them in the standard library:

```
type 'a ref = ref;
type 'a array = array;
val op := = ();
val ! = ();
val arrayoflist = ();
val lowerbound = ();
val arraysize = ();
val op sub = ();
```



```
val op update = ();
val arraytolist = ();
```

This transforms ML into a purely applicative language, by making all the side-effecting operations and types inaccessible.

3.9. Patterns and matches

The left hand side of an = in a value definition can be a simple variable, or a more complex *pattern* involving several distinct variables. Patterns are also used in formal parameters of functions and in case expressions. In all these situations a value has to be decomposed according to its structure; the matching process produces a set of *bindings* by associating the variables in the pattern with the corresponding parts of the value.

A pattern can be a _ sign (matching any value), an identifier (preceded by op if infix), a constant, a constructor, an infix constructor, a list pattern, a tuple pattern, a type specification pattern, or a parenthesized pattern.

```
syntax:     simp_pat ::=
                _
                {op} ide
                con
                [pat; .. ;pat]
                ( pat )

            pat ::=
                simp_pat
                con simp_pat
                simp_pat con simp_pat
                pat, .. ,pat
                pat : type
```

The pattern matching rules are given recursively on the structure of the pattern. The letter k denotes constructors, a and b denote variables, u and v denote values, and p and q denote patterns.

Pattern	Value	Match(Pattern,Value)
_	v	$\emptyset$
a	v	$\{a = v\}$
()	()	$\emptyset$
k	k	$\emptyset$
$k\ p$	$k\ v$	Match(p,v)
$[p_1;\ldots;p_n]$	$[v_1;\ldots;v_n]$	Match(p_1,v_1) ∪ .. ∪ Match(p_n,v_n) $n \geq 0$
$p_1,\ldots,p_n$	$v_1,\ldots,v_n$	Match(p_1,v_1) ∪ .. ∪ Match(p_n,v_n) $n \geq 0$

Example:

```
declaration:    val (a,b) :: [c;_] = [1,2; 3,4; 5,6];

bindings:           val a = 1 : int
                    val b = 2 : int
                    val c = 3,4 : int * int
```

The variables in a single pattern must all be distinct. Patterns are not expressions; they can only

appear as part of value declarations, functions and case expressions.

A *match* is a sequence of pattern-action pairs separated by vertical bars.

syntax: *match* ::=
pat . *exp* | .. | *pat* . *exp*

Matches are used to destructure values and to execute fragments of code depending on the form of values: they replace test and selection operations. The patterns in a match are matched in turn, from left to right, against some value *v*. The first successful match is used to destructure *v*, and the variables in the pattern are bound to the respective parts of *v*. Then the corresponding expression is executed, and its result is returned as the result of the matching. If none of the patterns matches *v*, then an exception "match" is raised.

Matches are not expressions; they can only appear in function expressions, case expressions and, in a sugared form, in clausal function definitions.

3.10. Functions

Functions can be introduced by a *curried definition*, for partially applicable functions, or by a *clausal definition*, for case analysis:

syntax: {op} *f simp_pat .. simp_pat* {: *type*} = *exp* (curried)

syntax: {op} *f simp_pat* {: *type*} = *exp* | .. | *f pat* {: *type*} = *exp* (clausal)

- The op keyword should be used when *f* is an infix; alternatively *pat f pat'* can be used instead of op *f (pat,pat')*.
- The *simp_pat* patterns are _, variables, constants, lists and parenthesized patterns, in particular they must be parenthesized when they have the form (*pat'* : *type*).
- The optional *type* expressions refer to the type of the result of the function.
- Curried functions cannot be defined by case analysis: if needed, the arguments can be analyzed after the =.

Unnamed functions can be introduced as expressions by the use of the *fun-notation*, which is just a match preceded by the keyword fun:

syntax: fun *match*

A fun expression denotes a function, which can be associated with an identifier by a normal value definition, for example:

```
declaration:    val plus = fun x . fun y . x + y

declaration:    val null = fun nil . true  | a :: b . false;
```

A curried function definition is a sugaring of the first declaration above, and a clausal function definition is a sugaring of the second declaration:

```
declaration:    val plus x y = x + y

declaration:    val plus nil = true  | plus (a :: b) = false;
```

The type of every expression and pattern can be specified by suffixing a colon and a type expression to it (note that parentheses are needed around type specifications in patterns, when they are

not part of larger patterns). To completely specify the type of a fun-expression we may use either of the two forms:

```
expression:     fun (a: bool). fun (b: int, c: int). (if a then b else c) : int;

expression:     (fun a. fun (b,c). if a then b else c) : bool -> ((int * int) -> int);

result:             fun : bool -> ((int * int) -> int)
```

In declarations, we have the following options (and more):

```
declaration:    val f (a: bool) (b: int, c: int) : int = if a then b else c;

declaration:    val f (a: bool) ((b,c) : int * int) = (if a then b else c) : int;

binding:            val f : bool -> ((int * int) -> int)
```

The first form should be preferred; note how the result type of the function precedes the = sign of the definition.

This explicit type information is often redundant. However, the system checks the explicit types against the types it infers from the expressions. Introducing explicit type information is a good form of program documentation, and helps the system in discovering type errors exactly where they arise. Sometimes, when explicit type information is omitted, the system is able to infer a type (not the intended one) from an incorrect program; this 'misunderstanding' is only discovered in some later use of the program and can be difficult to trace back.

The type of a function expression is determined by the type of its binder (or binders) and the type of its body (or bodies):

typing rule: if $p_1 : t$ and .. and $p_n : t$ and $e_1 : t'$ and .. and $e_n : t'$
then (fun $p_1 . e_1 | .. | p_n . e_n$) : $t \rightarrow t'$

The type automatically inferred by the system is the 'most polymorphic' of the typings which obey the typing rules. For example, from the previous typing rule we can infer that (fun x. x) : int -> int, and also that (fun x. x) : 'a -> 'a; the latter type is more polymorphic (in the sense that the int -> int can be obtained by instantiating the type variables of 'a -> 'a); in fact it is the most general typing, and is taken as the default type for (fun x. x).

3.11. Application

There are two lexical categories of identifiers, which determine how functions are syntactically applied to arguments: *nonfix* and *infix*. Any identifier can be declared to fall in one of these categories: see section "Lexical declarations" about how to do this.

Nonfix identifiers are the ordinary ones; they are applied by juxtaposing them with their argument (in this section we use *f* and *g* for functions and *a*, *b*, *c* for arguments):

syntax: *f a*

The expression *a* can also be parenthesized (as any expression can), obtaining the standard application syntax *f* (*a*). It is common *not* to use parentheses when the argument is a simple variable, a string (*f* "abc") or a list (*f* [1;2;3]). It is necessary to use parentheses when the argument is an infix expression like a tuple or an arithmetic operation (*f* (*a*,*b*,*c*), *f* (*a*+*b*)) because the precedence of application is normally greater than the precedence of the infix operators (*f a*,*b* and *f a*+*b* are interpreted as (*f a*),*b* and (*f a*)+*b*). Function application binds stronger than any operator: see the

appendix "Precedence of operators" for details. In case of partial application, the form *f g a*, is interpreted as *(f g) a*, and not as *f (g a)*; note that they are both equally meaningful.

Infix identifiers can be applied in two ways:

```
syntax:     a f b

syntax:     op f (a,b)
```

The first form is the expected one; an infix identifier has always a type matching ('a * 'b) -> 'c. The second possibility derives from the fact that there must be a way of passing an infix operator f as an argument to a function g; g(f) will produce a syntax error, but g(op f) is accepted. In general the op keyword coerces any infix identifier to a nonfix one.

The typing rule for function application states that the type of the argument must match the type of the domain of the function:

```
typing rule:    if f : t -> t' and a : t   then   (f a) : t'
```

For example, (fun x. x) has the type 'a -> 'a and all the instances of it, e.g. int -> int. Hence (fun x. x) 3 has type int.

3.12. Conditional

The syntactic form for conditional expressions is:

```
syntax:     if e1 then e2 else e3
```

The expression e_1 is evaluated to obtain a boolean value. If the result is true then e_2 is evaluated; if the result if false then e_3 is evaluated. Example:

```
expression:     val sign n =
                    if n < 0 then ~1
                    else if n = 0 then 0
                    else 1;
```

The **else** branch must always be present.

The typing rule for the conditional states that the if branch must be a boolean, and that the then and else branches must have compatible types:

```
typing rule:    if e1 : bool and e2 : t and e3 : t
                then   (if e1 then e2 else e3) : t
```

Note that the types of two branches only have to match, not to be identical. If one branch is more polymorphic than the other, than it also has the type of the other (by instantiation of the type variables), and the above rule can be applied.

3.13. Sequencing

When several side-effecting operations have to be executed in sequence, it is useful to use *sequencing*:

```
syntax:     (e1; .. ; en)          n≥2
```

(the parentheses are needed), which evaluates e_1 .. e_n in turn and returns the value of e_n. The

type of a sequencing expression is the type of e_n. Example:

```
expression:     (a := 4;  a := !a div 2;  !a)

result:             2 : int
```

Note that (e) is equivalent to e, and that () is the unity.

```
typing rule:    if e : t   then   (e1; .. ; en; e) : t          n ≥1
```

3.14. While

The while construct can be used for iterative computations. It is included in ML more as a matter or style and convenience than necessity, as all *tail recursive* functions (e.g. functions which end with a recursive call to themselves or to other functions) are automatically optimized to iterations by the compiler.

The while construct has two parts: a test, preceded by the keyword while and a body, preceded by the keyword do. If the test evaluates to true then the body is executed, otherwise () is returned. This process is repeated until the test yields false (if ever) or an exception occurs.

```
syntax:     while e1 do e2
```

The result of a terminating while construct is always (), hence whiles are only useful for their side effects. The body of the while is also expected to yield () at every iteration.

```
typing rule:    if e1 : bool and e2 : unit
                then   (while e1 do e2) : unit
```

As an illustration, here is an iterative definition of factorial which uses two local assignable variables count and result:

```
declaration:    val fact n =
                    let val count = ref n
                        and result = ref 1
                    in      (while !count <> 0 do
                                (result := !count * !result;
                                 count := !count-1);
                             !result)
                    end;
```

3.15. Case

A case expression is just a different syntax for pattern matching: see section "Patterns and matches" for a description of *matches*.

```
syntax:     case exp of match
```

The expression *exp* is matched against the *match*, which is a set of pattern-action pairs; the first pattern to match is activated, and the corresponding action is evaluated and returned as the result of the case expression.

```
declaration:    type color = red | purple | yellow
```

```
and fruit = apple | plum | banana;
```

bindings:

```
type color = red | purple | yellow
type fruit = apple | plum | banana
```

declaration:

```
val fruitcolor (fruit: fruit): color =
    case fruit of
        apple. red |
        banana. yellow |
        plum. purple;
```

binding:

```
val fruitcolor : fruit -> color
```

The case construct is a convenient form of saying if fruit=apple then red else if fruit=banana then yellow else if fruit=plum then purple else escape"match" (else escape"match is never executed in this case because the list of cases is exhaustive).

The typing rule for case is:

typing rule: if $x : t'$ and $p_1 : t'$ and .. and $p_n : t'$
and $e_1 : t$ and .. and $e_n : t$
then (case x of $p_1 . e_1 | .. | p_n . e_n$) : t

This says that all the patterns must have the same type as value being inspected, and that the values returned for each case must have the same type.

3.16. Scope blocks

A *scope block* is a control construct which introduces new variables and delimits their scope. Scope blocks have the same function of begin-end constructs in Algol-like languages, but they have a fairly different flavor due to the fact of being expressions returning values, instead of groups of statements. There are two kinds of scope blocks:

syntax: let *declaration* in *exp* end

syntax: *exp* where *declaration* end

The let construct introduces new variable bindings in the *declaration* part which can be used in the *exp* part (and there only). Newly introduced variables hide externally declared variables having the same name for the scope of *exp*; the externally declared variables remain accessible outside *exp*.

The where construct behaves just like let; it simply inverts the order in which the expression and the declaration appear.

The value returned by a scope block is the value of its expression part. Similarly the type of a scope block is the type of its expression part. The different kinds of declarations are described in section "Declarations".

typing rule: if $e : t$ then (let d in e end) : t

3.17. Exceptions and traps

An *exception* (sometimes also called a *failure*) can be raised by a system primitive or by a user program. When an exception is raised, the execution of the current expression is abandoned, and an *exception packet* is propagated outward, tracing back along the history of function calls and expression evaluations which led to the exception. If the exception is allowed to propagate up to the top

level, a *failure message* is printed. In this version of the ML system, exception packets are limited to string values, which are then called *exception strings*, and failure messages print the contents of the exception strings.

exception: Exception: *reason*

where *reason* is the content of the exception string mentioned above. Note that the exception string is not the value of a failing expression: failing expressions have no value, and exception strings are manipulated by mechanisms which are independent of the usual value manipulation constructs. The exception string is often the name of the system primitive or user function which raised the exception.

User exceptions can be raised by the escape construct:

syntax: escape *exp*

The expression *exp* above must evaluate to a string, which is the exception string.

The propagation of exceptions can only be stopped by the *trap* construct.

The result of the evaluation of e_1 ? e_2 is normally the result of e_1, unless e_1 raises an exception, in which case it is the result of e_2.

The result of the evaluation of e_1 ?? e_2 e_3 (where e_2 evaluates to a string list *sl*) is normally e_1, unless e_1 raises an exception, in which case it is the result of e_3 whenever the exception string is one of the strings in *sl*; otherwise the exception is propagated.

The result of the evaluation of e_1 ?\ v e_2 is normally the result of e_1, unless e_1 raises an exception, in which case it is the result of e_2, where the variable v is associated with the exception string and can be used in e_2.

syntax: e_1 ? e_2

syntax: e_1 ?? e_2 e_3

syntax: e_1 ?\ v e_2

Some system exceptions may be raised at any time, independently of the expression which is being evaluated. They are "interrupt", which is generated by pressing the *DEL* key during execution, and "collect", which is generated when there is no space left. Even these exceptions can be trapped.

The typing rule for escape says that an exception is compatible with every type, i.e. that it can be generated without regard to the expected value of the expression which contains it. The rules for the trap operators state that the exception handler must have the same type as the possibly failing expression, so that the resulting type is the same whether the exception is raised or not.

typing rules: (escape e) : 'a

if $e : t$ and $e' : t$ then (e ? e') : t

if $e : t$ and sl : string list and $e' : t$ then (e ?? sl e') : t

if $e : t$ and v : *string* and $e' : t$ then (e ?\ v e') : t

3.18. Type semantics and type specification

The set of all values is called V: it contains basic values, like integers, and all the composite objects built on elements of V, like pairs, functions, etc. The structure of V can be given by a recursive type equation, of which there are known solutions (+ is disjoint union, * is cartesian product and → is continuous function space):

$$V = Bool + Int + .. + (V + V) + (V * V) + (V \to V) \qquad [1]$$

All functions are interpreted as functions from V to V, so when we say that f has type int -> bool we do not mean that f has domain int and codomain bool in the ordinary mathematical sense. Instead we mean that *whenever f is given an integer argument it produces a boolean result*. This leaves f free to do whatever it likes on arguments which are not integers: it might return a boolean, or it might not. This idea leads to the following definition for function spaces:

$$A \to B = \{f \in V \to V \mid a \in A \text{ implies } f\,a \in B\} \qquad [2]$$

where → is the conventional continuous function space, while -> is the different concept that we are defining. A and B are *domains* included in V, and A -> B is a domain included in V → V, and hence embedded in V by [1].

In this way we can give meaning to monotypes like int -> int, but what about polytypes? Consider the identity function (fun x. x) : 'a -> 'a; whenever it is given an argument of type 'a it returns a result of type 'a. This means that when given an integer it returns an integer, when given a boolean it returns a boolean, etc. Hence by [2], fun x. x belongs to the domains int -> int, bool -> bool, and to $d \to d$ for any domain d. Therefore fun x. x belongs to the intersection of all those domains, i.e. to $\cap_{d \in T}\, d \to d$ (where T is the set of all domains). We can now take the latter expression as the meaning of 'a -> 'a.

In general a polymorphic object of type $\sigma['a]$ belongs to all the domains $\sigma[d/'a]$, and hence to their intersection.

Some surprising facts derive from these definitions. First, 'a is not the type of all objects, as one might expect; in fact the meaning of 'a is the intersection of all domains. The only element contained in all domains is the divergent computation, which is therefore the only object of type 'a.

Second, 'a -> 'a, as a domain, is smaller than any of its instances, for example int -> int. In fact any function returning an 'a when given an 'a, must return an int when given an int (i.e. 'a -> 'a $\subseteq$ int -> int), but an int -> int function is not required to return a boolean when given a boolean (i.e. int -> int $\supset$ 'a -> 'a). Hence a function like (fun x. x+1) : int -> int is not an 'a -> 'a.

Similarly, 'a list as a domain is smaller than int list, bool list, etc. In fact 'a list is the intersection of all list domains and only contains the empty list (and the undefined computation).

When *specifying* a type in ML, by the notation:

syntax: *exp* : *type*

the effect is to take the *meet* of the type of the expression inferred by the system and of the type following ':' specified by the user (if this meet exists) as the type of the expression. The meet of two domains is their intersection.

This implicit join operation explains why the specifications:

expression: 3 : 'a;

result: 3 : int

expression: (fun x. x) : int -> int;

result: fun : int -> int

expression: (fun x. x + 1) : 'a -> 'a;

result: fun : int -> int

are accepted, even if, according to the previous discussion, 3 does *not* have type 'a; (fun x. x) has a better type than int -> int; and (fun x. x + 1) may or may *not* have type 'a -> 'a, according to how it behaves outside the integer domain.

3.19. Equality

The standard equality predicate is *structural equality*: two objects are equal when they are the same atomic object, or all their components are equal.

The mathematical definition of equality over domains is in general undecidable, hence some restrictions are imposed on the objects which can be compared, in order to obtain a decidable predicate. Equality is considered to be a predefined overloaded operator, i.e. an infinite collection of decidable equality operators, whose types are instances of ('a * 'a) -> bool.

Equality is disallowed on polymorphic objects, on functions, on streams and on elements of abstract data types. When needed, these objects can be marked by data having a decidable equality, but then the user must write a special purpose equality routine.

Equality on reference object is not structural equality, but 'sameness', or pointer equality (e.g. ref 3 = ref 3 is false).

Equality over abstract types might be implemented as equality over the corresponding concrete representations. However in this case abstract types would not be transparent to a change of representation (e.g. from an int list to an int->int), as equality would behave differently on the different representations. Also, if we implement abstract sets by concrete multisets, equality could distinguish between sets which should be equal, giving dangerous insights on the chosen representation, as well as not corresponding to set equality.

Here are examples of 'right' and 'wrong' comparisons (at the top level). The 'wrong' comparisons produce compile time type errors:

Right	Wrong
() = ();	
true = false;	
3 = 3;	
"a" = "";	
(2,"a") = (2,"a");	
[] = ([]: int list);	[] = [];
[1;2] = [3];	
(fun a,b. a=b) : string*string -> bool;	fun a,b. a=b;
	fun a. a = fun a. a;
	set[3] = set[3];

where set is an abstract type.

4. Type expressions

A type expression is either a *type variable*, a *basic type*, or the application of a *type operator* to a sequence of type expressions. Type operators are usually suffix, except for cartesian product * and the infix operator ->.

4.1. Type variables

A type variable is an identifier starting with ' and possibly containing other ' characters. Type variables are used to represent polymorphic types.

4.2. Type operators

A basic type is a type operator with no parameters, like int. A parametric type operator, like -> or list, takes one or more arguments which are arbitrary type expressions, as in 'a -> ('a list) list). If an operator takes many parameters, these are separated by commas and enclosed in parentheses, as in ('a, 'b) tree. See appendix "Predefined type identifiers" for a list of the predefined type operators.

5. Declarations

Declarations are used to establish bindings of variables to values. Every declaration is said to *import* some variables, and to *export* other variables. The imported variables are the ones which are used in the declaration, and are usually defined outside the declaration (except in recursive declarations). The exported variables are the ones defined in the declaration, and that are accessible in the scope of the declaration.

A declaration can be a value binding, a type binding, a sequential composition, a private declaration, a module import, a lexical declaration or a parenthesized declaration.

syntax:

declaration ::=
- val *value_binding*
- type *type_binding*
- *abstract_declaration*
- *sequential_declaration*
- *private_declaration*
- *import_declaration*
- *lexical_declaration*
- (*declaration*)

5.1. Value bindings

Value bindings define values and functions, and are prefixed by the keyword val. After val there can be a simple definition, a parallel binding or a recursive binding.

syntax:

value_binding ::=
- *simple_value_binding*
- *parallel_value_binding*
- *recursive_value_binding*

5.1.1. Simple bindings

The simplest form of value binding, called a *value definition*, introduces a single pattern or function:

syntax:

simple_value_binding ::=
- *pat* = *exp*

- {op} *Ide simp_pat .. simp_pat* {: *type*} = *exp*
- {op} *Ide simp_pat* {: *type*} = *exp* | .. | *Ide pat* {: *type*} = *exp*

Each declaration imports the variables used in the expressions exp and exports the variables defined on the left of =. Here are some examples:

declaration: `val x = 2 * 3;`

binding: `val x = 6 : int`

declaration: `val a,b = 4,[];`

bindings:
```
val a = 4 : int
val b = [] : 'a list
```

declaration: `val K x y = x;`

binding: `val K : 'a -> ('b -> 'a)`

declaration: `val null nil = true | null (a :: b) = false;`

binding: `val null : ('a list) -> bool`

Note that function definitions require simple (i.e. possibly parenthesized) patterns as formal parameters. In particular, parameters which have the form of constructors applied to patterns must be parenthesized (e.g. val f (ref a) = a; is legal, but val f ref a = a; is not).

5.1.2. Parallel bindings

To bind the variables $x_1..x_n$ simultaneously to the values of $e_1..e_n$ one can write either of the following declarations:

val $x_1, .., x_n = e_1, .., e_n$;

val $x_1 = e_1$ and .. and $x_n = e_n$;

In the first case we use a composite pattern, while in the second case we use the infix operator and:

syntax:

parallel_value_binding ::=
- *value_binding* and *value_binding*

The meaning of and is that the bindings of the two sides are established 'in parallel', in the sense that the variables exported from the left side are not imported to the right side, and vice versa. The whole construct exports the union of the variables of the two declarations; it is illegal to declare the same variable on both sides of an and.

For example, note how it is possible to swap two values without using a temporary variable:

declaration: `val a = 10 and b = 5;`

bindings:
```
val a = 10 : int
val b = 5 : int
```

declaration: `val a = b and b = a;`

bindings:
```
val a = 5 : int
val b = 10 : int
```

5.1.3. Recursive bindings

The operator rec builds recursive bindings, and it is used to define recursive and mutually recursive functions. The binding rec *d* exports the variables exported by the binding *d*, and imports the variables imported by *d* and the variables exported by *d*.

syntax:
```
recursive_value_binding ::=
    rec value_binding
```

declaration:
```
val rec fib n = if n < 2 then 1 else fib(n-1) + fib(n-2);
```

Note that if rec were omitted, the identifier fib on the right of = would not refer to its defining occurrence on the left of =, but to some previously defined fib value (if any) in the surrounding scope.

A recursive value declaration can only define functions; simple value bindings are not allowed to be recursive. Moreover, all the bindings contained in a rec must be *syntactically* functions, e.g. simple, curried or clausal function bindings, or simple variable bindings having fun-expressions on the right of =. A definition like val rec g = f; is not accepted, even if f is a function.

5.2. Type bindings

Type bindings define new types and their constructors, and are prefixed by the keyword type. After type there can be a definition, a parallel binding or a recursive binding.

syntax:
```
type_binding ::=
    simple_type_binding
    parallel_type_binding
    recursive_type_binding
```

5.2.1. Simple bindings

The simplest form of type binding,introduces a single, possibly parametric, type and its constructors:

syntax:
```
simple_type_binding ::=
    type_lde = type_cases
    type_var type_lde = type_cases
    (type_var, .. ,type_var) type_lde = type_cases
```

syntax:
```
type_cases ::=
    con {of type_exp} | .. | con {of type_exp}
```

Here are examples of types with zero, one and two type parameters:

declaration:
```
type intpair = intpair of int * int;
```

bindings:
```
type intpair = intpair of int * int
con intpair : (int * int) -> intpair
```

declaration:
```
type 'a pair = pair of 'a * 'a;
```


bindings:
```
type 'a pair = pair of 'a * 'a
con pair : ('a * 'a) -> ('a pair)
```

declaration:
```
type ('a, 'b) pairpair = pairpair of ('a * 'b) * ('a * 'b);
```

bindings:
```
type ('a, 'b) pairpair = pairpair of ('a * 'b) * ('a * 'b)
con pairpair : (('a * 'b) * ('a * 'b)) -> (('a, 'b) pairpair)
```

All the type variables appearing on the right of the defining = sign must appear on the left as parameters of the type being defined. The list of parameters on the left of the = sign must be a sequence of distinct type variables.

5.2.2. Parallel bindings

As in value bindings, the and operator can be used to combine type binding in parallel, in the sense that the variables exported from the left side are not imported to the right side, and vice versa.

syntax:
```
parallel_type_binding ::=
    type_binding and type_binding
```

The whole construct exports the union of the variables defined by the two type bindings; it is illegal to declare the same variable on both sides of an and.

5.2.3. Recursive bindings

The operator rec is used to define recursive and mutually recursive types. The binding rec *tb* exports the variables exported by the type binding *tb*, and imports the variables imported by *tb* and the variables exported by *tb*.

syntax:
```
recursive_type_binding ::=
    rec type_binding
```

declaration:
```
type rec 'a tree = leaf of 'a | node of ('a tree) * ('a tree);
```

bindings
```
type 'a tree = leaf of 'a | node of ('a tree) * ('a tree)
con leaf : 'a -> ('a tree)
con node : (('a tree) * ('a tree)) -> ('a tree)
```

If rec were omitted, the type identifier tree on the right of = would not refer to its defining occurrence on the left of =, but to some previosly defined tree type (if any) in the surrounding scope.

5.3. Abstract type declarations

Abstract types can be defined by the *abstype-with* declaration, which is an abbreviation of a special form of private declaration (see section "Private declarations", below).

syntax:
```
abstract_declaration ::=
    abstype type_binding with declaration end
```

An abstract type defined by the abstype keyword is like a concrete type defined by the type keyword, but its constructors are only available in the declaration following the with keyword.

declaration:
```
abstype position = position of int * int
```

```
        with  val origin = position (0,0)
              and stepx (position(x,y)) = position(x+1,y)
              and stepy (position(x,y)) = position(x,y+1)
        end;
```

bindings:

```
abstype position
val origin = - : position
val stepx : position -> position
val stepy : position -> position
```

In this example, a position can only be achieved by small steps from the origin. We cannot 'jump' to an arbitrary position because there is no operation to directly build one.

Here is the definition of a parametric recursive type of binary trees with leaves of type 'a:

declaration:

```
abstype rec 'a tree = leaf of 'a | node of 'a tree * 'a tree
with  val mkleaf a = leaf a
      and mknode (t,t') = node(t,t')
      and isleaf (leaf _) = true | isleaf(node _) = false
      and left (node(t,_)) = t
      and right (node(_,t)) = t
      and tip (leaf a) = a
end;
```

bindings:

```
abstype 'a tree
val mkleaf : 'a -> 'a tree
val mknode : ('a tree * 'a tree) -> 'a tree
val isleaf : 'a tree -> bool
val left : 'a tree -> 'a tree
val right : 'a tree -> 'a tree
val tip : 'a tree -> 'a
```

The constructors leaf and node are only known during the definition of the basic operations on trees, and are not exported outside the abstract type definition. All users of the tree abstract type will be unable to take advantage of the concrete representation of trees given by 'leaf of 'a | node of 'a tree * 'a tree', thus making this type 'abstract'.

The with operator can be preceded by several type definitions connected by and and rec; this allows one to define mutually recursive abstract types:

declaration:

```
abstype rec a = .. b ..
      and b = .. a ..
with .. end;
```

5.4. Sequential declarations

Cascaded, or 'sequential' declarations are provided by the environment operator ';', which makes earlier declarations available inside later declarations.

syntax:

sequential_declaration ::=
declaration ; *declaration*

In d_1 ; d_2, the variables exported by d_1 are imported into d_2, but not vice versa. The variables exported by the whole declaration are the ones exported by d_2, plus the ones exported by d_1 which are not redefined in d_2.

expression:

```
let val a = 10;  val b = a + 5
in b end;
```

result:

```
15 : int
```

5.5. Private declarations

The export declaration allows one to hide some of the bindings of a declaration, while making other bindings available to the outside.

syntax:

private_declaration ::=
export *export_list* .. *export_list* from *declaration* end

syntax:

export_list ::=
abstype *type_ide* .. *type_ide*
type *type_ide* .. *type_ide*
val *ide* .. *ide*

The type identifiers and the value identifiers (which should be declared in declaration), are exported to the outside, while the other type identifiers and value identifiers defined in *declaration* are hidden. In the case of a type export, the constructors of that type are automatically exported. In the case of an abstype export the constructors are not exported. Constructors cannot be listed in a val section.

declaration:

```
export val a c
from  (val a,b = 1,2;
       val c = a + b)
end;
```

bindings:

```
val a = 1 : int
val c = 3 : int
```

declaration:

```
export val increment fetch
from  (val count = ref 0;
       val increment () = count := !count + 1
       and fetch () = !count)
end;
```

bindings:

```
val increment : unit -> unit
val fetch : unit -> int
```

declaration:

```
export abstype increasing_pair
       val increasing_pair high low
from  type increasing_pair = pair of int * int;
       val increasing_pair(x,y) =
             if x>y then escape"increasing_pair"
             else pair(x,y)
       and low (pair(x,y)) = x
       and high (pair(x,y)) = y
end;
```

```
bindings:       abstype increasing_pair
                val increasing_pair : (int * int) -> increasing_pair
                val low : increasing_pair -> int
                val high : increasing_pair -> int
```

In the first example, the variable b is unknown at the top level. The second example shows how a variable can be made private to a function or a group of functions: only the functions increment and fetch have access to count, so that nobody can corrupt count. The third example is an abstract data type increasing_pair: note that the constructor pair has been hidden, so that it is impossible to generate non-increasing pairs.

The 'abstype *tb* with *d*' declaration construct (see section "Abstract declarations") is just an abbreviation for 'export *xl* from type *tb; d* end', where the types defined in *tb* are listed in an abstype export list in *xl*, and the types and values defined in *d* are listed in appropriate export lists in *xl*.

The increasing_pair example above can be written more conveniently as:

```
declaration:    abstype increasing_pair = pair of int * int
                with  val increasing_pair(x,y) =
                          if x>y then escape"increasing_pair"
                          else pair(x,y)
                      and low (pair(x,y)) = x
                      and high (pair(x,y)) = y
                end;

bindings:       abstype increasing_pair
                val increasing_pair : (int * int) -> increasing_pair
                val low : increasing_pair -> int
                val high : increasing_pair -> int
```

5.6. Import declarations

The import environment operator acts on precompiled modules, and it is explained in detail in section "Modules". A declaration import *M* imports no variables from the surrounding environment, and exports the variables exported by the module *M*.

```
syntax:         import_declaration ::=
                    import module_name .. module_name

declaration:    import minmax;

bindings:       val min : (int * int) -> int
                val max : (int * int) -> int
```

where the module minmax contains definitions for the min and max functions.

5.7. Lexical declarations

Special kinds of declaration, introduced by the keywords nonfix and infix are used to specify lexical attributes of identifiers. These declarations have only a lexical meaning; they do not introduce bindings, nor causes evaluations.

All identifiers have a property called *fixity*, which can be *infix* or *nonfix* (i.e. normal). The fixity of an identifier determines the way arguments are syntactically supplied to it (see section "Application" for the patterns of usage). Fixity can be specified in a *lexical* declaration, before the identifier is used in declarations or expressions:


```
syntax:         lexical_declaration ::=
                    infix ide .. ide
                    nonfix ide .. ide

declarations:
                    infix  <--> ;
                    nonfix + ;
```

This example declares an infix operator <--> and puts the operator + back to a status of normal non-infix identifier. These operators can now be uses in expressions and declarations according to their new lexical status.

A lexical declaration can be used in scopes-blocks. In this case the newly defined fixity status is only active in the local scope.

6. I/O streams

Input-output is done on streams. A stream is like a queue; characters can be read from one end and written on the other end. Reads are destructive, and they wait indefinitely on an empty stream for some character to be written. In what follows, a "file" is a file on disk which has a "file name"; a "stream" is an ML object (it is a pair of Unix file descriptors, one open for input and the other one open for output).

```
file        : string -> stream
save        : (string * stream) -> unit
stream      : unit -> stream
channel     : string -> stream
input       : (stream * int) -> string
output      : (stream * string) -> unit
lookahead   : (stream * int * int) -> string
caninput    : stream -> bool
terminal    : stream
```

- Streams are associated with file names in the operating system. The operation file takes a string (a file name) and returns a new stream whose initial content is the content of the corresponding file. It escapes with string "file" if the stream is not available (e.g. the file name syntax is wrong, or the file is locked). If no file exists with that file name, a new empty stream is returned (hence, empty files and streams are indistinguishable from non-existent ones). The same file name can be requested several times; every time a new independent stream is generated.

- A copy of an existing stream can be associated with a file name (i.e. written to a file) by the **save** operation which takes a string (the file name) and a stream and returns unit. The stream is unaffected by this operation. An exception with string "save" is raised if the association cannot be carried out. Reads and writes on streams do not affect the files they come from. Conversely, a **save** operation on a file does not affect the streams which have been extracted from that file; it only affects the result of a subsequent file.

- The operation stream returns a new empty stream. It accounts for temporary (unnamed) files. Moreover, a stream-filename association can be removed by reassociating an empty stream with that file name.

- Input operations are destructive; the characters read are removed from the stream. input takes a stream *s* and a number *n* and returns a string of *n* characters reading them from the stream *s*. If there are less than *n* characters in the stream, it waits indefinitely until more characters are written on the stream. The wait can be interrupted by the *DEL* key producing an "interrupt" exception. caninput returns true if a stream is not empty; the input operations do not fail on empty streams, they wait indefinitely for something to be written on the stream.

• Output operations are constructive; the characters written are appended to the end of the stream. output writes a string of characters at the end of a stream.

• The operation lookahead reads characters from a stream without affecting it. The arguments are like in the string operation extract: the first argument is the source stream, the second argument is the starting position of the string to be extracted from the stream (the first character in a stream is at position 1), and the third argument is the length of the string to be extracted; it escapes with string "lookahead" if the numeric arguments are out of range.

• There are two flavors of streams, which can be called *internal* and *external* streams. Streams returned by file and stream are internal, because ML is the only process which can access them. Other streams, like the predefined terminal stream, are external because the ML system holds only one end of them, while the other end belongs to some external process. Some external streams may be read-only or write-only; the I/O operations escape when trying to read from a write-only only stream, or write to a read-only stream.

• The operation channel will be provided to open new external streams, for example to allow direct communication between two ML systems, or between ML and an external process. The way this will work is still to be defined.

All the above operations may escape for various I/O error. Multiplexed read and multiplexed write operations can be obtained by passing the same stream to several readers and writers respectively (i.e. to different parts of a program).

7. Modules

A *module* is a set of bindings (values, functions and types) which can be compiled and stored away, and later *imported* as a declaration wherever those bindings are needed. Modules are identified by *module names*. Module names can syntactically be represented as simple identifiers, or as strings containing Unix file paths (to access modules in directories other than the current one). Here is a module called Pair which defines a type and two functions.

```
module:     module Pair
            body
                abstype 'a pair = pair of 'a * 'a
                with  val newpair (a,b) = pair(a,b)
                      and left (pair(a,_))  =  a
                      and right (pair(_,b))  =  b
                end
            end;
```

Module definitions can only appear at the top level, and they cannot rely on bindings defined in the surrounding scope (e.g. previous top level definitions). All the bindings used by a module must either be defined in the module itself, or imported from other modules.

The processing of a module definition is called a module *compilation*, whose only effect is to produce a separately compiled version of the module. Module definitions do not evaluate to values, and they do not introduce new bindings.

Modules can be defined interactively, but usually they are contained in external source files. In the latter case, a command 'use "Mod.ml";' (see section "Loading source files") can be used to compile a module definition contained in the file Mod.ml. Once a module is compiled, it can be imported:

```
declaration:    import Pair;

bindings:       module /usr/lib/ml/lib
                abstype 'a pair
                val newpair : ('b * 'b) -> ('b pair)
                val left : 'c pair -> 'c
```



```
                val right : 'd pair -> 'd
```

a declaration import M can also appear in local scopes and inside functions, wherever a declaration is accepted (the binding module /usr/lib/ml/lib above is explained later).

Importing can result in *loading* a module, when that module is imported for the first time in an interactive session, or in *sharing* an already loaded module, all the subsequent times. The loading/sharing mechanism is explained below in "Module sharing".

No evaluation takes place when a module is compiled. Instead, the module body is evaluated every time the module is *loaded*. If the module body contains side effecting operations (such as input/output), they have effect at loading time, i.e. they do not have effect if the module is being shared.

More precisely, we can say that a module is an *environment generator*; loading a module corresponds to generating a new environment, and sharing a module corresponds to using an already generated environment.

7.1. Module hierarchies

A module A can import other modules B, C, etc. The import relations between modules determine a module hierarchy. The hierarchy is dynamic, because of the possibility of conditional imports and of hidden imports caused by functions imported from other modules.

A module can import other modules for several reasons.

```
module:     module A1
            body
                val f a =
                    let import B1
                    in .. end
            end;

module:     module A2
            body
                export
                    val f g
                from
                    import B2;
                    val f a = ..
                    and g b = ..
                end
            end;

module:     module A3
            body
                import B3;
                val f a = ..
                and g b = ..
            end;
```

The first example above shows a module B1 locally imported for the private use of a function in A1. The second example shows a module B2 imported for private use in a module A2; the export construct guarantees that B2 is not exported from A2. The third example shows a module B3 which is imported by A3, and also reexported.

Sometimes an imported module B *must* be reexported from an importing module A. This happens when a type identifier t defined in B is contained in the (types of the) bindings exported by A.

Exporting those binders without exporting B may potentially generate objects of an unknown (in the current scope) type t, on which no operations are available. Hence the following restriction applies: whenever a type identifier is involved in the exports of a module, its type definition must also be exported.

There is an alternative notation for modules which are imported and reexported:

```
module:     module A
            includes B C
            body
                ..
            end;

module:     module A
            body
                import B C;
                ..
            end;
```

The two forms above are equivalent; the first one is just an abbreviation for the second one. The includes keyword should be understood as the set-theoretical inclusion of the bindings of B and C; not as the inclusion of the source lines constituting the definition of B and C.

Every module automatically imports a standard library module called /usr/lib/ml/lib, which contains all the predefined ML types, functions and values (remember that a module cannot access any outside binding, except the ones explicitly imported; this also applies to the predefined ML identifiers). The library module is shown as a module /usr/lib/ml/lib binding, on import:

```
module:     module A
            body
                val a = 3
            end;

declaration:  import A;

bindings:       module /usr/lib/ml/lib
                val a = 3 : int
```

The module /usr/lib/ml/lib binding is only an abbreviation for all the bindings defined in /usr/lib/ml/lib.ml: they are really imported and (re-)defined at this point.

Similarly, when importing a module B which exports a module A, the bindings of A are not explicitly listed. Instead a module A binding is presented as an abbreviation.

```
module:     module A
            body
                val a = 3
            end;

module:     module B
            includes A
            body
                val b = 5
            end;

declaration:  import B;
```



```
bindings:       module /usr/lib/ml/lib
                module A
                val b = 5 : int
```

In what follows, when we want to make those hidden bindings explicit, we write them indented under the respective module bindings:

```
declaration:  import B;

bindings:       module /usr/lib/ml/lib
                module A
                    val a = 3 : int
                val b = 5 : int
```

7.2. Module sharing

In an interactive session, a compiled module can be imported several times, but it is only loaded *once*, the first time it is imported. All the following imports of that module simply fetch the already loaded module. For example, two consecutive top level import A; declarations are semantically equivalent to only one of them, and no extra work is actually done in the second import.

At any moment there is at most one copy of a module in the system, and that copy is shared among all the modules which import it. Sharing means that types are shared, and that values are shared; for example:

```
module:     module A
            body
                abstype T = T of int
                with  val absT n = T n
                    and repT (T n) = n
                end;
                val a = ref 3
            end;

module:     module B
            includes A
            body
                val BabsT,BrepT = absT,repT
                and b = a
            end;

module:     module C
            includes A
            body
                val CabsT,CrepT = absT,repT
                and c = a
            end;

module:     module D
            includes B C
            body
            end;

declaration:  import D;
```

```
bindings:        module /usr/lib/ml/lib
                 module B
                     module /usr/lib/ml/lib
                     module A
                         module /usr/lib/ml/lib
                         type T
                         val absT : T -> int
                         val repT : int -> T
                         val a = (ref 3) : int ref
                     type T
                     val BabsT : T -> int
                     val BrepT : int -> T
                     val b = (ref 3) : int ref
                 module C
                     module /usr/lib/ml/lib
                     module A
                         module /usr/lib/ml/lib
                         type T
                         val absT : T -> int
                         val repT : int -> T
                         val a = (ref 3) : int ref

                     type T
                     val CabsT : T -> int
                     val CrepT : int -> T
                     val c = (ref 3) : int ref
```

Note that A is imported by D through two different import paths.

Sharing of types means that the module A is not typechecked twice: if it were, than we would have two *different* abstract types called T and CrepT(BabsT 3), for example, would produce a type error.

Sharing of values means that the module A is not evaluated twice: hence b and c are the *same* reference variable, so that any side effect on b will be reflected on c, and vice versa.

Sharing of values also has the desirable effect that multiply imported functions (e.g. the library functions) are not replicated.

7.3. Module versions

The system automatically keeps track of module versions, to make sure, for example, that when a module is significantly modified, all the modules which depend on it are recompiled. The system however does not automatically recompile obsolete modules; it simply produces an error message and an exception. For example, when an obsolete module A is trying to import a new version of module B, we have:

```
declaration:     import A;

exception:           Module A must be compiled
                     Exception: link
```

This error message is generated in a variety of circumstances, but the effect is always the same: module A must be recompiled.

A new version of a module is generated whenever that module is recompiled and its external

interface (i.e. the types of the exported bindings) changes.

8. The ML system

8.1. Entering the system

The ML system is entered under Unix by typing ml as a shell command. A library of standard functions is then loaded. This library is installation dependent and is meant to contain the functions which are most used locally. After the system prompt, you can start typing expressions or definitions, or loading source files.

8.2. Loading source files

An ML source file looks exactly like a set of top-level phrases, terminated by semicolons. A file prog.ml containing ML source code can be loaded by the use top-level command:

```
use "prog.ml";
```

where the string surrounded by quotes can be any Unix file name or path. The file extension .ml is normally used for ML source files, but this is not compulsory.

Source files may again contain use commands to load other source files in a cascade. There is a Unix-dependent limit on the depth of recursive loading: the message Cannot open file: *filename* is printed when such limit is exceeded. The file *filename* will not be loaded, but the loading of the previously opened files will continue.

Typing *DEL* while loading a file will interrupt loading and bring you back to the top level. This might not happen immediately, because some parts of the compilation process are not interruptible, but it will have effect as soon as the current phrase has finished compiling.

8.3. Exiting the system

Type *Control-D* to exit the system. No program or data created during an ML session is preserved, except for compiled modules.

8.4. Error messages

The most common syntax and type errors are described in section "Errors". Here are some unfrequent messages which are generated in particular situations.

An 'unimplemented' error is given when a value or type identifier is specified in an export export list but is not defined in the corresponding from part. This is different from the normal 'undefined' error for undefined identifiers.

```
- export val a from val b = 3 end;
Unimplemented Identifier: a

- export type t from type u = int end;
Unimplemented Type Identifier: t
```

A type error occurs when two isomorphism types having the same name are defined, and are allowed to interact:

```
- let abstype A = A of int
      with val absA n = A n end;
      abstype A = A of int
      with val repA (A n) = n end
  in repA(absA 3) end;
Type Clash in:    (repA (absA 3))
```

```
Looking for :     A
I have found :    A
Clash between two different types having the same name: A
```

Equality has its own type errors:

```
- let val f a = a
  in f = f end;
Invalid type of args to "=" or "<>": 'a -> 'a
I cannot compare functions

- val f a = (a = a);
Invalid type of args to "=" or "<>": 'a
I cannot compare polymorphic objects

- let abstype A = int
      with  val absA n = A n
            and repA (A n) = n
  in absA 3 = absA 3 end;
Invalid type of args to "=" or "<>": A
I cannot compare abstract types
```

In modules, all variables must be either locally declared or imported from other modules: global variables cannot be used:

```
- val a = 3;
> val a = 3 : int

- module A
  body val b = a end;
Modules cannot have global variables.
Unbound identifier: a
```

Modules which have never been compiled, or which are obsolete, produce a link exception:

```
- import A;

Module A must be compiled
Exception: link
```

8.5. Monitoring the compiler

The top level command monitor; gives access to a menu of commands for monitoring the internal functions of the ML compiler. In most cases these commands produce some check prints of internal data structures. Some options, like printing the parse trees, can be useful in becoming familiar with the system.

```
- monitor;
"y" for Yes; <CR> for No.
ParseTree?
Types?
TypeVariables?
Match?
StackCode?
OptimizerOff?
AssemblerCode?
ObjectCode?
Environment?
StopBeforeExec?
CheckHeap?
MemoryAllocation?
Watch? hex:
Timing?

-
```


The menu stops at every line, waiting for a y followed by a *CarriageReturn* for yes, or a *CarriageReturn* for no. The menu can be exited at any point by *DEL*: the options chosen up to that point will have effect; the other ones are unchanged. To reset the options to the initial default state, reenter the menu and answer no to all questions. All options have effect until changed again.

- *ParseTree* prints the parse tree of every subsequent top level phrase. This can be useful to see how the precedence of operators works.
- *Types* prints the type of every subsequent top level phrase, before executing it. It is useful only in debugging the system, as the type is the same as the type of the result which is printed after evaluation.
- *TypeVariables* prints all the hidden attributes of type variables. The following is for typechecking wizards only. 'a[i] means that 'a has an occurrence level i (i.e. it is used in a fun nesting of depth i). Generic type variables are written 'a[I] (I for infinity). Weak type variables are written _'a[i], and non-generic type variables are written l'a[i].
- *Match* prints the intermediate structures generated during the compilation of patterns. Not for human consumption.
- *StackCode* for every subsequent top-level phrase, prints the intermediate Functional Abstract Machine code produced by the compiler [Cardelli 83], after peephole optimization.
- *OptimizerOff* switches off the peephole optimizer for the abstract machine code. The optimizer works mainly on multiple jumps, function return, partial application, and tail recursion.
- *AssemblerCode* for every subsequent top-level phrase, prints the VAX assembler instructions produced from the abstract machine code.
- *ObjectCode* for every subsequent top-level phrase, prints the final hexadecimal result of the compilation.
- *Environment* for every subsequent top-level phrase, prints all the the types currently defined with their definition, and all the variables currently defined with their value and type.
- *StopBeforeExec* for every subsequent top-level phrase, stops immediately before executing the result of compilation, asking for a confirmation to proceed.
- *CheckHeap* for every subsequent top-level phrase, makes a complete consistency check of the data in the heap, and reports problems.
- *MemoryAllocation* reports every Unix memory allocation or deallocation, except the ones caused by Pascal.
- *Watch* is used for internal system debugging.
- *Timing* at the end of every evaluation, gives the parsing time (Pars), typechecking time (Anal), translation to abstract machine code and optimization time (Comp), translation to VAX code time (Assm), total compilation time (Total) and run time (Run). All in milliseconds.

8.6. Garbage collection and storage allocation

Garbage collection is done by a two-spaces copying compacting garbage collector. The size of the spaces grows as needed; garbage collection may escape with string "collect" if Unix refuses to grant more space when needed. For technical reasons, the allocation of very large data structures may escape with string "alloc", even if there is still memory available.

The frequency of garbage collection depends on the amount of active data and follows a simple adaptive algorithm; when there is little active data garbage collection is frequent, when there is much active data garbage collection is less frequent.

Garbage collection is not interruptible: pressing the *DEL* key during collection has no effect until the end of collection.

Data structures are kept in different pages according to their format. Pages are linked into three lists: the 'other space' list, the 'active' list and the 'free' list. There are always as many pages in the 'other space' list as in the union of the 'active' and 'free' lists.

Appendix A. Lexical classes

Ascii characters are classified into the following categories:

- Illegal — Unacceptable in a source program. These characters are ignored, and a warning message is printed.
- Eof — End-of-file character. It terminates an interactive session, or stops the process of reading a source ml file. A top level control-D is interpreted as Eof, thereby exiting the system.
- Blank — Characters which are interpreted as a space character " ".
- Digit — Digits.
- Letter — Letters and other characters which can be used to form identifiers.
- Symbol — A class of character which can be used to form operators (see appendix "Syntax of lexical entities").
- Escape — Escape character in strings. It behaves as a Symbol outside strings.
- Quote — String quotation character.
- Delim — One-character punctuation marks and parentheses.

Moreover, any sequence of legal character enclosed between { and } or end of file is a *comment* (except when { and } appear in a string). Comments can be nested. Each comment is considered equivalent to a single space character. An isolated } is treated as a Delim and may produce various kinds of syntax errors.

```
nul Eof;      soh Illegal;  stx Illegal;
etx Illegal;  eot Illegal;  enq Illegal;
ack Illegal;  bel Illegal;  bs  Illegal;
ht  Blank;    lf  Blank;    vt  Blank;
ff  Blank;    cr  Blank;    so  Illegal;
si  Illegal;  dle Illegal;  dc1 Illegal;
dc2 Illegal;  dc3 Illegal;  dc4 Illegal;
nak Illegal;  syn Illegal;  etb Illegal;
can Illegal;  em  Illegal;  sub Illegal;
esc Illegal;  fs  Illegal;  gs  Illegal;
rs  Illegal;  us  Illegal;  ' ' Blank;
'!' Symbol;   '"' Quote;    '#' Symbol;
'$' Symbol;   '%' Symbol;   '&' Symbol;
''' Letter;   '(' Delim;    ')' Delim;
'*' Symbol;   '+' Symbol;   ',' Delim;
'-' Symbol;   '.' Delim;    '/' Symbol;
'0' Digit;    '1' Digit;    '2' Digit;
'3' Digit;    '4' Digit;    '5' Digit;
'6' Digit;    '7' Digit;    '8' Digit;
'9' Digit;    ':' Symbol;   ';' Delim;
'<' Symbol;   '=' Symbol;   '>' Symbol;
'?' Symbol;   '@' Symbol;   'A' Letter;
'B' Letter;   'C' Letter;   'D' Letter;
'E' Letter;   'F' Letter;   'G' Letter;
'H' Letter;   'I' Letter;   'J' Letter;
'K' Letter;   'L' Letter;   'M' Letter;
'N' Letter;   'O' Letter;   'P' Letter;
'Q' Letter;   'R' Letter;   'S' Letter;
'T' Letter;   'U' Letter;   'V' Letter;
'W' Letter;   'X' Letter;   'Y' Letter;
'Z' Letter;   '[' Delim;    '\' Escape;
```

']'	Delim;	'^'	Symbol;	'_'	Letter;
'~'	Symbol;	'a'	Letter;	'b'	Letter;
'c'	Letter;	'd'	Letter;	'e'	Letter;
'f'	Letter;	'g'	Letter;	'h'	Letter;
'i'	Letter;	'j'	Letter;	'k'	Letter;
'l'	Letter;	'm'	Letter;	'n'	Letter;
'o'	Letter;	'p'	Letter;	'q'	Letter;
'r'	Letter;	's'	Letter;	't'	Letter;
'u'	Letter;	'v'	Letter;	'w'	Letter;
'x'	Letter;	'y'	Letter;	'z'	Letter;
'{'	Delim;	'|'	Symbol;	'}'	Delim;
'~'	Symbol;	DEL	Illegal;		

Appendix B. Keywords

abstype
and
body
case
do
else
end
export
escape
from
fun
if
import
in
includes
infix
let
local
module
nonfix
of
op
rec
then
type
use
val
where
while
with
:
?
??
?\
=

Appendix C. Predefined identifiers

true	Logic true	Nonfix constant
false	Logic false	Nonfix constant
not	Logic not	Nonfix
&	Logic and	Infix
or	Logic or	Infix
~	Complement	Nonfix
+	Plus	Infix
-	Difference	Infix
*	Times	Infix
div	Divide	Infix
mod	Modulo	Infix
=	Equal	Infix
<>	Different	Infix
>	Greater than	Infix
<	Less than	Infix
>=	Greater-eq	Infix
<=	Less-eq	Infix
size	String length	Nonfix
extract	Substring extraction	Nonfix
explode	String explosion	Nonfix
implode	String implosion	Nonfix
explodeascii	String to Ascii conv.	Nonfix
implodeascii	Ascii to string conv.	Nonfix
^	String concat.	Infix
intofstring	String to int conv.	Nonfix
stringofint	Int to string conv.	Nonfix
nil	Empty list	Nonfix constant
::	List cons	Infix constructor
hd	List head	Nonfix
tl	List tail	Nonfix
null	List null	Nonfix
length	List length	Nonfix
@	List append	Infix
map	List map	Nonfix
rev	List reverse	Nonfix
fold	List folding	Nonfix
revfold	List rev folding	Nonfix
ref	New reference	Nonfix constructor
!	Dereferencing	Nonfix
:=	Assignment	Infix
array	New array	Nonfix
arrayoflist	List to array	Nonfix
lowerbound	Array lower bound	Nonfix
arraysize	Array size	Nonfix
sub	Array indexing	Infix
update	Array update	Infix
arraytolist	Array to list	Nonfix
o	Function comp	Infix
file	Stream from file	Nonfix
save	Stream to file	Nonfix
stream	Empty stream	Nonfix
channel	External channel	Nonfix

input	Read from stream	Nonfix
output	Write to stream	Nonfix
lookahead	Peek from stream	Nonfix
caninput	Stream status	Nonfix
terminal	Terminal stream	Nonfix

Appendix D. Predefined type identifiers

unit	Unit Type	Nonfix
bool	Boolean Type	Nonfix
int	Integer Type	Nonfix
string	String Type	Nonfix
*	Cartesian Product	N-ary infix
->	Function Space	Infix
list	List Type	Suffix
ref	Reference Type	Suffix
array	Array Type	Suffix

Appendix E. Precedence of operators

Infix operators in order of decreasing precedence (see section "Lexical declarations").

{application}	*left associative*
sub	*left associative*
* div mod	*left associative*
+ - ^	*left associative*
:: @	*right associative*
= <> > < >= <=	*right associative*
&	*right associative*
or o	*right associative*
{user-infix}	*left associative*
update :=	*right associative*
.. , .. , ..	*n-ary infix*

Appendix F. Metasyntax

Strings between quotes '"' are terminals.

Identifiers are non-terminals.

Juxtaposition is syntactic concatenation.

'|' is syntactic alternative.

'[ ]' is the empty string.

'[..]' is zero or one times (i.e. optionally) ' .. '.

'{ .. }n' is n or more times ' .. ' (default n=0).

'{ .. / -- }n' means n (default 0) or more times '..' separated by '--'.

Parentheses '(..)' are used for precedence.

Appendix G. Syntax of lexical entities

```
Letter ::=
  "a" | .. | "z" | "A" | .. | "Z" |  "_".

Digit ::=
  "0" | .. | "9".

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

Character ::= .. (see appendix "Lexical classes" for a list of legal characters)

Ide ::=
  Letter {Letter | Digit | "'"} |
  {Symbol}1.

Integer ::=
  {Digit}1.

String ::=
  """ {Character} """.

Con ::=
  "(" ")" |
  Integer |
  String |
  Ide.

TypeIde ::=
  Ide.

TypeVar ::=
  "'" Ide.
```

Appendix H. Syntax

Syntactic alternatives are in order of decreasing precedence.

```
Phrase ::=
  [Exp | SimpleDecl | Module | Use | Monitor] ";".

Module ::=
  "module" ModuleName ["includes" {ModuleName}1] "body" Decl "end".

Use ::=
  "use" String.

Monitor ::=
  "monitor".

SimpleExp ::=
  ["op"] (Ide | Con) |
  "[" {Exp / ";"} "]" |
  "(" {Exp / ";"}1 ")".

Exp ::=
  SimpleExp |
  Exp  SimpleExp |
  Exp ":" Type |
  Exp Ide Exp |
  {Exp / ","}2 |
  "escape" Exp |
  "if" Exp "then" Exp "else" Exp |
  "while" Exp "do" Exp |
  "let" Decl "in" Exp "end" |
  Exp "where" Decl "end" |
  "case" Exp "of" Match |
  Exp "?" Exp |
  Exp "??" Exp Exp |
  Exp "?\" Ide "." Exp.
  "fun" Match.

SimpleDecl ::=
  "val" ValBind |
  "type" TypeBind |
  "abstype" TypeBind "with" Decl "end" |
  "export" ExportList "from" Decl "end" |
  "import" {ModuleName}1 |
  "infix" {Ide}1 | "nonfix {Ide}1.

Decl ::=
  SimpleDecl |
  Decl ";" Decl |
  "(" Decl ")".

ExportList ::=
  {"type" {Ide / ","}1 | "val" {Ide / ","}1}.

ValBind ::=
```



```
  Pat "=" Exp |
  ["op"] Ide {Pat}1 [":" Type] "=" Exp |
  {["op"] Ide Pat [":" Type] "=" Exp / "|"}2 |
  ValBind "and" ValBind |
  "rec" ValBind.

TypeBind ::=
  [TypeParams] TypeIde "=" {Ide ["of" Type] / "|"}1 |
  TypeBind "and" TypeBind |
  "rec" TypeBind.

TypeParams ::=
  TypeVar | "(" {TypeVar / ","}1 ")".

SimplePat ::=
  "_" |
  ["op"] Ide |
  Con |
  "[" {Pat / ";"} "]" |
  "(" Pat ")".

Pat ::=
  SimplePat |
  Con SimplePat |
  Pat ":" Type.
  SimplePat Con SimplePat |
  {Pat / ","}2 |

Match ::=
  {Pat "." Exp / "|"}1

Type ::=
  TypeVar |
  [TypeArgs] TypeIde |
  {Type / "*"}2 |
  Type "->" Type |
  "(" Type ")".

TypeArgs ::=
  Type |
  "(" {Type / ","}1 ")".

ModuleName ::= Ide | String.
```

Notes:

- The top-level command monitor provides a menu of check-prints for monitoring the inner workings of the compiler.
- The top-level command use *filename* loads a file of ML definitions and expressions.
- Recursive definitions can only contain bindings which are syntactically function bindings.

Appendix I. Escape sequences for strings

The escape character for strings is \; it introduces characters according to the following code:

\1 .. \9	One to nine spaces
\0	Ten spaces
\R	Carriage return
\L	Line feed
\T	Horizontal tabulation
\B	Backspace
\E	Escape
\N	Null (Ascii 0)
\D	Del (Ascii 127)
\^c	Ascii control char ^c (c any appropriate char)
\c	c (for any other char c different from #)
\#s	128 + s (for any of the previous sequences s)

Strings are printed at the top level in the same form in which they are read; that is surrounded by quotes, with all the \ and with no non-printable characters. Output operations instead print strings in their crude form.

Appendix J. System installation

The ML system usually comes on a tape containing a single directory mlkit (and sometimes a backup copy of it mlkit.BACKUP). The tape can be read by a tar -x mlkit command, which creates an mlkit subdirectory in the current directory; mlkit can be kept in any user or system directory.

The mlkit directory contains several files and subdirectories. README repeats the information contained in this section. VERSION contains the system version. doc contains this manual in troff format, a paper on the ML abstract machine, and a list of known ML sites. src contains the source programs. progs contains some example ml programs. pub contains the ML object code, libraries and shell scripts.

To install the system enter the pub subdirectory, and read the install script to make sure that its execution is not going to cause any damage on your system. To run install you probably need write permission on /usr/lib and /usr/bin. When you are sure that everything is ok, type install.

Apart from install, pub contains an usr-bin-ml script and an usr-lib-ml subdirectory. The install procedure simply moves usr-bin-ml to /usr/bin/ml, and usr-lib-ml to /usr/lib/ml. usr-lib-ml contains the real ML executable code, called mlsys, a file which is loaded every time the system is entered, called boot, a precompiled library module, called lib.sp and lib.im, and the library module source, called lib.ml. The library lib.ml should be considered as part of the ML system, and it is loaded (by boot) every time the system is entered; it can be extended by the users, but not reduced. usr-lib-ml also contains other user-defined ML library modules, which have to be explicitely imported when needed in user programs, and a file genlib.ml which recompiles all the libraries.

After install, the ML system can be run from any directory by typing ml. If you instead wish to call the system FORTRAN, say, just rename /usr/bin/ml to /usr/bin/FORTRAN.

If for some reason you need to recompile the system, you should proceed as follows. Most of the work is done by the Makefile file supplied with the source programs (say: make in mlkit/src). This generates an mlsys executable file which should be stripped (say: strip mlsys) and moved to /usr/lib/ml/mlsys.

The new ML system is now available, but the ML library has to be recompiled. When loading the new system for the first time, an error message 'Module /usr/lib/ml/lib must be compiled' will appear. Immediately after, type 'use "/usr/lib/ml/genlib.ml";': this will generate the new libraries for all the future uses of the new system. Now exit the system (don't try to work: the library has not been loaded yet). From now on, the library will be loaded automatically when entering the system.

The library module lib.ml can be changed at will, as long as the basic ML primitives (like @) defined there are preserved. It contains some basic ML functions, and can be extended with the locally most used utilities. After editing the file /usr/lib/ml/lib.ml, enter ML and type 'use "usr/lib/ml/lib.ml";' to compile and install the new library. Exit the system and reenter it, to actually load the new library.

Appendix K. Introducing new primitive operators

This appendix describes how to add to the ML system a new function "foo: t" that cannot be otherwise defined in ML. It has to be defined as a new primitive operation in the Functional Abstract Machine [Cardelli 83] and implemented in C, Pascal, Assembler, or any other language which respects VAX procedure call standards.

This procedure is complex, dangerous and not advisable. Before attempting it, one should consider implementing "foo" as an external program activated through the channel primitive as a subprocess.

- File mlglob.h
  - Add "OpFoo" to enumeration type "SMOperationT" IN ASCII ALPHABETIC ORDER!.
  - Add "OpFoo: ();" to the case list of type "SMCodeT".
  - Add "AtomFoo: AtomRefT" in section "{Parser Vars}".
  - If foo is prefix (infix, etc.) add "SynOpFoo" to the enumeration type "SynPrefixOpClassT" ("SynInfixOpClassT", etc.).
- File mlscan.p
  - Add "AtomFoo := TableInsertAtom(3,'foo');" in procedure "SetupTable" (where "3" is the length of "foo").
  - Add "PushSynRoleIde(AtomFoo);" in the same procedure (assuming that foo is nonfix, else use PushSynRoleInfix, etc. as appropriate, with second argument "SynOpFoo").
- File mlanal.p
  - Add "Predefined(AtomFoo,OpFoo,n,t);" to the procedure "SetupEnvironment", where foo: t (you can understand how to express t from the code of SetupEnvironment), and n is the arity of foo (i.e. the number of arguments it expects on the stack).
- File mldebg.p
  - Add "SetupMon(EmitSimpleOp(OpFoo));" to the procedure "SetupEnvValues" (if foo is not monadic, use SetupBin etc. Monadic, diadic, etc means that foo takes 1,2, etc. arguments on the top of the stack and returns a result there, after popping the arguments). It is essential that the order of setups in this procedure matches the order of definitions in SetupEnvironment (in mlanal.p).
- File mlconv.p
  - Add "OpFoo: Operand:=0{nil};" to the case list in function "ConvertOperand".
- File amglob.h
  - Add "#define OpFoo n" IN ASCII ALPHABETIC ORDER in section "Fam OpCodes"; make sure that all the opcodes are SEQUENTIALLY NUMBERED.
  - Add "extern Address DoFoo();" in section "Externals".
  - If foo may escape with string "foo", declare "extern Address *StrFoo;" in section "FailStrings".
- File amevalop.c
  - Add "OpFoo: CallOp((Address)DoFoo,n); break;" to the case list in procedure "Assem", where n is the arity of Foo (i.e. how many arguments it expects on the stack).
- File ammall.c
  - If foo may escape with string "foo", declare "Address *StrFoo;" and add "StrFoo = PushGCBox(StringFromC("foo"));" to the procedure "AllocFailStrings".
- File amdebg.c
  - Add "case OpFoo: printf("Foo"); break;" to the case list of procedure "AMStatPrint".
- File amfooop.c

  - Create this file, containing the C function "Address DoFoo(argn, .. ,arg1)", implementing the desired behavior. The order of arguments must be the inverse of the order in which "foo(arg1, .. ,argn)" is called from ML. WARNING: do not attempt to allocate or change ML data structures in this function without deeply understanding how the garbage collector works. It is safe to inspect (read only) the ML data structures passed as parameters (see appendix "Vax data formats"), and to return them unchanged. The result of this function must match the ML type declared in mlanal.p. If DoFoo needs to produce an ML exception, it can do so by calling "DoFailwith(*StrFoo)"
- File Makefile
  - Add a line "FOOOP = amglob.h amfooop.c".
  - Add "amfooop.o" to the definition of "MLSYS".
  - Add a line "amfooop.o: $(FOOOP); CC amfooop.c".

Appendix L. System limitations

There is a limit on the size of any single ML object, due to storage allocation problems. This limit is determined by the constant "PageSize" in the file "amglob.h". All the limitations mentioned on this section are further constrained by this restriction.

The semantic limit on the length of strings is 64K chars.

The syntactic limit on the length of string quotations in a source program is also 64K chars.

Several memory areas inside the system have fixed size. When one of these is exceeded, compilation fails and a message is printed. The only way to fix this is to search for the point in the source program where the error message is generated, increase the corresponding area (usually by redefining a constant) and recompile the system.

Appendix M. VAX data formats

Each segment "+---+" in the pictures is one byte. The symbol "^" below a data structure represents the location pointed to by pointers to that structure; fields preceding "^" are only used during garbage collection and are inaccessible to the ML operations (and hence to the user). Unboxed data is kept on the stack, or in the place of pointers in other data structures; unboxed data does not require storage allocation. Pointers can be distinguished from unboxed data as the former are > 64K. There are automatic conversions between SmallIntegers and BigIntegers, so that the ML operations only see the type int.

Unit

0	(unity)	(unboxed)

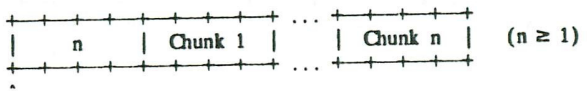
Boolean

0	(false)	(unboxed)
1	(true)	(unboxed)

SmallInteger

-32768 .. +32767	(unboxed)

BigInteger

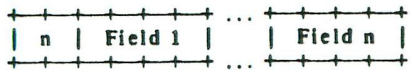


Small Record

Medium Record

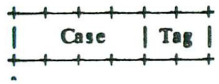
Record

0 (nullrecord) (unboxed)



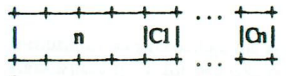
(n > 0)

Variant



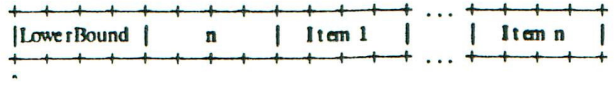
Reference

String



(n ≥ 0)

Array

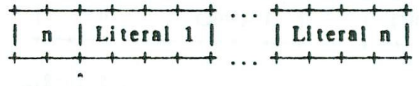


(n ≥ 0)

Text

(n ≥ 1)
(*) when pointed from a Closure.
Literals may be Nil or point to Literals

Literals



(n ≥ 1)
Literal may be a Text, a String or a BigInteger

Closure

(n ≥ 0)
Text points to a Text

Appendix N. Ascii codes

0 nul	1 soh	2 stx	3 etx	4 eot	5 enq	6 ack	7 bel
8 bs	9 ht	10 nl	11 vt	12 np	13 cr	14 so	15 si
16 dle	17 dc1	18 dc2	19 dc3	20 dc4	21 nak	22 syn	23 etb
24 can	25 em	26 sub	27 esc	28 fs	29 gs	30 rs	31 us
32 sp	33 !	34 "	35 #	36 $	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 del

Appendix O. Transient

This appendix contains a list of restrictions and limitations due to the accidents of this particular implementation. They are not consequences of rational thougth or planning but rather incomplete implementations of features, known bugs to be fixed or problems which are harder to fix than they are worth.

• Several minor problems still afflict modules. These will be fixed gradually in the short and medium term. Meanwhile the following things should be noted.

With the exception of the library functions in /usr/lib/ml/lib.ml, modules must not use predefined functions as argumentless functional objects, e.g. + in fold (op +) [1;2;3;4] 0. At the moment this will produce and 'Unbound identifier' error. However, it is possible to replace (op +) by (fun(x,y). x+y).

Under some complicated circumstances the sharing of types will fail. The scenario is: module A privately imports module B, and then exports a type T defined in B by an export export list; then module C imports A and B and makes T-from-A interact with T-from-B. This will very likely never happen to you. However, if you get an unexplained type conflict (probably two different abstract types having the same name), this might be it. You should be able to fix this by making sure that that type is exported by includes declarations, as opposed to export export lists.

If a module A is recompiled, all the modules depending on it must be recompiled, even if the interface of A has not changed.

Sometimes, when a module B which imports A has to be recompiled, instead of the message Module B must be compiled, the less precise message Module importing A must be compiled is produced.

There is no check on the restriction that modules exporting a type identifier (maybe as part of the type of an exported binder), must also export the type definition. However, nothing bad can come of this.

On some occasions, precompiled modules can give undefined type errors when importing them, or when compiling modules which import them. This can happen if an export declaration rearranges the order of types and values in the body of a module. To fix this, put the export lists back in an appropriate dependency order. You can look at the .sp file for that module to see what is happening. Nothing bad can come of this.

Consider the following scenario: module A is defined; module B, which imports A, is defined; module B is imported; module A is redefined. At this point module B is obsolete, and one would expect a version error in importing B again. This would happen if module B had not already been imported; however an import B at this point will share the already imported module B, without any complaint. At this point we have a B including the old A, while we might expect the new A to be in action. To get the new A into play, you must recompile B and reimport it. Nothing bad can come of this, except confusion.

• Input/Output. The current plans are to implement an efficient buffering with in-core cache for internal streams, and to use unbuffered 'raw' I/O on external streams. Internal streams would still look unbuffered to the user, but all the disk I/O would be done in chunks. The current situation is fluid.

At the moment a limited number of streams can be open in an ML session and they cannot be closed. When this limit is exceeded, the operations Stream and File fail. This problem will be solved in a future version by an automatic cache of open streams.

The operations implemented so far are file, save, stream, caninput, input, output and terminal (not yet working: lookahead and channel).

On end of stream, inchar waits forever; type *DEL* to exit input waits and infinite loops (a trappable exception "interrupt" is produced) (ignore the longjmp botch and System Crash! messages which are sometimes generated).

• Bignums are not yet implemented. Integers currently range between ~32768 and 32767.

• The exception "corruption" is produced when something goes very wrong (e.g. after a very deep recursion which overruns the stack); the system is currently unsafe after a corruption. You should rarely be thrown out of the system, but on extreme corruptions you may get a Run out of memory message and a core dump.

• Deep recursions producing stack overflow will badly corrupt the system; (this is the most likely cause of crash). The ml exception "corruption", or the message System Crash! may be generated. If you come across this problem, try to rewrite your programs in a linear recursive style, so that the compiler can optimize them to iterations, or directly in iterative style using while-do. This problem cannot be solved easily, given the current Unix memory management primitives.

• The copying garbage collection algorithm is recursive. Because of this, the system can crash during garbage collection when (1) very deep (thousands of levels) lists or other structures are present in memory and (2) not enough virtual memory can be obtained for the system stack, e.g. because of Unix limits on the size of page tables, or for lack of swap space on disk.

• A failure "alloc" may be generated when compiling big programs or big modules. The absolute size of the program is not directly relevant: the important factors are the size of every individual fun-expression and the number of definitions in a module. The preferred programming style should consist in writing a large number of small modules (1-5 pages) containing small (10-100 line) functions. The size of type definitions has no relevance to these problems.

Appendix P. Differences from Standard ML

This appendix contains a list of the differences between this implementation and the official definition of the ML language. All the differences are either temporary (due to the conversion process from earlier implementations), or compatible extensions of the standard (some of which are intended to be transferred to the standard in the future), or bugs. This manual is supposed to reflect exactly the current state of the implementation. It will be upgraded to the standard together with the evolution of the implementation.

• The failure mechanism is not the standard one, and should be converted to the standard in the near future.

• The I/O system is not part of the standard (Standard ML does not define any I/O system). It is however a partial implementation of a proposal for I/O in Standard ML. Also, see Appendix "Transient".

• The module system is not part of the standard (Standard ML does not define any separate compilation mechanism). A module system is being studied to be embedded in Standard ML, it includes parametric modules and persistent data. The current implementation bears resemblance to a special case of that proposal. The "export" declaration construct is part the module system, and not part of Standard ML. Also, see Appendix "Transient".

• The "local" declaration construct is not yet implemented.

• The typechecking of references is more permissive than the one defined for Standard ML. Programs using polymorphic references in this systems might not pass typechecking in Standard ML. This strict extension of the standard will probably be preserved.

• The tests for the correct typechecking of equality are not yet implemented. As it stands, the system is more permissive than it should be, but not wrong.

• The tests for correct scoping of type identifiers are not yet fully implemented. As it stands, the system is more permissive than it should be, but not wrong.

• The scoping of constructors is not yet the one required by the standard. As it stands, the system is more permissive than it should be, but not wrong.

• The evaluation order of function applications is wrong.

• The check against multiple occurrences of the same identifier in a pattern has not yet been implemented. The result of doing so is undefined.

• The typechecking of the "case" statement is more restrictive than it should be, but not wrong.

• Integers and strings are not yet allowed in patterns.

• Integers are only 16-bit integers.

• Curried clausal definitions are not yet supported.

• Pattern matching should issue a series of warning messages in some situations: this is not yet implemented.

• The "<=>" form of abstract types is not yet implemented.

• Arrays are not part of Standard ML. The arrays provided by this implementation are intended as a predefined abstract data type, whose definition is contained in section "Arrays".

Escape sequences for strings are extended by the sequence \#s, described in Appendix "Escape sequences for strings". This is not part of the standard, but is necessary for writing terminal drivers.

References

[1] L.Cardelli. "The Functional Abstract Machine", Bell Labs Technical Memorandum TM-83-11271-1, Bell Labs Technical Report TR-107, 1983.

This paper describes the abstract machine on which the ML implementation is based. It contains a formal definition of the machine, which should guide implementations on different kinds of hardware.

[2] M.Gordon, R.Milner, C.Wadsworth. "Edinburgh LCF", Lecture Notes in Computer Science, n.78, Springer-Verlag, 1979.

This manual contains the original definition of the ML language. The present ML language is quite different in syntactic details and type declarations, but the basic semantic principles are still the same.

[3] R.Milner. "A theory of type polymorphism in programming", Journal of Computer and System Science 17, 348-375, 1978.

This paper estabilishes the semantic foundations of the ML type system, and contains a correctness proof for a simplified ML typechecker.

ML Syntax

```
Phrase ::=
  [Exp | SimpleDecl | Module |
   "use" String | "monitor"] ";".

Module ::=
  "module" ModuleName
  ["includes" {ModuleName}1]
  "body" Decl "end".

SimpleExp ::=
  ["op"] (Ide | Con) |
  "[" {Exp / ";"} "]" |
  "(" {Exp / ";"}1 ")".

Exp ::=
  SimpleExp |
  Exp SimpleExp |
  Exp ":" Type |
  Exp Ide Exp |
  {Exp / ","}2 |
  "escape" Exp |
  "if" Exp "then" Exp "else" Exp |
  "while" Exp "do" Exp |
  "let" Decl "in" Exp "end" |
  Exp "where" Decl "end" |
  "case" Exp "of" Match |
  Exp "?" Exp |
  Exp "??" Exp Exp |
  Exp "?\" Ide "." Exp.
  "fun" Match.

SimpleDecl ::=
  "val" ValBind |
  "type" TypeBind |
  "abstype" TypeBind "with" Decl "end" |
  "export" ExportList "from" Decl "end" |
  "import" {ModuleName}1 |
  "infix" {Ide}1 | "nonfix" {Ide}1.

Decl ::=
  SimpleDecl |
  Decl ";" Decl |
  "(" Decl ")".

ExportList ::=
  {("abstype" | "type" | "val") {Ide / ","}1}1.

ValBind ::=
  Pat "=" Exp |
  ["op"] Ide {SimplePat}1 [":" Type] "=" Exp |
  {["op"] Ide SimplePat [":" Type] "=" Exp / "|"}2 |
  ValBind "and" ValBind |
  "rec" ValBind.

TypeBind ::=
  [Params] TypeIde "=" {Ide ["of" Type] / "|"}1 |
  TypeBind "and" TypeBind |
  "rec" TypeBind.

Params ::=
  TypeVar | "("{TypeVar / ","}1 ")".
```

Syntactic alternatives and infix operators are listed in order of decreasing precedence.

```
SimplePat ::=
  "_" |
  ["op"] Ide |
  Con |
  "[" {Pat / ";"} "]" |
  "(" Pat ")".

Pat ::=
  SimplePat |
  Con SimplePat |
  Pat ":" Type |
  SimplePat Con SimplePat |
  {Pat / ","}2 |

Match ::= {Pat "." Exp / "|"}1

Type ::=
  TypeVar |
  [TypeArgs] TypeIde |
  {Type / "*"}2 |
  Type "->" Type |
  "(" Type ")".

TypeArgs ::= Type | "(" {Type / ","}1 ")".

Letter ::= "a" | .. | "z" | "A" | .. | "Z" | "_".
Digit ::= "0" | .. | "9".
Symbol ::= !#%&$+-/:<=>?@\~`|*.
Character ::= ht | .. | cr | " " | .. | "~".
Ide ::= Letter {Letter | Digit | "'"} | {Symbol}1.
Integer ::= {Digit}1.
String ::= """ {Character} """.
Con ::= "(" ")" | Integer | String | Ide.
TypeIde ::= Ide.
TypeVar ::= "'" Ide.
ModuleName ::= Ide | String.
```

Keywords

abstype and body case do else end export escape from fun import if in includes infix let local module nonfix of op rec then type use val where while with : ? ?? ?\ =

Operations

{application}L {sub}L {* div mod}L {+ - ^}L {:: @}R {= <> > < >= <=}R {&}R {or o}R {user-infix}L {update :=}R true false not ~ size extract explode implode explodeascii implodeascii intofstring stringofint nil hd tl null length map rev fold revfold ref ! array arrayoflist lowerbound arraysize sub update arraytolist file save stream channel input output lookahead caninput terminal user-nonfix

Constructors

integers strings tuples () true false nil :: ref user-defined

Metasyntax

Strings between quotes ' " ' are terminals.
Identifiers are non-terminals.
Juxtaposition is syntactic concatenation.
' | ' is syntactic alternative.
'[..]' is zero or one times (i.e. optionally) ' .. '.
'{ .. }n' is n (default 0) or more times ' .. '.
'{ .. / -- }n' is n or more times ' .. ' separated by ' -- '.
Parentheses '(..)' are used for precedence.

Modules for Standard ML (Draft)

David MacQueen

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

0. Preface

This proposal is a preliminary version that is being distributed to elicit comments from the ML/Hope community. It should be considered an addendum to Robin Milner's Standard ML proposal [MIL84].

1. Introduction

Writing large programs in ML, as in any language, poses difficult problems of organization. Most modern programming languages contain constructs designed to help a programmer organize large systems. These constructs, variously known as modules, packages, or clusters, support the partitioning of a system into reasonably sized components and provide secure methods for combining program components to form systems. These same constructs usually make it possible to build libraries of shareable, generic components and provide for separate compilation of these components.

Because of its semantic simplicity and regularity, ML is a particularly good base for the design of facilities for modularity. Because of ML's polymorphic type system, modules in ML can provide a greater degree of generality and flexibility than is possible in other typed languages, which are typically derived from Pascal. In the design presented here, I have tried to exploit the existing structure of ML as far as possible and where it is necessary to augment that structure I have attempted to do so in the spirit of the ML design. This proposal concentrates on the language design aspect; it does not address implementation techniques or relevant programming environment issues, topics that should be dealt with in future reports. There are a considerable number of examples meant to motivate as well as illustrate the design, but one of the difficulties of describing language constructs for "programming in the large" is that small scale examples may not be very realistic.

This proposal is based on the fruits of a long collaboration with Rod Burstall on prototype designs for modules in Hope [MAC81], and on theoretical investigations with Ravi Sethi and Gordon Plotkin that were motivated by those designs. The module designs for Hope were in turn heavily influenced by the Clear specification language of Burstall and Goguen [BUR77]. Many hours of discussion with Luca Cardelli helped to solidify the ideas, and a limited prototype of modules implemented by Luca in Pose 3 of his ML system [CAR84] made it possible to gain valuable experience programming with modules.

1.1. The problem: managing environments

In its simplest form, a *module* is just a named collection of declarations whose purpose is to define an environment. It follows that one approach to the design of a module facility is to start with the notions of declarations and environments and consider how they can be made into relatively self-sufficient entities. In other words, what is the most natural way to promote declarations and environments to a quasi-first-class status?

The evaluation of a declaration produces an environment. However, since a declaration often contains free identifiers of various kinds, its evaluation will require an environment that

binds these free identifiers. Thus the meaning of a declaration *dec* is, roughly, a function

$$[[dec]]: Env \rightarrow Env$$

Let us call the argument of this function the *environment of definition*, and the resulting environment the *defined environment*. For the usual case of a declaration embedded in an ML program, the environment of definition defaults to the statically prevailing environment of the declaration, and the defined environment is used to augment the prevailing environment to obtain a new prevailing environment for the scope of the declaration. The prevailing environment is guaranteed to be an appropriate environment of definition if the program as a whole (the declaration and its context) is well typed.

When we consider a declaration in isolation, however, we must explicitly constrain the environment of definition by specifying a *type signature* for it such that the declaration is well typed with respect to that signature. The signature of the environment of definition is not uniquely determined in general, but together with the declaration it determines, by type inference, a most general signature for the defined environment. Furthermore, a signature for the environment of definition is all that is required for compilation of the declaration.

It follows that a minimal form of module would be a declaration together with a signature specifying typing information for its free identifiers. If $M = <dec, sig>$ is such a pair, then it represents a function

$$[[M]]: sig \rightarrow sig'$$

where $sig' = TYPE(dec, sig)$ is the signature inferred for *dec* given the context specification *sig*, and a signature used as a type represents the collection of all environments satisfying that signature.

The next question is how to employ such a module; where should we obtain an appropriate input environment (the environment of definition for *dec*) and what should we do with the resulting defined environment? One possibility is to treat the module as a mere abbreviation for the declaration associated with it. "Instantiating" (i.e. using or applying) the module at a given point in a program could then be viewed as equivalent to textually inserting the declaration at that point, and would be valid if the prevailing environment at that point satisfied the signature component of the module (with respect to which the declaration had been compiled). As for an ordinary declaration, the prevailing environment would serve as the environment of definition, and the defined environment would be concatenated onto the prevailing environment.

This treatment of modules as essentially precompiled macros for declarations is sound, but it has unfortunate consequences for program structure. Suppose a module named *A* provides certain resources (e.g. defines certain types and operations) that are to be used in a "private" way in defining modules *B* and *C* (i.e. the facilities of *A* are not relevant in the context in which *B* and *C* are to be used). The program fragment

```
use "A";
use "B";
use "C";
```

(where `"A"` is the name of a file containing the compiled form of module *A* and `use` causes the file to be loaded and the declaration to be evaluated) is not satisfactory because the environments simply accumulate, meaning that the bindings defined in *A* are still accessible, even though they are only relevant to the instantiation of *B* and *C*. We could restrict the scope of the bindings from *A* by modifying the program as follows:

```
local use "A"
in use "B";
   use "C"
end;
```

but this leaves the problem that the instantiation of *C* does not see the bindings of *A* directly, but only as overlaid by *B*, so rebinding of identifiers in *B* may interfere with the access of *C* to *A*. If

we try to separate the instantiations of *B* and *C* as in

```
local use "A" in use "B" end;
. . .
local use "A" in use "C" end;
```

we have introduced a new problem because *A* has been instantiated twice, producing two different environments. If functions defined in *B* and *C* attempt to communicate via an abstract type defined in *A*, this communication will fail, because *B* and *C* now use two different versions of that abstract type that will not match. It is becoming clear that the macro model of module usage does not give us enough control of either the environment of definition or the use of the defined environment.

We could solve part of the problem by allowing the defined environment of a module instantiation to be named, so that the same environment could be used in several places. But complete control requires in addition that the environment of definition be explicitly specified in terms of named environments. Since only the signatures of these named environments are necessary for compilation, it is easy and natural to abstract with respect to them, obtaining a parameterized form of module. The explicit specification of prerequisite environments in terms of named module instances also is a powerful structuring discipline for program design.

1.2. Design principles

The design presented in the following section is based on the following principles and ideas.

1. We strictly separate the environment of definition from the environment of use. We require explicit specification of the complete environment of definition in terms of *antecedent* instances of specified signatures.
2. All modules are viewed as parametric, by abstraction with respect to their antecedent instances (even in the case where they have no antecedents). A module is a function which produces environments of a particular signature when applied to argument instances of specified signatures.
3. We provide for separate definition of interfaces (signatures) and their implementations (modules). This separation is essential for parametric modules.
4. The defined environment of a module must be closed. For example, no types may appear in its signature that are not defined directly or indirectly in the environment. This requirement implies that the defined environment must sometimes inherit certain antecedents.
5. We introduce environments of named signatures, modules, and instances. Such environments may be partially persistent, forming permanent systems or libraries.
6. We minimize the visibility of information by requiring explicit declaration of inheritance (information hiding).
7. Shared antecedents required for coherent interaction between module parameters must be explicitly specified.

2. Basic proposal

This section describes the three constructs making up the module proposal: signatures, modules, and instances. The syntax used here will be the most basic form; some abbreviations and enhancements that make common idioms more convenient will be described in Section 3.

The central concept in this proposal is that of an environment structure (called an *instance* here {Note Instance Terminology}) consisting of a set of bindings of types, values, and exceptions. An instance has two main functions: (1) it is a hybrid collection of entities associating types, values, and exceptions; and (2) it provides names for its constituent entities. The notion of a hybrid compound structure is the fundamental one; the naming of the components is merely a convenience.

The notion of an instance is actually somewhat more complex than just indicated, because of the problem of modeling inheritance. It will be very useful to build new environments in terms of existing ones, and in such cases the new environment will often depend overtly on its antecedents (typically types of values bound in the new environment will involve types inherited from the antecedents). To express these dependencies, we will allow environments to contain their required antecedent environments as components (i.e. as instance bindings). In a sense, each instance carries with it its own family tree, or at least as much of its family tree as is necessary to make use of the instance.

Instances, as environment structures, will have their own kind of type, called a signature. Basically, a signature gives appropriate type specifications for each of the bindings making up an instance (the arity of a type constructor, the type of a value or exception, and the signature of an antecedent instance). From another point of view, however, instances themselves can often be considered a generalized form of type, an "interpreted" type, wherein a type (or types) is combined with operations and exceptions with which to manipulate its values. We exploit the parallel between types and instances in defining parameterized modules, which partake of the natures of both polymorphic functions and polymorphic type constructors.

2.1. Basic forms: signatures, modules, and instances

As mentioned above, a signature is a type specification for an environment. The notation for anonymous signatures is

```
sig
   instance specs
   type specs
   val specs
   exception specs
end
```

The various kinds of specifications can be interspersed, as long as names are introduced before being used (except in the case of recursive concrete type specifications). The following signature specifies instances with a type `elem` and a binary predicate `eq` (presumably representing an equivalence or equality relation) over `elem`:

```
sig
   type elem
   val eq: elem * elem -> bool
end
```

(we will consider built-in types like `bool` to be pervasive, i.e. meaningful everywhere).

Of course, writing out all signatures in full soon becomes very cumbersome, so as usual we introduce a new kind of declaration for naming signatures. Thus we can declare

```
instance EQ = sig
    type elem
    val eq: s * s -> bool
end
```

As another simple example, here is a declaration of a signature for stacks as a unary type constructor with appropriate operations and exceptions.

```
signature STACK = sig
    type 'a stack
    val nilstack: 'a stack
    and push: 'a * 'a stack -> 'a stack
    and empty: 'a stack -> bool
    and pop: 'a stack -> 'a stack
    and top: 'a stack -> 'a
```

```
    exception pop: unit and top: unit
end
```

A *module* is a declaration supplied with an explicit context in the form of a set of instance parameters with specified signatures. The purpose of the declaration is to create a new environment structure (that is, an *instance* of the module) relative to the particular context provided by a set of actual instance parameters. The declaration part of the module is type checked relative to the signatures of the parameters, which must bind all the free identifiers occurring in the declaration, yielding an inferred signature that must agree (in a sense to be defined later) with any explicitly declared signature for the module result.

Here is the rather jejune example of a module for stacks that implements the signature given above.

```
module StackMod () : STACK
    type 'a stack = nilstack' | push' of 'a * 'a stack
    exception pop: unit and top: unit
    val nilstack = nilstack'
    and push = push'
    and empty(nilstack') = true |
        empty _ = false
    and pop(push'(_,s)) = s |
        pop _ = escape pop
    and top(push'(x,_)) = x |
        top _ = escape top
end
```

In this particular example the declaration making up the body of `StackMod` is entirely self-contained (i.e. there are no free identifiers), so the module has no parameters. Nevertheless, the module must still be applied (to a trivial, empty parameter set) to produce an instance:

```
instance Stack = StackMod ()     { Stack: STACK }
```

It is possible to instantiate such a module more than once, producing several instances, and the `stack` type constructors in each instance will be distinct, as will the types of the operators such as `push`. Multiple instances of a "pure" module such as Stack are sometimes useful, but usually a single instance will do, since all instances provide essentially the same resource and a single instance could be shared by all "clients" without interference.

However, when instances have state it is often appropriate to create more than one instance of a module even though that module has no parameters. For example, we could define a module implementing stacks of integers such that each separate instance of the module was a stack.

```
module StackMod' ()
    local
        val stack: int list ref = ref nil
          (local stack data structure)
    in
        val push x = (stack := x :: !stack)
        and pop () = (stack := tl(!stack))
        and top () = hd(!stack)
    end
end
```

Note that this version of stacks does not introduce a new stack type with each instance. Instead, the module itself (or more precisely its result signature) plays the role of a type, and is in fact quite similar to a class in Simula or Smalltalk, with its instances corresponding to objects of the class.

Another point illustrated by this example is that modules need not have their result signature specified. When it is not specified, type inference can be used to determine an anonymous signature for the module's instances. In this case the signature would be


```
sig
    val push: int -> unit
    and pop: unit -> unit
    and top: unit -> int
end
```

This version of the stack module can be used to create several instances, each of which serves as a separate stack object with its own internal stack data structure.

```
instance Stack1 = StackMod' ()

instance Stack2 = StackMod' ()
```

When defining a module that implements a signature like `STACK`, it is natural to declare the type component `stack` as a new type, so that each instantiation of the module creates a new, unique type constructor. In fact, as currently defined Standard ML has no other means of producing a type binding; both concrete and abstract type declarations are "generative", meaning that they create brand new types when evaluated. But consider the signature `EQ` of a type with an equality predicate. In this case it is probably not interesting to create instances with entirely new types. Rather the typical use of this signature and its instances is to impose structure on an existing type.

For example, we might like to define an instance of `EQ` wherein the type component is `int` and the `eq` predicate is ordinary integer equality. We can do this with the aid of a new form of "transparent" type declaration which simply binds a given type to a name (similar to the old type abbreviation declaration). Here is a possible syntax for such declarations:

```
type elem is int
```

(replacing the "=" symbol with "is") {Note Type Declaration Notation}. With this new form of declaration, we can use the following module declaration to produce the desired instance.

```
module IntEqMod () : EQ
    type elem is int
    val eq = (op =)
end

instance IntEq = IntEqMod ()
```

Another interesting example of the use of this form of declaration is given by the following module for producing lexicographic orderings on lists.

```
signature ORDSET = sig
    type s
    val le: s * s -> bool
end

module LexOrdMod(O: ORDSET): ORDSET
    type s is O.s list
    val le(nil,_) = true |
        le(_,nil) = false |
        le(x::l, y::m) = if O.le(x,y) / O.le(y,x) then le(l,m)
                         else if O.le(x,y) then true
                         else false
end
```

```
module IntOrdMod () : ORDSET
    type S is int
    val le = op <=
end

instance IntOrd = IntOrdMod()
{ IntOrd: ORDSET
  IntOrd.s = int
  IntOrd.le: int * int -> bool }

instance LexOrdInt = LexOrdMod(IntOrd)
{ LexOrdInt: ORDSET
  LexOrdInt.s = int list
  LexOrdInt.le: int list * int list -> bool }

instance LexOrdIntList = LexOrdMod(LexOrdInt)
{ LexOrdIntList: ORDSET
  LexOrdIntList.s = int list list
  LexOrdIntList.ls = int list list * int list list -> bool }
```

2.2. Inheritance

We can distinguish two classes of instance parameters. One class of parameter provides some utilities used internally to implement the desired environment. The parameter `L` in the following module definition is an example of this class.

```
signature LIST = sig
    val filter: ('a -> bool) * 'a list -> 'a list
    and forall: ('a -> bool) * 'a list -> bool
    and exists: ('a -> bool) * 'a list -> bool
    and choose: ('a -> bool) * 'a list -> 'a
    and map: ('a -> 'b) * 'a list -> 'b list
    and apply: ('a -> unit) * 'a list -> unit
    exception choose: unit
end

module ListMod () : LIST
    val filter = . . .
end

instance List = ListMod ()

module AssociationMod (L: LIST)
    type 'a Association = association of 'a list ref

    val EmptyAssociation(): 'a Association =  association(ref []);

    and LookupAssociation(TestItem: 'a -> bool, GenItem: unit -> 'a,
                          association Items: 'a Association): 'a =
            L.choose(TestItem, !Items)
            trap L.choose (Items := GenItem() :: !Items; hd(!Items))

    and MapAssociation(F: 'a -> 'b,
                       association(ref Items): 'a Association): 'b list =
            L.map(F, Items)
```



```
    and ApplyAssociation(F: 'a -> unit,
                         association(ref Items): 'a Association): unit =
            L.apply(F, Items)
end
```

This example also illustrates the basic method of referring to components of an instance by qualified identifiers such as `L.choose` (Note Qualified Identifier Notation). In Section 3 we will describe a declaration construct for "opening" instances to allow unqualified use of the names bound in them.

The result signature of `AssociationMod` is

```
sig
    type 'a Association = association of 'a list ref

    val EmptyAssociation : unit -> 'a Association
    and LookupAssociation : ('a -> bool) * (unit -> 'a)
                                * 'a Association -> 'a
    and MapAssociation : ('a -> 'b) * 'a Association -> 'b list
    and ApplyAssociation : ('a -> unit) * 'a Association -> unit
end
```

This signature is entirely independent of the signature `LIST`, and furthermore, the functions defined in `AssociationMod` will not raise the exception `choose` from the parameter instance L. Thus when we create an instance of `AssociationMod` by

```
instance Association = AssociationMod(List)
```

the user of `Association` need not be concerned with the fact that `List` was used in its creation.

The other class consists of those instance parameters that are relevant to the *use* of the derived instance as well as its definition. Consider the following example of a module that, given an instance of `EQ` (a type with and equality function), defines membership in and equality between lists of elements of that type.

```
signature LISTEQ = sig
    instance E: EQ
    val member: E.elem * E.elem list -> bool
    and eqlists: E.elem list * E.elem list -> bool
end

module ListEqMod (E': EQ) : LISTEQ
    instance E = E'
    val member(e,nil) = false |
        member(e,e'::l) = E.eq(e,e') orelse member(e,l)
    and eqlists(nil,nil) = true |
        eqlists(e1::l1,e2::l2) = E.eq(e1,e2) andalso eqlists(l1,l2)
end
```

In this case, the types of `member` and `eqlists` in `LISTEQ` are clearly dependent on the type `elem` inherited from the instance `E`, so without this instance as a component, the signature (and the corresponding instances) would not be self-contained. But there is a subtle issue here: why not inherit just the type `E.elem` from the instance `E` instead of the whole instance, since only the type is involved in the signature `LISTEQ`? The reason is that the membership and equality functions for lists over a given type are predicated on a particular equality function for the elements, and so proper use of the list functions may require knowledge of that equality function. Another kind of dependence involves exceptions, since functions in a derived instance may raise exceptions declared in a parameter instance, so the parameter instance would have to be inherited

if we wished to selectively handle those exceptions. In short, an instance parameter should be inherited whenever it is required as context for the proper use or interpretation of the derived instance.

The instance declaration

```
instance E = E'
```

in `ListEqMod` is necessary to cause the inheritance of the parameter `E'` as a component of the result of the module. This is a rather cumbersome form, however (especially if we had used `E` instead of `E'` as the formal parameter name), and we will provide a derived form for conveniently specifying which parameter instances are to be inherited. {Note Explicit Inheritance}

An inherited instance component of a module may be a component instance of a parameter, as illustrated by the following example. A signature for sets over an EQ structure is given by:

```
signature SET = sig
    instance E: EQ
    type set

    val singleton: E.elem -> set
    and union: set * set -> set
    and diff: set * set -> set
    and member: E.elem * set -> bool
    and subset: set * set -> bool
    and eqsets: set * set -> bool
end
```

An implementation of sets using lists with equality is:

```
module SetFromList (LE: LISTEQ, LF: LIST) : SET
    instance E = LE.E
    type set = mkset of E.elem list

    val singleton e = mkset [e]
    and union(mkset l, mkset m) =
            mkset(LF.filter((fun x. not(LE.member(x,m))), l) @ m)
    and diff(mkset l, mkset m) =
            mkset(LF.filter((fun x. not(LE.member(x,m))), l))
    and member(e, mkset l) = LE.member(e,l)
    and subset(mkset l, mkset m) = LF.forall((fun x. LE.member(x,m)), l)
    val eqsets(s1,s2) = subset(s1,s2) / subset(s2,s1)
end
```

Here the inherited instance `E` is defined to be a component of the parameter instance `LE`. Note also that the other parameter `LF` is used only for implementation and so is not inherited.

2.3. Sharing

The inheritance relation between instances defines a dependency hierarchy, and since several instances may be built on the same antecedents, the form of this hierarchy is a directed acyclic graph rather than a tree. For example, Figure 1 shows the instance hierarchy for Luca Cardelli's prototype SML compiler. The sharing of antecedents among instances is not just incidental, for the common antecedents form the basis for communication between instances.

In the absence of parametric modules, any required sharing between instances of modules can be insured "by construction". That is, the hierarchy is built from the bottom up, and later instance constructions refer to specific shared instances created earlier.

When parametric modules are introduced, it sometimes becomes necessary to place explicit sharing constraints on parameter instances. These sharing constraints express assumptions about

common antecedents that are essential for integrating the resources provided by the different parameters; they insure that the parameters "fit together" properly. In ordinary polymorphic types, sharing is expressed by repeated occurrences of a type variable, as in

```
val map: ('a -> 'b) * 'a list -> 'b list
```

Our problem is to express sharing of component instances as well as types. The solution is to introduce a new kind of declaration that indicates sharing by equating *paths* through the inheritance hierarchy, where a path is a sequence of subinstance names separated by "." and terminating with either a subinstance name or a type name. The syntax for these declarations is {Note Sharing Notation}

```
sharing path1 = path2 = ... = pathn
```

To illustrate the problem and its solution in detail, consider the following set of signature and module definitions which might be part of a facility for bit-mapped graphics.

```
signature POINT = sig
    type point
    val point: int * int -> point
    and x_coord: point -> int
    and y_coord: point -> int
    and draw: point -> unit
    . . .
end

signature RECT = sig
    instance P: POINT
    type rect
    val rect: P.point * P.point -> rect
    and top_left: rect -> P.point
    and bottom_right: rect -> P.point
    and draw: rect -> unit
end

module RectMod(P: POINT) : RECT
    instance P = P
    type rect = mkrect of P.point * P.point
    . . .
end

signature CIRCLE = sig
    instance P: POINT
    type circle
    val circle: P.point * int -> circle
    and center: circle -> P.point
    and radius: circle -> int
    and draw: circle -> unit
end

module CircleMod(P: POINT) : CIRCLE
    instance P = P
    type circle = mkcircle of P.point * int
    . . .
end
```

```
signature GEOMETRY = sig
    instance R: RECT
    instance C: CIRCLE
    sharing R.P = C.P
    val circumscribe: R.rect -> C.circle
    . . .
end

module GeometryMod(R: RECT, C: CIRCLE) : GEOMETRY
    instance R = R
    instance C = C
    sharing R.P = C.P

    val circumscribe r = ...
    . . .
end
```

Note the sharing declarations in both the `GEOMETRY` signature and the `GeometryMod` module. These indicate that the parameters `R` and `C` should be based on the same `POINT` component (named `P` in both `RECT` and `CIRCLE`).

Now suppose we define two different instances of `POINT` corresponding to two coordinate systems (e.g. transformed versions of an underlying screen coordinate system), and then define `RECT` and `CIRCLE` instances based on these two coordinate systems.

```
instance Point1: POINT = ...     (expresses one coordinate system)

instance Point2: POINT = ...     (a different coordinate system)

instance Rect1 = RectMod(Point1)

instance Rect2 = RectMod(Point2)

instance Circle1 = CircleMod(Point1)

instance Circle2 = CircleMod(Point2)
```

Then we can create a `GEOMETRY` instance based on `Rect1` and `Circle1`, or `Rect2` and `Circle2`, but not based on `Rect1` and `Circle2`, because the sharing constraint would be violated.

```
instance Geometry1 = GeometryMod(Rect1, Circle1)        (OK)

instance Geometry2 = GeometryMod(Rect2, Circle2)        (OK)

instance GeometryX = GeometryMod(Rect1, Circle2)        (WRONG)
```

It is important to note that the sharing specification in `GeometryMod` is essential for proper type checking of the module, since the functions from the parameter instances `R` and `C` will attempt to interact in terms of the type `point` that they inherit from their respective `POINT` components `R.P` and `C.P`. The sharing declaration will allow the type checker to identify these two versions of the type `point` (i.e. `R.P.point = C.P.point`).

2.4. Type checking

2.4.1. Signature matching

The type checking of module definitions and instantiations involves one way matching of a *candidate* signature against a *target* signature. When checking the body of a module against its declared signature, the declared signature is the target and the inferred signature of the body is the candidate. When checking a module application, the declared parameter signature is the target and the signature of the corresponding actual parameter is the candidate. The matching is based on the assumption that polymorphic types appearing in a signature are generic, that is, implicitly universally quantified.

The process of signature matching works as follows:

1. Match corresponding instances
   a. For each instance component in the target

      ```
      instance X: SIGX
      ```

      there is a corresponding instance with the same name

      ```
      instance X: SIGX'
      ```

      in the candidate, and `SIGX'` matches `SIGX`.
   b. There are no extra instances in the candidate not corresponding to target instances.
2. Match types
   a. Identify corresponding instances in the candidate and target.
   b. For each abstract type in the target there is a corresponding type (abstract or concrete) in the candidate with the same arity (i.e. an abstract type can be matched by either a concrete or an abstract type; in the case of a concrete type, the data constructors are discarded). {Note Exact Matches}
   c. For each concrete type specification

      ```
      type ('a1,..,'am) tycon = con_1 of ty1 | .. | con_n of tyn
      ```

      in the target there is a corresponding specification

      ```
      type ('a1,..,'am) tycon = con_1 of ty1' | .. | con_n of tyn'
      ```

      in the candidate (modulo change of bound type variables and permutation of the order of the data constructors) and each `tyi` is an instance of the corresponding `tyi'` (i.e. the candidate data constructors are at least as general as their counterparts in the target).
3. Match value and exception bindings.
   a. Identify corresponding instances *and* types from candidate and target.
   b. For every value specification

      ```
      val id: ty
      ```

      in the target, there is a corresponding specification

      ```
      val id: ty'
      ```

      in the candidate, and `ty` is an instance of `ty'`.
   c. Specifications of exceptions in the target and candidate must match exactly, since exception exported from a module must have ground types.

2.4.2. Typing instance components

Consider the instance `IntOrd` from the example on lexicographic orderings, whose signature is `ORDSET`. What is the type of the qualified identifier `IntOrd.le`? The type specification of `le` in `ORDSET` is `elem*elem->bool`, where elem is the type component of the signature, so to

get the type of `IntOrd.le` we instantiate that type with the type bound to elem in `IntOrd`, namely `int`, yielding the type `int*int->bool`.

The type specifications for values and exceptions in a signature are specifications relative to the actual type components of instances of the signature, so to get the true type of a value or exception component we must combine the type schema of the signature with the type bindings in the instance itself. The types in the signature schema (and the instance components as well, which behave like types in this respect) are really bound dummy variables, and the nature of their binding is a form of existential quantification. The nature of these schemas is also closely related to dependent sum types in intuitionistic type theory.

In summary, the rule for determining the type of a value or exception component of an instance is to take the type schema for that component in the signature and replace all type constructor names bound in that instance or its antecedents with their bindings in the instance. (When the instance is a module formal parameter, we create dummy type components for the parameter and use them to instantiate the schema.)

3. Derived forms and extensions

This section presents some derived forms which make the module facilities considerably less cumbersome to use.

3.1. Global references to signatures and instances.

In the basic module proposal, any instance that is to be used in a module must be passed as a parameter. This is a simple and uniform convention, but it is also somewhat unnatural in a case where the same instance will be passed as a parameter every time the module is instantiated. For example, the same `List` instance implementing the list manipulation utilities will probably always be passed as the `LP` parameter to the `SetFromList` module whenever it is instantiated. For such situations, it would be convenient to simply refer to the particular instance (`List` in this case) as a global instance name. The definition of `SetMod` would then become

```
module SetFromList (LE: LISTEQ) : SET
    instance E = LE.E
    type set = mkset of E.elem list

    val singleton e = mkset [e]
    and union(mkset l, mkset m) =
          mkset(List.filter((fun x. not(LE.member(x,m))), l) @ m)
    and diff(mkset l, mkset m) =
          mkset(List.filter((fun x. not(LE.member(x,m))), l))
    and member(e, mkset l) = LE.member(e,l)
    and subset(mkset l, mkset m) = List.forall((fun x. LE.member(x,m)), l)
    val eqsets(s1,s2) = subset(s1,s2) / subset(s2,s1)
end
```

This new definition is equivalent to the old definition together with the proviso that `List` is always to be used as the second parameter when instantiating `SetFromList`. We have in fact already been following this practice of using global identifiers in the case of signatures.

The use of such global (or free) identifiers implies that there must be some context in which they are bound, and the fact that we want to separately compile modules implies that this context should be persistent, i.e. that it should exist independently of any particular invocation of ML. These requirements can be satisfied by introducing the notion of a *system* or *library*. *that consists of a permanent collection of named signatures and (precompiled) modules and directions for reconstituting certain named instances "on demand" (alternatively, it may be possible to make the instances themselves persistent). The system concept could also provide programming support features such as the ability to automatically recompile all modules which are affected by a change in one module (like*

the "make" command in Unix). We will leave the detailed description of a system facility to a separate document.

3.2. Direct instance definitions

In the majority of cases, a module will only be instantiated once, creating a single instance. Since this usage is so common, it is worthwhile to provide a special syntax direct definition of a single instance:

```
inst
    declarations
end
```

This form of definition does not allow parameters, of course. Any other instances used in the body must be referenced as globals. (Note Instance Syntax)

As examples, here are the abbreviated forms of the definitions of the instances `Stack` and `IntOrd`.

```
instance Stack: STACK = inst
    type 'a stack = nilstack' | push' of 'a * 'a stack
    exception pop: unit and top: unit
    val nilstack = nilstack'
    and push = push'
    . . .
end

instance IntOrd: ORDSET = inst
    type s is int
    val le = op <=
end
```

3.3. Inheritance declarations

To avoid the awkward form

```
instance E = E'
```

used in the `ListEqMod` example to cause the parameter `E'` to be inherited as an instance component, we introduce the declaration

```
inherit instance-name-seq
```

This declaration can only be used in a module, and it indicates which of the module's parameters are to be inherited as instance components of the module's instances. The definition of `ListEqMod` becomes

```
module ListEqMod (E: EQ) : LISTEQ
    inherit E
    . . .
end
```

3.4. Opened instances

The use of qualified names (or "paths") to refer to components of instances can become very tedious in deeply nested hierarchies such as the SML compiler example (Appendix C). In that example we find specifications like

```
val LookupVar : E.T.L.Ide * E.Env * int * (E.T.L.Ide * int) E.T.A.Associ
                    -> E.T.TypeExp * Displacement
```

where a module high in the hierarchy is referring to types introduced several levels below. Equally cumbersome names will have to be used in expressions.

Often these qualified names are not essential, because there is only one binding of a given identifier among the antecedents and so no danger of ambiguity. To gain direct rather than qualified access to the identifiers bound in an instance, we can declare that instance to be "open". In a signature, we use an instance specification of the form

```
open instance name: signature
```

while in module and instance definitions we use the declaration

```
open instance-name1 . . instance-namen
```

to gain direct access to the bindings of the named instances. For example, the signature LISTEQ and the module ListEqMod could be written as

```
signature LISTEQ = sig
    open instance E: EQ
    val member: elem * elem list -> bool
    and eqlists: elem list * elem list -> bool
end

module ListEqMod (E: EQ) : LISTEQ
    inherit E
    open E
    val member(e,nil) = false |
        member(e,e'::l) = eq(e,e') orelse member(e,l)
    and eqlists(nil,nil) = true |
        eqlists(e1::l1,e2::l2) = eq(e1,e2) andalso eqlists(l1,l2)
end

instance IntListEq = ListEqMod(IntEq)
```

The effects of "open" specifications in signatures and "open" declarations in modules are independent. An open specification in a signature has effect with respect to a client using instances of that signature, flattening out the signature from the client's point of view. Thus the open specification in the signature `LISTEQ` indicates that the "effective" signature is

```
sig
    instance E: EQ
    type elem                                   (= E.elem)
    val eq: elem * elem -> elem                 (= E.eq)
    and member: elem * elem list -> bool
    and eqlists: elem list * elem list -> bool
end
```

so that a qualified name like `IntListEq.eq` is valid (and is equivalent to `IntListEq.E.eq`).

The open declaration in a module makes the bindings of the named instances directly available *within that module*. In other words, the scope of bindings made accessible by an open declaration in a module is limited to the module body -- they are not exported along with the module's bindings. Hence the open declaration in `ListEqMod` makes it possible to refer to `eq` without qualification in the remainder of the body of the module, but does not affect the module's external interface (even if its signature was inferred rather than declared). In order to cause an instance component of a module to be open in an inferred signature, one must use declarations such as

```
open instance E = E'
```

or


```
open inherit E
```

Open declarations can also be used in ordinary ML expressions and declarations, where the scoping of the revealed bindings follows the usual rules. Thus if we declare

```
open List
```

at top level, the identifiers `filter`, etc are accessible as ordinary bound variables until they are rebound (think of the top level ML interaction as a module body with indefinite extent).

3.5. Views

Sometimes one wants to make an instance `X` of some signature `SIG1` masquerade as an instance of some other, presumably simpler signature `SIG2`, so that `X` can be passed to a module requiring a `SIG2` parameter. Often this can be accomplished by restricting and renaming the bindings of `X`. This process can be thought of as creating a new "view" of the instance [GOG83], or as applying a "signature morphism" to the instance. Such a signature transformation can easily be expressed as a parameterized module, or as an *ad hoc* definition of a new instance derived from the old. For example, suppose we wanted to consider an instance of `SET` as an `ORDSET` ordered by the subset relation, so that we could lexicographically order lists of such sets. The module that expresses this view uniformly on all instances of `SET` is

```
module SetOrd (S: SET) : ORDSET
    type elem is S.set
    val le = S.subset
end
```

Using this module we can get an ordering of lists of sets of integers by

```
instance IntSetListOrd = LexOrdMod(SetOrd(IntSet))
```

Alternatively, we could create an *ad hoc* view of `IntSet` as an `ORDSET` as follows.

```
instance IntSetListOrd = LexOrdMod(inst type elem is IntSet.set
                                        val le = IntSet.subset
                                   end)
```

It is debatable whether any additional syntactic sugaring is needed for this process of creating new views of instances, but we have considered some additional forms such as

```
select type T >> S
       val f >> b
       . . .
from X
```

for selecting a renaming certain components of an instance X;

```
omit type T
     val f
     . . .
from X
```

for simply restricting the signature of X without renaming; and

```
rename type T >> S
       val f >> g
       . . .
in X
```

for renaming certain components while leaving the rest of the signature unchanged.

3.6. Possible future extensions

If we allow instance parameters and global references to instances in modules, the question arises as to whether it would be sensible to allow the same to be done with modules themselves. Modules are typed as functions from their parameter signatures to their result signatures, so they can be viewed as typed mappings over environments. The notion of higher order modules which accept and produce such mappings seems like a reasonable analogue of ordinary higher order functions. However, it is not clear that the type theory which justifies instance parameters can be stretched to accomodate module parameters as well. It is also not clear how useful such an extension would be at this time.

In the other direction, it sometimes seems natural to define a module with value parameters as well as or instead of instance parameters. Recall that Simula classes allowed value parameters, and modules that define state but not type behave rather like Simula classes. The current design allows a form of value parameters to modules, in the sense that the values can be packaged as a simple instance.

This proposal maintains a strict segregation between values and their types on the one hand, and modules and instances and their signatures on the other. Thus, values cannot be passed to modules, and instances cannot be passed to ordinary functions. A long range problem is to understand whether this separation can be relaxed to some extent. Intuitionistic type theory is an example of a formalism that does not maintain such a strict seperation.

References

[BUR77] R. M. Burstall and J. A. Goguen, *Putting theories together to make specifications*, Proc. 5th Int. Joint Conf on Artificial Intelligence, Cambridge, Mass., August, 1977, pp. 1045-1058.

[CAR84] L. Cardelli, *ML under Unix*, Polymorphism, this issue.

[GOG83] J. A. Goguen, *Parameterized programming*, Proceedings of Workshop on Reusability in Programming, A. Perlis, ed.

[MAC81] D. B. MacQueen, *Structure and parameterization in a typed functional language*, Symp. on Functional Languages and Computer Architecture, Gothenburg, Sweden, June, 1981, pp. 525-537.

[MIL84] R. Milner, *A proposal for Standard ML*, Polymorphism, this issue.

Appendix A. Notes

Instance Terminology

I am not entirely happy with the term "instance". It has a certain currency in the literature of modules that makes it seem the conventional term for roughly the concept that I am trying to capture. But it is not descriptive of the object it names. I would prefer the word "environment", but the well established generic meaning of that term makes it unsuitable for a specialized technical use, and it is too long. In previous work on Hope I used the term "structure", which is suggestive of the notion of a mathematical structure such as an algebra, vector space, etc. The term "algebra" also has its appeal, since instances can be seen as a generalization of multi-sorted algebras. (Work with Rod Burstall on generalizing Hope also lead to the term "variety" for what is here called a "signature"; we called modules "implementations".) However, in the end I have stuck with the rather colorless and nondescriptive "instance" as the safest and least objectionable choice.

Type Declaration Notation

This design for modules seems to have uncovered a compelling need for an ordinary, transparent, "naming" type declaration (i.e. one in which we simply name an existing type structure). In fact, I have become convinced that this style of type declaration should be the standard one. It must be granted that type abstraction probably requires the "generation" of a new, unique type, but the opaque, generative nature of concrete type declaration seems to be an aberration that is only barely justified by several practical advantages (additional type information is born by the data constructors, and they also mediate the unwinding of recursive types). I believe that a simplified and rationalized type system could do entirely without generative type declarations.

In the mean time, it might be more consistent if the declaration keyword `type` and the use of "`=`" were reserved for the simple, transparent form of type declaration. The unusual, generative forms of type declaration would have special keywords, such as "`abstype`" and "`datatype`", and they might also use "`is`" instead of "`=`" as the binding symbol.

Qualified Identifier Notation

In this proposal I have adopted the Pascal style of notation for component selection using the period. Mesa, Ada, Euclid, and Modula (I and II) also follow this usage. There may well be a serious syntactic clash with the current use of the period in function and case expressions. In that case there is always the Clu notation, using the "$" sign, as an alternative.

Explicit Inheritance

It seems most natural to view the parameter instance bindings as constituting a form of "environment of definition" for the declaration forming the module body, and therefore these bindings should not by default be included in the resulting environment (separation of environment of definition from defined environment). If we were to decide to include them by default, we would face the problem of subsequently removing those instance bindings which were only meant to be used in implementation. At one stage in the development of the design, I considered using the declared result signature of a module as a filter to remove such auxiliary bindings.

Sharing Notation

Earlier attempts at expressing sharing used a notation closer to that of polymorphic types, where sharing is indicated by repeated occurrences of a type variable. There are two problems with this approach when it is extended to cover instances. First, the notation becomes somewhat difficult to read when expressing sharing of subinstances nested several layers deep in the inheritance hierarchy. Second, the notation is slightly misleading in that the relation between a signature and its subinstance and type components is not analogous to the relation between a type constructor and its arguments. The type and instance components of a signature are not parameters, but

are locally bound by a kind of existential quantification. The actual type and instance components are associated with the instance itself, and not its signature. The current notation of path equations was therefore chosen as a more direct and honest solution of the problem.

Exact Matches

The policy chosen for signature matching is that the target signature and the candidate should specify exactly the same set of names of each kind. An alternative would be to allow the candidate to specify superfluous names which would be ignored in the matching process, but this was viewed as less safe. The one exception to the rule that the names should agree is that a concrete type specification in the candidate is allowed to match a simple (i.e. abstract) type specification in the target, which means that the data constructors associated with the type in the candidate are ignored.

Instance Syntax

The syntax for direct instance definition of an instance allows anonymous instances to be defined, and therefore resembles the notation for signatures. Declarations of named signatures and instances conform to the pattern

keyword name = expression

But the declaration of modules does not conform to this pattern, which is an irregularity that probably should be fixed. The one justification for this nonuniformity is that there is no occasion in the current design for the use anonymous module expressions. All modules must be named and declared at top level.

Appendix B: Syntax

Basic Forms

SIGNATURES *sig*

sig ::=
- *sig_id* (signature name)
- `sig` *spec*$_1$ .. *spec*$_n$ `end` (signature specification)

spec ::=
- `type` *tb*
- `type` {*tvarseq*$_1$} *tycon*$_1$ `and` .. `and` {*tvarseq*$_n$} *tycon*$_n$
- `val` {op} *id*$_1$: *ty*$_1$ `and` .. `and` {op} *id*$_n$: *ty*$_n$
- `exception` *id*$_1$: *ty*$_1$ `and` .. `and` *id*$_n$: *ty*$_n$
- *inst_spec*
- *share_spec*

inst_spec ::=
- `instance` *inst_id*$_1$: *sig*$_1$ `and` .. `and` *id*$_n$: *sig*$_n$

share_spec ::=
- *share path_eq*$_1$ *and* .. *and path_eq*$_n$

path_eq ::=
- *path*$_1$ = .. = *path*$_n$

path ::=
- *tycon*
- *inst_path*[. *tycon*]

inst_path ::=
- *inst_id*$_1$ . .. . *inst_id*$_n$ ($n \geq 1$)

dec ::=
- `signature` *sb* (signature declaration)

sb ::= (signature binding)
- *sig_id* = *sig*
- *sb*$_1$ `and` .. `and` *sb*$_n$ (multiple, n ≥ 2)

INSTANCE BINDINGS *ib*

ib ::=
- *inst_id* = *inst*
- *ib*$_1$ `and` .. `and` *ib*$_n$ (multiple, $n \geq 2$)

inst ::=
- *inst_path*
- *mod_id* (*inst_seq*) (module instantiation)

dec ::=
- *instance ib* (instance declaration)

MODULES *mod*

```
mod ::=
    module mod_id ( {param_spec_seq} ) {: sig}
      mod_spec*
      dec
    end

param_spec ::=
    inst_id : sig

mod_spec ::=
    mod_inst_dec
    share_spec

mod_inst_dec ::=
    instance inst_id = inst_path
```

QUALIFIED NAMES

```
var' ::=
    var
    inst_path.var

con' ::=
    con
    inst_path.con

exid' ::=
    exid
    inst_path.exid

tycon' ::=
    tycon
    inst_path.tycon
```

Note: These generalized names may be used in expressions and types, but not in binding occurrences in declarations.

Derived Forms

OPEN

```
spec ::=
    open inst_spec

mod_spec ::=
    open mod_inst_dec

dec ::=
    open inst_id+
```

INHERIT SPECIFICATION

```
mod_spec ::=
    inherit inst_id+
```

DIRECT INSTANCE DEFINITION

```
inst ::=
    inst
      mod_spec*
      dec
    end
```

Note: Sharing specs are unnecessary in direct instance definitions.

Appendix C: Examples

Unification

This example defines a generic unification module for unifying terms in different languages.

```
signature EXPR = sig
    type exp

    val isvar: exp -> bool
    and prune: exp -> exp
    and occurs: exp * exp -> bool
    and instantiate: exp * exp -> unit
    and sameop: exp * exp -> bool
end

signature UNIFY = sig
    instance Exp: EXP

    val unify: exp * exp -> unit
    exception unify: exp * exp
end


module Unify (Exp: EXP) : UNIFY
    exception unify: Exp.exp * Exp.exp

    val rec unify(e,e') =
            let val e = Exp.prune e
                and e' = Exp.prune e'
            in if Exp.isvar e
               then if Exp.occurs(e, e')
                    then if Exp.isvar e' then () else raise unify (e,e')
                    else Exp.instantiate(e,e')
               else if Exp.isvar e'
                    then unify(e',e)
                    else if Exp.sameop(e,e')
                         then unifyargs(args e, args e') ? raise unify (e,e')
                         else raise unify (e,e')

     and unifyargs(nil,nil) = () |
         unifyargs(arg::args, arg'::args') =
             (unify(arg,arg'); unifyargs(args,args')) |
         unifyargs _ = escape unifyargs

 end
```


Symbol Tables

This example constructs symbol tables from finite mappings and stacks.

Signatures

```
signature EQ =  sig
    type elem
    val eq: elem * elem -> bool
end

signature MAP = sig
    instance Id: EQ
    type 'a map
    val nilmap: 'a map
    and assoc: Id.elem * 'a * 'a map -> 'a map
    and select: Id.elem * 'a map -> 'a
    and inmap: Id.elem * 'a map -> bool
    exception select: Id.elem
end

signature STACK = sig
    type 'a stack
    val nilstack: 'a stack
    and push: 'a * 'a stack -> 'a stack
    and empty: 'a stack -> bool
    and pop: 'a stack -> 'a stack
    and top: 'a stack -> 'a
    exception pop: unit and top: unit
end

signature SYNTAB = sig
    instance Id: EQ
    type 'a table
    val nilst: 'a table
    and extend: 'a table -> 'a table
    and contract: 'a table -> 'a table
    and putst: Id.elem * 'a * 'a table -> 'a table
    and access: Id.elem * 'a table -> 'a
    exception access: Id.elem
    and putst: Id.elem
end
```

Implementations

```
module Map (Id: EQ) : MAP
    type 'a map = nilmap' | assoc' of Id.elem * 'a * 'a map
    exception select: Id.elem
    val nilmap = nilmap'
    and assoc = assoc'
    and select(i1,assoc'(i2,v,m)) = if Id.eq(i1,i2) then v else select(i
        select(i,_) = raise select i
    and inmap(i, nilmap') = false |
        inmap(i1, assoc'(i2,_,m)) = Id.eq(i1,i2) orelse inmap(i1,m)
end

module Stack () : STACK
```

```
    type 'a stack = nilstack' | push' of 'a * 'a stack
    exception pop: unit and top: unit
    val nilstack = nilstack'
    and push = push'
    and empty(nilstack') = true |
        empty _ = false
    and pop(push'(_,s)) = s |
        pop _ = escape pop
    and top(push'(x,_)) = x |
        top _ = escape top
end

module Symtab (M: MAP, S: STACK) : SYMTAB
    instance Id is M.Id
    type 'a table = mktable of ('a array) stack
    exception access: Id.elem
    and contract: unit
    val nilst = mktable(S.nilstack)
    and extend(mktable s) = mktable(S.push(M.nilarray,s))
    and contract(mktable s) = mktable(S.pop s) trap S.pop raise contract
    and putst(i,x,mktable s) = mktable(S.push(M.assoc(i,x,S.top s),S.pop s))
                               trap S.top raise putst i
    and access(mktable s, i) =
            (let m = S.top s
             in if M.inmap(i,m)
                   then M.select(i,m)
                   else M.access(mktable(S.pop s), i)
            ) trap S.top raise access i
end
```

Note: In the definition of the module Symtab, I haven't bothered to define table as a abstract type. Since the result signature of the module was declared to be SYMTAB, in which table is specified simply as a type, the constructor mktable will not be exported, so externally table has become an abstract type.

Here is an alternative definiton of the Map module:

```
module Map (Id: EQ) : MAP
    type 'a map is (Id.elem * 'a) list
    exception select: Id.elem
    val nilmap = nil
    and assoc(i,v,m) = (i,v)::m
    and select(i1,(i2,v)::m) = if Id.eq(i1,i2) then v else select(i1,m) |
        select(i,_) = raise select i
    and inmap(i, nil) = false |
        inmap(i1,(i2,_)::m) = Id.eq(i1,i2) orelse inmap(i1,m)
end
```

In this definition, the type constructor map is simply an abbreviation for a type schema with one free type variable. We can then write

```
instance MapInt = Map(IntEq)

hd(MapInt.assoc(5,true,MapInt.nilmap))          (* => 5,true: int*bool *)
```


Prototype Standard ML Compiler

This example gives the module structure of Luca Cardelli's prototype Standard ML compiler, which was written in Pose 3 of Unix ML (a close approximation of Standard ML with a limited module facility). Figure 1 shows a map of the *instance* structure of the compiler. The signatures are simply edited versions of the signatures automatically derived by the ML type checker. The outline given here makes no use of the direct instance abbreviation, but the inherited instance specification is used.

List

```
signature LIST = sig
    val forall : ('a -> bool) -> ('a list) -> bool
    and exists : ('a -> bool) -> ('a list) -> bool
    and choose : ('a -> bool) -> ('a list) -> 'a
    and midmap : ('a -> 'b) * (unit -> 'b) -> 'a list -> 'b list
    and midapply : ('a -> unit) * (unit -> 'b) -> 'a list -> unit
    and midfold : ('a * 'b -> 'b) * ('b -> 'b) -> 'a list -> 'b -> 'b
end

module ListMod () : LIST
    . . .       {/usr/lib/ml/List.ml}

instance List = ListMod ()
```

Association

```
signature ASSOCIATION = sig
    type 'a Association = association of ('a list) ref

    val EmptyAssociation : unit -> 'a Association
    and LookupAssociation : ('a -> bool) * (unit -> 'a) *
                            'a Association -> 'a
    and MapAssociation : ('a -> 'b) * 'a Association -> 'b list
    and ApplyAssociation : ('a -> unit) * 'a Association -> unit
end

module AssociationMod (L: LIST) : ASSOCIATION
    . . .       {/usr/lib/ml/Association.ml}

instance Association = AssociationMod(List)
```

Option

```
signature OPTION = sig
    type rec 'a Option = none | some of 'a
end

module OptionMod () : OPTION
    type rec 'a Option = none | some of 'a
end

instance Option = OptionMod ()
```

TimeStamp

```
signature TIMESTAMP = sig
    type Stamp;
    val Stamp : unit -> Stamp
```

```
    val SameStamp : Stamp * Stamp -> bool
    val OlderStamp : Stamp * Stamp -> bool
end

module TimeStampMod () : TIMESTAMP
    . . .       {/usr/lib/ml/TimeStamp.ml}

instance TimeStamp = TimeStampMod ()
```

Lex

```
signature LEX = sig
    type rec Ide = ide of string
    val PrintIde : Ide * stream -> unit
end

module LexMod () : LEX
    type Ide = ide of string
    val PrintIde(ide String, Stream: stream): unit =
          output(Stream,String)
end

instance Lex = LexMod ()
```

TypeExp

```
signature TYPEEXP = sig
    instance O: OPTION
    instance T: TIMESTAMP
    instance A: ASSOCIATION
    instance L: LEX
    type rec TypeExp = typeVar of T.Stamp * (TypeExp O.Option ref) |
                       typeOper of L.Ide * T.Stamp * (TypeExp list)

    val NewTypeVar : unit -> TypeExp
    val NewTypeOper : L.Ide -> (TypeExp list) -> TypeExp
    val Prune : TypeExp -> TypeExp
    val OccursInType : T.Stamp * TypeExp -> bool
    val EmptyStampEnv : unit -> (T.Stamp * 'a) A.Association
    val LookupStampEnv : T.Stamp * (unit -> 'a) *
                         (T.Stamp * 'a) A.Association -> 'a
    val PrintType : stream * TypeExp -> unit
end

module TypeExpMod(O: OPTION, T: TIMESTAMP, A: ASSOCIATION, L: LEX, Li: LIST)
          : TYPEEXP
    inherit O T A L
    . . .       {/usr/luca/ml/sml/TypeExp.ml}

instance TypeExp = TypeExpMod(Option, TimeStamp, Association, Lex, List)
```

GenericVar

```
signature GENERICVAR = sig
    instance T: TYPEEXP
    type rec FrozenVars = frozenVars of T.TypeExp list;

    val EmptyFrozenVars : FrozenVars
```



```
    and ExtendFrozenVars : T.TypeExp * FrozenVars -> FrozenVars
    and FreshType : T.TypeExp * FrozenVars -> T.TypeExp
end

module GenericVarMod (T: TYPEEXP, Li: LIST) : GENERICVAR
    inherit T
    . . .       {/usr/luca/ml/sml/GenericVar.ml}

instance GenericVar = GenericVarMod(TypeExp, List)
```

Env

```
signature ENV = sig
    instance T: TYPEEXP
    type rec Env = env of EnvCell list
    and EnvCell = nullCell |
                  varCell of T.L.Ide * T.TypeExp |
                  conCell of T.L.Ide * T.TypeExp |
                  typeCell of T.TypeExp * Env |
                  modCell of T.L.Ide * Env

    val EmptyEnv : Env
    and RaiseEnv : Env * int -> Env
    and ExtendValEnv : T.L.Ide * T.TypeExp * Env -> Env
    and ExtendConEnv : T.L.Ide * T.TypeExp * Env -> Env
    and ExtendTypeEnv : T.TypeExp * Env * Env -> Env
    and ExtendModEnv : T.L.Ide * Env * Env -> Env
    and RetrieveVar : T.L.Ide * Env -> T.TypeExp
    and RetrieveCon : T.L.Ide * Env -> T.TypeExp
    and RetrieveType : T.TypeExp * Env -> Env
    and RetrieveMod : T.L.Ide * Env -> Env
end

module EnvMod (T: TYPEEXP) : ENV
    inherit T
    . . .       {/usr/luca/ml/sml/Env.ml}

instance Env = EnvMod(TypeExp)
```

Lookup

```
signature LOOKUP = sig
    instance E: ENV
    type rec Displacement = displNone | displLocal of int * (int list)
                                      | displGlobal of int * (int list)

    val LookupVar : E.T.L.Ide * E.Env * int *
                    (E.T.L.Ide * int) E.T.A.Association
                      -> E.T.TypeExp * Displacement
    and LookupCon : 'a * 'b * 'c * 'd -> E.T.TypeExp * Displacement
    and LookupGlobals : (E.T.L.Ide * int) E.T.A.Association * E.Env * int
                        (E.T.L.Ide * int) E.T.A.Association -> Displacem
end

module LookupMod (E: ENV) : LOOKUP
    inherit E
    . . .       {/usr/luca/ml/sml/Lookup.ml}
```

```
instance Lookup = LookupMod(Env)
```

Errors

```
signature ERRORS = sig
    val Crash : string -> 'a
    val UnboundVal : 'a -> 'b
    val UnboundCon : 'a -> 'b
    val TypeError : unit -> 'a
end

module ErrorsMod () : ERRORS
    . . .       {/usr/luca/ml/sml/Errors.ml}

instance Errors = ErrorsMod()
```

Unify

```
signature UNIFY = sig
    instance T: TYPEEXP

    val Unify : TypeExp * TypeExp -> unit
end

module UnifyMod (T: TYPEEXP, E: ERRORS) : UNIFY
    inherit T
    . . .       {/usr/luca/ml/sml/Unify.ml}

instance Unify = UnifyMod(TypeExp, Errors)
```

AbsSyntax

```
signature ABSSYNTAX = sig
    instance Lk: LOOKUP
    type rec SynModule = synModule of Lk.E.T.L.Ide * SynDec
    and SynExp = synVar of Lk.E.T.L.Ide * (Lk.Displacement ref) |
                 synCon of Lk.E.T.L.Ide * (Lk.Displacement ref) |
                 synInt of int |
                 . . .
    and SynMatch = synMatch of SynClause list
    . . .
    and SynTypeAlt = synTypeConstant of Lk.E.T.L.Ide |
                     synTypeConstructor of Lk.E.T.L.Ide * SynType
end

module AbsSyntaxMod (Lk: LOOKUP) : ABSSYNTAX
    inherit Lk
    . . .       {a copy of the signature specifications}

instance AbsSyntax = AbsSyntaxMod()
```

Analyse

```
signature ANALYSE = sig
    instance AS: ABSSYNTAX
    instance G: GENERICVAR
    sharing AS.Lk.E.T = G.T

    val AnalyseExp : AS.SynExp * AS.Lk.E.Env * int
```



```
                     * (G.T.L.Ide * int) G.T.A.Association
                     * G.FrozenVars -> G.T.TypeExp
    and IntType : G.T.TypeExp
    and StringType : G.T.TypeExp
    and TupleType : G.T.TypeExp list -> G.T.TypeExp
    and FunType : G.T.TypeExp * G.T.TypeExp -> G.T.TypeExp
end

module AnalyseMod (AS: ABSSYNTAX, G: GENERICVAR, U: Unify) : ANALYSE
    inherit AS G
    sharing AS.Lk.E.T = G.T = U.T
    . . .       {/usr/luca/ml/sml/Analyse.ml}

instance Analyse = AnalyseMod(AbsSyntax, GenerivVar, Unify)
```

FamCode

```
signature FAMCODE = sig
    type rec FamCode = famCode of (int * FamOp) list
    and FamOp = opNone |
                opAnd of unit |
                opApplFrame of unit |
                opArray of unit |
                . . .
end

module FamCodeMod ()
    . . .       {copy of signature spec}

instance FamCode = FamCodeMod()
```

Compile

```
signature COMPILE = sig
    instance AS: ABSSYNTAX
    instance FC: FAMCODE

    val Compile : AS.SynExp -> FC.FamCode
end

module CompileMod (AS: ABSSYNTAX, FC: FAMCODE, Er: ERRORS) : COMPILE
    inherit AS FC
    . . .       {/usr/luca/ml/sml/Compile.ml}

instance Compile = CompileMod(AbsSyntax, FamCode, Errors)
```

SML Compiler Module Map

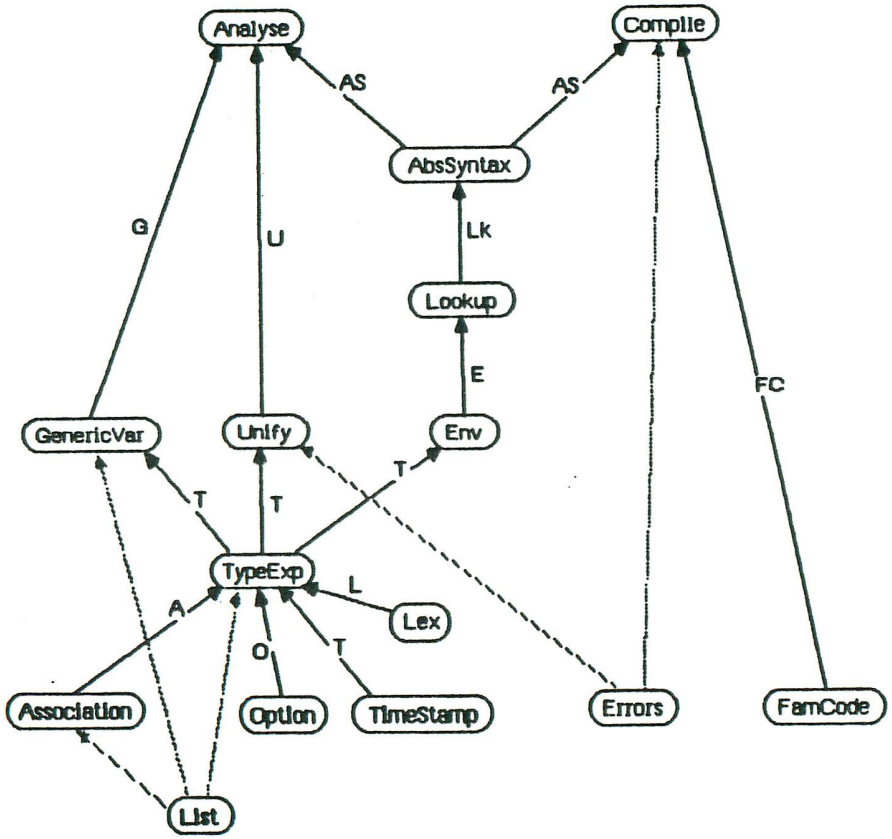


Figure 1

Stream Input/Output

Luca Cardelli

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

This is a proposal for sequential Input/Output in Standard ML. My ideas on I/O have been deeply influenced by the Unix operating system, however I do not consider this as an uncritical suggestion of doing I/O "like in Unix" (e.g. Common Lisp turns out to be particularly close to this proposal). I have been under the influence of other operating systems for longer periods of time, and I have often tried to figure out nice sets of I/O primitives, but nothing came out of it. If some ideas have come out of Unix, I like to think it is because they are inherently simple and powerful.

I think I now understand that I/O issues are often obfuscated by convoluted abstractions and misplaced efficiency considerations. Even when the right primitives are available, they are hidden under layers of software. In a few words, most of the time they have got it wrong.

Programming languages fall into two categories with respect to I/O. System programming languages often directly call the operating system through bizarre parameter lists, offering a jungle of 'primitives' with no mediating abstraction. On the other hand, I/O systems in algorithmic languages are often overrestricted and oversimplified toys (e.g. Pascal), meant as the intersection of all possible I/O systems. Very few languages have files as first class objects, which is the single most important requirement for any non-trivial use of I/O.

It is time to include powerful I/O systems *within* programming languages, particularly because Unix-like operating systems, which can easily support them, are becoming universal standards, and most modern operating systems have the necessary primitives. Crude I/O systems are particularly crippling on personal computers, where many common I/O limitations simply do not apply.

Files and processes

Streams are communication windows on the external world of *files* and *processes*. The basic design goal of this proposal is the following:

> External files and processes should offer a uniform interface.

The above property entails great simplicity and uniformity in operating systems, and has particularly been explored in Unix. Unix is totally based on the *file* paradigm: everything is a file; even a terminal process "is" a pair of files because it is identified with its I/O streams. Similarly, everything could be based on the *process* paradigm. There a file "is" the process which reads and writes it.

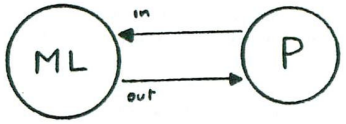
The important point is the consistent view of the outside world. If the file and process interfaces

Unix is a trade mark of Bell Laboratories

are uniform, then programs can be written without embedding knowledge of *who* or *what* they will be talking to. This makes them very flexible and usable in different and unpredictable situations.

Files can be *read* and *written*; temporary files are often alternatively read and written. Processes can accept *questions* and provide *answers*: interesting processes require a dialogue. This suggests (even if it does not demand) that communication streams should be *bipolar*, admitting both input and output actions(1). Bipolar temporary files can be implemented very efficiently by keeping them in memory most of the time. Bipolar channels to other processes can be set up with cleaner protocols than pairs of unipolar channels. A typical bipolar file is one used as an interface between two passes of a compiler. A typical bipolar channel is *terminal*, for terminal I/O (as opposed to unipolar input and output streams).

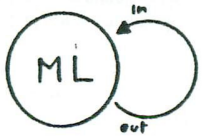
A bipolar stream connected to a process looks like this:



Such a stream is also called a *channel*, or an *external* stream. P may be another ML system.

Any output operation should always be immediately successful. An input operation may have to wait for P to produce some input: in this case the input operation should *hang*. The alternative to hanging is failing on empty stream: this requires *polling* of the input (to discover when data is ready), which is bad style and it is either wasteful (for frequent polling), or it degrades the interaction (for unfrequent polling, e.g. when P is the terminal). To detect the possibility of hanging, it is then necessary to be able to test for empty stream.

A bipolar stream connected to a file is obtained by letting P be the identity process (for an initially empty file) or a process which executes a single initial output, and then becomes the identity (for a file with some initial contents):



Such a stream is also called a *file* or an *internal* stream (as no external process is actually involved). The above discussion implies that input operations should hang on empty files, even if in this situation the wait would never end(2).

Some stream can be unipolar. I/O operations fail on these stream streams if they have the wrong polarity.

(1) Bipolar files are called *pipes* in Unix, and *bidirectional streams* in Common Lisp. This proposal does not in any way assume the existence of pipes, and even under Unix will not be implemented by pipes. Bipolar files are simply pairs of file descriptors pointing to the same file.
(2) The "break" key can be used to stop infinite waits, producing an "interrupt" exception.

External processes

Why should we bother including arbitrary external processes in an I/O system? Aren't files enough?

So far I have tried to show how natural and compelling is the extension of file primitives to processes. Another important observation(3) is that process channels can put a stop to the dozens of meaningless extensions that people will inevitably end up hacking in any ML system. For example somebody may require a "date" primitive, or want to read his mail from ML.

The *dirty* alternatives are: (a) put "date", "mail", etc. primitives in the ML compiler; (b) supply a general "external function call" feature to call the date or mail routines, or any routine, endangering the ML system and undermining its type security; (c) introduce an operating system escape mode.

The *clean* alternative is to dynamically create channels to the "date" and "mail" *processes*: this way all the possible interaction goes through character channels which guarantee the isolation and protection of ML, while allowing two-way communication.

Will external processes change the ML programming style? I do not think so: it will just make possible things which are now impossible or awkward. The "whole world" outside ML will be available from the system by a clean, powerful and secure set of primitives, even if at the moment we do not know what this external world will look like.

Can I implement it?

This I/O system is not implementable in many high level languages (e.g. Pascal), essentially whenever files are not objects in the language. However most operating systems provide the necessary primitives. Basically, it is necessary to be able to open, close, read and write simple character files, and to position at random locations in a file. It must also be possible to perform unbuffered I/O to processes and devices, and to obtain and store file descriptors. Multiple file descriptors for the same file are very convenient but, as far as I can see, not strictly necessary.

A good test for any I/O system is whether it can support an Emacs-like screen editor. If your favorite language cannot do it, it can probably appeal to the operating system. If your operating system cannot do it, it is probably not worth using anyway. Conversely, if your language or operating system can support a screen editor, then there is a good chance that they can implement this proposal.

Here is a non-exhaustive list of systems and languages on which this proposal is certainly implementable:

C/Unix	(VAX, Perq, 68000's, etc.)
VAX/VMS	
Multics	
Apollo AEGIS	
Lisp	(Franz Lisp, Common Lisp, Lisp-Machine Lisp, etc.)

The proposed streams are particularly similar to Unix's pipes and Common Lisp's bidirectional stream.

Stream Primitives

Stream primitives can be divided into two groups: stream creation and stream actions. Streams can be created in a variety of ways (from files, processes and devices), but once created they admit the same set of I/O operations.

Stream creation primitives are:

(3) Suggested by Mark Manasse


```
stream      : unit -> stream
file        : string -> stream
save        : string -> stream -> unit
channel     : string -> stream
terminal    : stream
```

stream creates a new empty stream at every invocation.

file creates a stream out of a file (the argument is the file name). The file is not affected by this operation, nor by further operations on the stream(s) extracted from it (except "save"). If the file does not exist, an empty one is created.

save stores the *current* contents of a stream on a file (the first argument is a filename). The stream is not affected; any preexisting file having that filename is deleted. Because of the way "file" operates, there is no distinction between empty and non-existing files. Hence a file can be erased by a "save" operation on it with an empty stream as argument.

channel creates a new stream which is a channel to the process identified by its argument. The argument (which is operating system dependent) will usually describe an executable program which is activated as a separate process with its input and output connected to the stream.

terminal is the standard terminal channel: input form this channel will read from the user terminal, and output will print to the user terminal. Auxiliary terminals and other devices can be linked by specialized channel commands. The semantics of the terminal *device* is not specified in detail, to accommodate different kinds of terminals and protocols. Moreover, terminal echoing and buffering are local properties of the device, not of the terminal channel, and are not part of the semantics of the I/O operations. Properties of the terminal device can conceivable be changed when needed by communication to the operating system through an auxiliary channel.

Stream action primitives are:

```
input       : stream -> int -> string
output      : stream -> string -> unit
lookahead   : stream -> (int # int) -> string
caninput    : stream -> int -> bool
```

input inputs n characters from a stream. Input is *unbuffered* (i.e. the characters are available as soon as the process at the other end produces them) and *uninterpreted* (i.e. they may include backspaces, line dels, etc.: all the editing characters normally interpreted by the operating system(4)). The effect of buffering can be achieved by n-char "output" operations at the other end of the stream, and n-char "input" operations at this end. Interpretation must be programmed. If not all the n characters are currently available, the operation *hangs* until they can be read from the stream. The n characters are then *extracted* from the stream and are no longer available to succeeding input operations.

output outputs a string to a stream. Output is *unbuffered* (all the characters are immediately available at the other end of the stream) and *uninterpreted* (all Ascii characters can be transmitted on a stream). Buffering can be achieved by emitting long strings (this is normally going to be more efficient than emitting one character at a time). Interpretation of special signals (e.g. a character meaning "this is the end of the message") has to be done at the other end of the stream. An output

(4) The terminal process normally interprets editing characters locally, and sends out one line at a time. In this case line editing happens even if the stream is uninterpreted. In a different mode (needed for example in screen editors) the terminal process transmits immediately every key stroke. In this case the uninterpreted stream transmits the characters unchanged to the other end, where interpretation must be programmed.

operation *inserts* characters in a stream: any succeeding "output" operation will insert characters after those.

lookahead behaves very much like "input", but does not affect the stream. Given two integers, it returns a segment of a stream: the first integer is the starting position of the segment in the stream (the first character is at position 1), and the second integer is the length of the segment. Like "input", it hangs until enough characters are available. The efficiency of lookahead should be expected to deteriorate for long or remote segment.

caninput determines whether n character are *currently* available for read on a stream: if they are not, it returns false without hanging. The normal use is "if caninput s n then input s n" or "if caninput s (n+m-1) then lookahead s (n,m)" when it is undesirable to hang.

All the above primitives can rise exceptions because of I/O errors, failing with their respective names.

Examples

Here is how to copy a stream to another stream:

```
val CopyStream (InStream: stream, OutStream: stream) : unit =
    while caninput InStream 1 do
        output OutStream (input InStream 1);
```

Without realizing it, we have just written a program which can print a file to the terminal:

```
val PrintFile (FileName: string) : unit =
    CopyStream(file FileName, terminal);
```

Using CopyStream in the "opposite" direction, we can write a program to append memos to the end of (possibly preexisting) memo files; the memo is read from the terminal, and is terminated by the break key which produces an "interrupt" exception:

```
val Memo (FileName: string) : unit =
    let val MemoStream = file FileName
    in (CopyStream(terminal, MemoStream)
        ? save FileName MemoStream)
```

And here is a program which prints the date (under Unix):

```
val Date () =
    CopyStream(channel "/bin/date", terminal);
```

Input prompts can be programmed this way: Prompting_InChar is a version of InChar (i.e. of "input s 1"), which prompts for input.

```
local val lastchar = ref ""
in val Prompting_InChar (stream: stream) (prompt: string) : string =
    (if !lastchar = "\L"
     then output stream prompt
     else ();
     lastchar := input stream 1;
     !lastchar);
```

Note that, because of bipolar streams, we do not need to pass two streams to Prompting_InChar.

Notes

Multiplexing. A stream is said to be input (output) multiplexed when it is used by different parts of a program for conceptually distinct purposes. Multiplexing is possible and well defined. Multiplexed input and output operations simply interleave; any of those operations may affect the result of all the succeeding I/O operations in all the other parts of the program. The predefined *terminal* stream is multiplexed between the user and the ML system: both can do I/O on the user terminal. This multiplexing is set up so that every DEL char (Ascii 127, but this may depend on implementations) typed at the keyboard is captured by the ML system and never reaches the user program. The ML system uses this character to induce an "interrupt" exception in the user program, so that computations can be interrupted even when they are on wait or input-loop on the terminal input.

Device Mapping. Initially the "terminal" *stream* is connected to the standard "terminal" *device*. The terminal stream can be redirected to other devices, files or processes by operating-system commands. However this should rarely be necessary: the correct practice is to parameterize every program with respect to the streams it uses, so that we can pass it "terminal" or some other stream.

ML-to-ML communications. These should happen through auxiliary channels (i.e. not through "terminal" streams). The protocol to establish these channels may require running ad-hoc external processes, or communicating directly with the operating system; hence it is not described here.

Dead channels. If the process at the other end of a channel dies for some reason, and if its death can be detected, then all the operations on that channel should fail. When dead channels cannot be detected, they look just like empty streams.

Saving channels. "save" can be used to write the current contents of an external stream (i.e. the portion which has been already generated by the other side of the channel but not yet absorbed by this side) on a file, without affecting the stream.

Stream lifetime. Only the garbage collector can eliminate streams when they are no longer needed. If that stream is a channel, after the elimination it will look like a dead channel to processes at the other end. Operating systems usually allow only a very small number of files and channels to be active at the same time. If a larger number of streams are generated, and cannot be garbage collected, some of the old files or channels can be temporarily deactivated, until needed again. Deactivate channels are not dead.

Streams never end. There is no way of telling whether a file or a channel is "finished": more input may come at any moment. Still, we want to be able to test for empty stream at any particular moment. The only way to do this is to see whether it is possible to read one more character: at some level this implies a real read operation on the stream. This is why instead of an *emptystream* primitive (which can be defined as "caninput s 1") we have the more explicit *caninput*. Moreover, an n-char caninput is needed in conjunction with n-char input and lookahead operations. Caninput never hangs: it always immediately returns the current state of the stream.

Streams never say ouch! Some devices (typically terminals) may require the transmission of full 8-bit characters to execute special functions, as opposed to the Ascii 7-significant-bits characters. Hence a character should be intended as an 8-bit quantity. Characters with the 8th bit high should be denoted inside strings by a new escape sequence "\#c", where "c" is any of the old characters or escape sequences, e.g. "\#L" (high L), "\#\L" (high linefeed), "\#\^C" (high control-C), etc. Any 8-bit character can be transmitted on a stream; no exceptions. Some systems use particular characters for particular functions (e.g. end-of-transmission), or use the 8th bit for parity checking. In these situations all the anomalous characters have to be transparently encoded and decoded to transmit them on streams. When reading a stream from the terminal, it may be necessary to have an end-of-stream character to terminate the input (e.g. ^D in Unix). This must be explicitly programmed. Note that it may be sufficient to use the break key and trap the corresponding exception.

The power of lookahead. "lookahead" may seem too general. Here is an application where all its power is needed. Suppose one wants to write an input scanner which recognizes arbitrary

regular expressions (these programs actually exist under Unix and are widely used). After recognizing an initial segment of the input, one wants to look ahead a few more characters (without starting from scratch again); a failure to recognize those characters might require to backtrack before the current position. Hence one wants to "lookahead" arbitrary segments of the input, and actually commit himself to "input" only when everything is settled. An alternative could be to have and "uninput" primitive, replacing "lookahead", which puts character back on the stream so that they can be "input" again. Unfortunately "uninput" interacts badly with multiplexing.

Substandards. Robin Milner suggests the following substandard to facilitate implementations. All internal streams are unipolar; "stream" creates unipolar output streams, "file" creates unipolar input streams. "input s 0" and "output s "" " can be used to test the polarity of a stream (because of the exceptions they may generate) and to determine whether an implementation is substandard. Another substandard can consist in implementing a "peek s n" primitive, equivalent to "lookahead s (1,n)", instead of the full "lookahead".

Operating system requests. Operating-system dependent operations can be performed by communication with the operating system itself. This can be done by opening a channel to an operating-system command interpreter. For example, under Unix the current date could be obtained by opening a channel to the "date" process (val DateProcess = channel"/bin/date"; val Date = ReadOneLine DateProcess;), or by opening a channel to a shell, and asking it to execute the "date" command (val ShellProcess = channel"/bin/sh"; output ShellProcess "date val Date = ReadOneLine ShellProcess). Note that this practice is potentially dangerous; it should be used only in cases of extreme necessity, and until we agree on new language primitives which can do the same job.

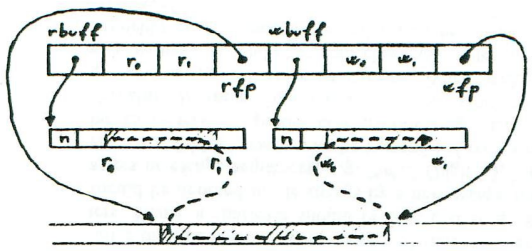
File Protection. If a file is write-protected (e.g. system files), then a "file" operation on it will open an unipolar stream of the appropriate polarity. Similarly, if a file is read-protected. If a file is both read and write protected, then a "no-polarity" stream should be opened, on which both input and output operations fail. Protection schemes seem to be too operating-system dependent to be included in this proposal. Under Unix the protection of a file could be changed by opening a channel to a shell process, and instructing it to change the protection by a Unix command.

Are streams fast? In I/O, speed is only a factor of the implementation effort (given some basic raw speed of the storage devices), because there are so many things that can be optimized. I am confident that the stream primitives can approach the raw speed of the operating system primitives within sensible bounds. However this could be compromised when the operating system gets in the way by not providing simple and efficient character-level I/O primitives.

Implementation notes

These are ideas about the implementation of streams in Unix-like environments. The critical assumption seems to be the presence of multiple file descriptors pointing to the same file. I think that, in this restricted context, multiple descriptors can be simulated in environments which only provide unique descriptors (at some cost), but I do not have very clear ideas about this problem.

Internal Streams. Internal streams are the ones obtained by "file" and "stream". They should be kept in memory as far as possible, and all movements to disk should happen in large chunks. A way of doing this is to use two circular buffers and two file descriptors for every internal stream. The file descriptors are only allocated at the first read and the first write operation respectively, and not at the time of creation of the stream. The semantics of "file" requires that the file originating the stream be copied before any write operation on the stream; the copy should not be made as long as simple read operations are performed.



If the stream contents are shorter than the sum of the lengths of the read and write buffers, everything is kept in memory. When the buffers underflow or overflow, data is read form, or written to the file in buffer units. Note that internal stream I/O is buffered even if the I/O primitives look unbuffered to the user!

The stream primitives can be based on the following InChar and OutChar routine sketches (referring to the previous figure), where +(n) is sum modulo n, and n is the size of the buffers:

```
OutChar(c) =
    if w1 +(n) 1 = w0
    then WriteBufferFull(c)
    else (wbuffer[w1] := c; w1 := w1 +(n) 1)

WriteBufferFull(c) =
    if rfp = wfp {file is empty}
    then  if r1 +(n) 1 = r0
          then (WriteBufferToFile(); wbuffer[w1] := c; w1 := w1 +(n) 1)
          else (rbuffer[r1] := c; r1 := r1 +(n) 1)
    else (WriteBufferToFile(); wbuffer[w1] := c; w1 := w1 +(n) 1)

InChar() =
    if r0 = r1
    then ReadBufferEmpty
    else (result := rbuffer[r0]; r0 := r0 +(n) 1; result)

ReadBufferEmpty() =
    if rfp = wfp {file is empty}
    then  if w0 +(n) 1 = w1
          then Hang()
          else (result := wbuffer[w0]; w0 := w0 +(n) 1; result)
```

```
else (ReadBufferFromFile(); result := rbuffer[r0]; r0 := r0 + (n) 1; result)
```

The "save" primitive may produce unnecessary copying of files, e.g. when a file is saved for the last time it would be sufficient to rename the temporary file implementing the stream instead of copying it. Instead of changing the semantics of save (I don't see any way of doing it which would not require unnecessary copying in some situation), I think one should implement a "copy-by-need" strategy. For example, "save" could always rename the temporary file to the real file without copying, but the first time the stream is subsequently affected, the real file is copied back into a temporary stream file.

External streams. External streams are "terminal" and the ones obtained by "channel". They are unbuffered, but they still need an unbounded input buffer to implement the lookahead and can-input operations. This unbounded input buffer can be implemented as a fixed length in-memory buffer with an overflow file on disk, if needed. The file descriptors here are replaced by device or channel descriptors.

LCF_LSM

A system for specifying and verifying hardware

Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abstract

The LCF_LSM system is designed to show that it is practical to prove the correctness of real hardware. The system consists of a programming environment (LCF) and a specification language (LSM). The environment contains tools for manipulating and reasoning about the specifications. Verification consists in proving that a low-level (usually structural) description is behaviourally equivalent to a high-level functional description. Specifications can be fully hierarchical, and at any level devices can be specified either functionally or structurally.

As a first case study a simple microcoded computer has been verified. This proof is described in a companion report. In this we also illustrate the use of the system for other kinds of manipulation besides verification. For example, we show how to derive an implementation of a hard-wired controller from a microprogram and its decoding and sequencing logic. The derivation is done using machine checked inference; this ensures that the hard-wired controller is equivalent to the microcoded one. We also show how to code a microassembler. These examples illustrate our belief that LCF is a good environment for implementing a wide range of tools for manipulating hardware specifications.

This report has two aims: first, to give an overview of the ideas embodied in LCF_LSM, and second, to be a user manual for the system. No prior knowledge of LCF is assumed.

N.B. This is the second printing of Tech. Report No. 41. Various corrections and additions have been made.

Proving a Computer Correct

with the LCF_LSM Hardware Verification System

Mike Gordon
Computer Laboratory
Corn Exchange Street
Cambridge CB2 3QG

Abtract

A machine-generated correctness proof of a simple computer is described.

At the machine code level the computer has a memory and two registers: a 13-bit program counter and a 16-bit accumulator. There are 8 machine instructions: halt, unconditional jump, jump when the accumulator contains 0, add contents of a memory location to accumulator, subtract contents of a location from accumulator, load accumulator from memory, store contents of accumulator in memory, and skip. The machine can be interrupted by pushing a button on its front panel.

The implementation which we prove correct has 6 data registers, an ALU, a memory, and a microcode controller. This controller consists of a ROM holding 26 30-bit microinstructions, a microprogram counter, and some combinational microinstruction decode logic.

Formal specifications of the target and host machines are given, and we describe the main steps in proving that the host correctly fetches, decodes and executes machine instructions.

The utility of LCF_LSM for general manipulation is illustrated in two appendices. In Appendix 1 we show how to code a microassembler. In Appendix 2 we use the LCF_LSM inference rules to design a hard-wired controller equivalent to the original microcoded one.

N.B. This report should be read in conjunction with *LCF_LSM: A system for specifying and verifying hardware*. University of Cambridge computer laboratory technical report Number 41.

Tactics and Tacticals in Cambridge LCF

Lawrence Paulson

University of Cambridge

July 1983

The tactics and tacticals of Cambridge LCF are described. Tactics reason about logical connectives, substitution, and rewriting; tacticals combine tactics into more powerful tactics. LCF's package for managing an interactive proof is discussed. This manages the subgoal tree, presenting the user with unsolved goals and assembling the final proof.

While primarily a reference manual, the paper contains a brief introduction to goal-directed proof. An example shows typical use of the tactics and subgoal package.

Addenda to the Mailing List

Malcom Atkinson
Dept. of Computing Science
Lilybank Gardens
Glasgow, G12 8QQ
Scotland

S. Arun-Kumar, NCSDCT
Tata Inst. of Fundamental Research
Homi Bhabha Road
Bombay 400 005
India

Ted Biggerstass
ITT Programming
1000 Oronoque Lane
Statford CONN 00497
USA

John Cartmell
Software Sciences Limited
London & Manchester House Park Street
Macclesfield Cheshire SK11 6SR
England

Bob Dickerson
Dept. of Computer Science
Hatfield Polytechnic
P.O. Box 109
Hatfield, Herts., AL10 9AB
England

Hal Eden
Burrough ASG
6655 Lookout Road
Boulder, Colorado 80301
USA

Lars Ericson
New York University
Courant Institute of Math. Sciences
Dept. of Computer Science
251 Mercer Street
New York, N.Y. 10012
USA

Michael Jacobs, Room 3G-242
AT&T Bell Laboratories
150 J.F.Kennedy Parkway
Short Hills, NJ 07078
USA

Edward Johnston
Computer Science Lab
GTE Lab
40 Sylvan Road
Waltham MASS 02254
USA

Neil D. Jones
Institute of Datalogy
Univ. of Copenhagen
Sigurdsgade 41
DK-2200 Copenhagen N
Denmark

Mishra Prateek
Dept. of Computer Science
University of Utah
Salt Lake City, UT 84112
USA

C.M.P.Reade
Dept. of Computer Science
Brunel University
Uxbridge, Middlesex, UB8 3PH
England

Masato Takeichi
Dept. of Computer Science
The University of Electro-Communications
1-5-1 Chofugaoka, Chofu-shi
Tokyo 182
Japan

Mitchell Wand
Computer Science Dept.
Indiana University
101 Lindley Hall
Bloomington, IN 47405
USA
Tel 812-335-5733

Peter Welch
Computer Laboratory
University of Kent
Canterbury
England

J. C. Woodcock
GEC Research Laboratories
Hirst Research Center
East Lane
Wembley, Middlesex HA9 7PP
England

Daniel Zlatin
Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan
S7N 0W0
Canada

David Zuffin, Room IW 1D-414
AT&T Bell Laboratories,
Indian Hill West,
1100 East Warrenville Road,
Naperville, IL 60566
USA