LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Standard ML

Standard ML

by

Robert Harper, David MacQueen and Robin Milner

ECS-LFCS-86-2
(Also published as CSR-209-86)

LFCS Report Series

March 1986

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

STANDARD ML

by

Robert Harper, David MacQueen and Robin Milner

PART I (35 pages)	The Standard ML Core Language (Revised)	Robin Milner
PART II (6 pages)	Standard ML Input/Output	Robert Harper
PART III (35 pages)	Modules for Standard ML	David MacQueen

SUMMARY

In May 1985, the two-year process of designing Standard ML was brought to a successful conclusion at a meeting of fifteen people in Edinburgh. This report records the result.

Part I describes the final form of the Core Language, and supersedes the report "The Standard ML Core Language", CSR-168-84. The Core Language has undergone several careful reviews and revisions over the past two years, and is expected to be quite stable in future. It has been fully implemented and its operational semantics has been defined. The principal changes in this final version are the addition of record types, and a number of minor revisions of the syntax, including the introduction of new keywords fun and datatype.

The basic I/O facilities described in Part II are also the result of several design iterations and appear to be sufficiently well understood that little revision should be necessary.

While the module facilities described in Part III have been under consideration for almost as long as the core language, the fact that they contain quite a few new concepts makes it more likely that the experience of implementing these facilities and making use of them will suggest some revisions. For this reason, the report on modules is more expository and somewhat less formal than the other two documents. At some point in the not too distant future the material in all three documents should be unified to yield a reference manual for Standard ML.

PART I

The Standard ML Core Language(Revised)

Robin Milner
University of Edinburgh

The Standard ML Core Language (Revised)

Robin Milner
University of Edinburgh

Contents

1 Introduction **4**
1.1 How this proposal evolved 4
1.2 Design principles 5
1.3 An example 6

2 The bare language **8**
2.1 Discussion 8
2.2 Reserved words 8
2.3 Special constants 9
2.4 Identifiers 9
2.5 Comments 11
2.6 Lexical analysis 11
2.7 Delimiters 11
2.8 The bare syntax 11

3 Evaluation **12**
3.1 Environments and Values 12
3.2 Environment manipulation 14
3.3 Matching patterns 14
3.4 Applying a match 14
3.5 Evaluation of expressions 15
3.6 Evaluation of value bindings 16
3.7 Evaluation of type and datatype bindings 16
3.8 Evaluation of exception bindings 17
3.9 Evaluation of declarations 17
3.10 Evaluation of programs 18

4 Directives **18**

5 Standard bindings **19**
5.1 Standard type constructors 19
5.2 Standard functions and constants 20
5.3 Standard exceptions 21

6 Standard Derived Forms **21**
6.1 Expressions and Patterns 21
6.2 Bindings and Declarations 23

7 References and equality **24**
7.1 References and assignment 24
7.2 Equality 24

8 Exceptions **25**
8.1 Discussion 25
8.2 Derived forms 25
8.3 An example 26
8.4 Some pathological examples 26

9 Type-checking **28**

10 Syntactic restrictions **29**

11 Relation between the Core language and Modules **30**

12 Conclusion **31**

1 Introduction

1.1 How this proposal evolved

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

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

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

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

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

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

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

The language also owes much to criticisms and suggestions from many other people: Guy Cousineau, Bob Harper, Jim Hook, Gerard Huet, Dave Matthews, Robert Milne, Kevin Mitchell, Brian Monahan, Peter Mosses, Alan Mycroft, Larry Paulson, David Park, David Rydeheard, Don Sannella, David Schmidt, John Scott, Stefan Sokolowski, Bernard Sufrin, Philip Wadler. Most of them have expressed strong support for the design; any inadequacies which remain in the Core Language are my fault, but I have tried to represent the consensus.

1.2 Design principles

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

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

A second design principle is to generalise well-tried ideas where the generalisation is apparently natural. This has been applied in generalising ML "varstructs" to HOPE patterns, in broadening the structure of declarations (following Cardelli's declaration connectives which go back to Robert Milne's Ph.D. Thesis) and in allowing exceptions which carry values of arbitrary polymorphic type. It should be pointed out here that a difficult decision had to be made concerning HOPE's treatment of data types - present only in embryonic form in the original ML - and the labelled records and variants which Cardelli introduced in his VAX version. Each treatment has advantages which the other lacks; each is well-rounded in its own terms. Though a combination of these features was seen to be possible, it seemed at first (to me, but some disagreed!) to entail too rich a language. Thus the HOPE treatment alone was adopted in [5]. However, at the

design meeting of June '84 it was agreed to experiment with at least two different ways of adding labelled records to the Core as a smooth extension. The outcome – decided at the May '85 meeting – is the inclusion of a form of labelled records (but not variants) nearly identical to Cardelli's, and its marriage with the HOPE constructions now appears harmonious.

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

1.3 An example

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

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

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

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

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

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

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

The layered pattern construct "as" was first introduced in HOPE, and yields both brevity and efficiency. The discerning reader may be able to find one 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 bindings (Section 5);
4. References and equality (Section 7);
5. Type-checking (Section 9).

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

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

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

$$dec_1 \ ; _ \ dec_n \ ;$$

each terminated by a semicolon (where each dec_i is not itself of the form *dec* ; *dec*'). In evaluating a program, the bindings created by dec_1 are reported before dec_2 is evaluated, and so on. In the complete language, an expression occurring in place of any dec_i is an abbreviated form (see Section 6.2) for a declaration binding the expression value to the variable "it"; such expressions are called *top-level* expressions.

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

2.2 Reserved words

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

```
abstype   and   andalso   as   case   do   datatype
else   end   exception   fn   fun   handle   if   in
infix   infixr   let   local   nonfix   of   op   open
orelse   raise   rec   then   type   val   with   while
(  )   [  ]   {  }   ,   :   ;   ...   |   ||   =   =>   _   ?
```

2.3 Special constants

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

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

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

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

2.4 Identifiers

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

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

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

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

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

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

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

1. At the start of a component in a record type, record pattern or record expression, id is a record label.

2. Elsewhere in types id is a type constructor, and must be within the scope of the type binding or datatype binding which introduced it.

3. Following **exception**, **raise** or **handle**, or in the context "**exception** *exn* = id", id is an exception name.

4. Elsewhere, id is a value constructor if it occurs in the scope of a datatype binding which introduced it as such, otherwise it is a value variable.

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

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

In the Core language, an identifier may be given infix status by the **infix** or **infixr** directive; this status only pertains to its use as a *var* or a *con*. If id has infix status, then "exp_1 id exp_2" (resp. "pat_1 id pat_2") may occur wherever the application "id(exp_1,exp_2)" (resp. "id(pat_1,pat_2)") would otherwise occur. On the other hand, non-infixed occurrences of id must be prefixed by the keyword "**op**". Infix status is cancelled by the **nonfix** directive. (*Note*: the tuple expression "(exp_1,exp_2)" is a derived form of the numerically labelled record expression "{#1=exp_1,#2=exp_2}", and a similar derived form exists for numerically labelled record patterns. See Section 6.1)

2.5 Comments

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

2.6 Lexical analysis

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

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

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

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

2.7 Delimiters

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

2.8 The bare syntax

The syntax of the bare language is presented in Table 1. The following metasyntactic conventions are adopted:

Conventions

1. The brackets "<< >>" enclose optional phrases.
2. Repetition of iterated phrases is represented by "_"; this must not be confused with "...", a reserved word used in flexible record patterns.
3. For any syntax class *s*, we define the syntax class *s_seq* as follows:

$$s_seq \ ::= \ s$$
$$(s_1, _ , s_n) \quad (n \geq 1)$$

4. Alternatives are in order of decreasing precedence.

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

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

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

3 Evaluation

3.1 Environments and Values

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

A *value* v is either a constant (a nullary constructor), a construction (a constructor with a value), a record, a reference, or a function value. A record value is a set of label-value pairs, written "$\{lab_1$=v$_1$, _ ,lab_n=v$_n\}$", in which the labels are distinct; note that the order of components is immaterial. The labels lab_i in a record value must be either all identifiers, or else they must be the numeric labels #1, #2, _ , #n; the two kinds of label may not be mixed. A function value f is a partial function which, given a value, may return a value or a packet; it may also change the store as a side-effect.

An *exception* e, associated to an exception name exn in any exception environment, is an object drawn from an infinite set (the nature of e is immaterial, but see Section 3.8). A *packet* p=(e,v) is an exception e paired with a value v, called the *excepted* value. 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 or datatype binding	VE		
Exception binding	EE		
Declaration	E	or	p

Table 1: The Syntax of the Bare Language

EXPRESSIONS *exp*

aexp ::=		
var		(variable)
con		(constructor)
{ lab_1=exp_1, _ , lab_n=exp_n }		(record,$n \geq 0$)
(*exp*)		
exp ::=		
aexp		(atomic)
exp aexp	L	(application)
exp : *ty*	L	(constraint)
exp **handle** *handler*	R	(handle exc'ns)
raise *exn* **with** *exp*		(raise exc'n)
let *dec* **in** *exp* **end**		(local dec'n)
fn *match*		(function)
match ::=		
$rule_1$ | _ | $rule_n$		($n \geq 1$)
rule ::=		
pat => *exp*		
handler ::=		
$hrule_1$ || _ || $hrule_n$		($n \geq 1$)
hrule ::=		
exn **with** *match*		
? => *exp*		

DECLARATIONS *dec*

dec ::=		
val *vb*		(values)
type *tb*		(types)
datatype *db*		(datatypes)
abstype *db* **with** *dec* **end**		(abstract datatypes)
exception *eb*		(exceptions)
local *dec* **in** *dec'* **end**		(local dec'n)
dec_1<<;>> _ dec_n<<;>>		(sequence, $n \geq 0$)

PROGRAMS : dec_1 ; _ dec_n ;

PATTERNS *pat*

apat ::=		
_		(wildcard)
var		(variable)
con		(constant)
{ lab_1=pat_1, _ , lab_n=pat_n <<, ...>>}		(record,$n \geq 0$) [a]
(*pat*)		
pat ::=		
apat		(atomic)
con apat	L	(construction)
pat : *ty*	L	(constraint)
var<<:*ty*>> **as** *pat*	R	(layered)

VALUE BINDINGS *vb*

vb ::=		
pat = *exp*		(simple)
vb_1 **and** _ **and** vb_n		(multiple,$n \geq 2$)
rec *vb*		(recursive)

TYPE BINDINGS *tb*

tb ::=		
<<*tyvar_seq*>> *tycon* = *ty*		(simple)
tb_1 **and** _ **and** tb_n		(multiple,$n \geq 2$)

DATATYPE BINDINGS *db*

db ::=		
<<*tyvar_seq*>>*tycon* = *constrs*		(simple)
db_1 **and** _ **and** db_n		(multiple,$n \geq 2$)
constrs ::=		
con_1<<**of** ty_1>>| _ | con_n<<**of** ty_n>>		

EXCEPTION BINDINGS *eb*

eb ::=		
exn<<:*ty*>><< =*exn'*>>		(simple)
eb_1 **and** _ **and** eb_n		(multiple,$n \geq 2$)

TYPES *ty*

ty ::=		
tyvar		(type variable)
<<*ty_seq*>>*tycon*		(type constr'n)
{ lab_1:ty_1, _ , lab_n:ty_n }		(record type,$n \geq 0$)
ty -> *ty'*	R	(function type)
(*ty*)		

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

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

3.2 Environment manipulation

We may write <(id_1,v_1) _ (id_n,v_n)> for a value environment VE (the id_i being distinct). Then VE(id_i) denotes v_i, <> is the empty value environment, and VE+VE' means the value environment in which the associations of VE' supersede those of VE. Similarly for exception environments. If E=(VE,EE) and E'=(VE',EE'), then E+E' means (VE+VE',EE+EE'), E+VE' means E+(VE',<>), etc. This implies that an identifier may be associated both in VE and in EE without conflict.

3.3 Matching patterns

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

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

_	:	the empty value environment <> is returned.
var	:	the value environment <(*var*,v)> is returned.
con<<*pat*>>	:	if v = *con*<<v'>> then *pat* is matched to v', else failure.
var as *pat*	:	*pat* is matched to v returning VE; then <(*var*,v)>+VE is returned.
{ lab_1=pat_1, _ , lab_n=pat_n, <<, ...>>}	:	if v={ lab_1=v_1, _ ,lab_m=v_m } , where $m \geq n$ if "..." is present and $m = n$ otherwise, then pat_i is matched to v_i returning VE_i, for each i; then VE_1+ _ +VE_n is returned.

3.4 Applying a match

Assume environment E. Applying a match "pat_1=>exp_1| _ |pat_n=>exp_n" to value v returns a value or packet as follows. Each pat_i is matched to v in turn, from left to right, until one succeeds returning VE_i; then exp_i is evaluated in E+VE_i.

If none succeeds, then the packet (ematch,()) is returned, where ematch is the standard exception bound by predeclaration to the exception name "match". But matches which may fail are to be detected by the compiler and flagged with a warning; see Section 10(2).

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

3.5 Evaluation of expressions

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

var : the value VE(*var*) is returned.

con : the value VE(*con*) is returned.

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

{ lab_1=exp_1, _ , lab_n=exp_n }:
the exp_i are evaluated in sequence, from left to right, returning v_i respectively; then the record { lab_1=v_1, _ , lab_n=v_n } is returned.

raise *exn* **with** *exp* : *exp* is evaluated, returning value v; then the packet (e,v) is returned, where e = EE(*exn*).

exp **handle** *handler* : *exp* is evaluated; if *exp* returns a value v, then v is returned; if it returns a packet p = (e,v) then the handling rules of the *handler* are scanned from left to right until a rule is found which satisfies one of two conditions:

1. it is of form "*exn* **with** *match*" and e = EE(*exn*), in which case *match* is applied to v;
2. it is of form "? => *exp*'", in which case *exp*' is evaluated.

If no such hrule is found, then p is returned.

let *dec* **in** *exp* **end**	:	*dec* is evaluated, returning E'; then *exp* is evaluated in E+E'.
fn *match*	:	f is returned, where f is the function of v gained by applying *match* to v in environment E.

3.6 Evaluation of value bindings

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

pat = *exp*	:	*exp* is evaluated in E, returning value v; then *pat* is matched to v; if this returns VE', then VE' is returned, and if it fails then the packet (ebind,()) is returned, where ebind is the standard exception bound by predeclaration to the exception name "bind".
vb_1 **and** _ **and** vb_n	:	vb_1,_,vb_n are evaluated in E from left to right, returning VE_1, _ ,VE_n; then VE_1+ _ +VE_n is returned.
rec *vb*	:	*vb* is evaluated in E', returning VE', where E' = (VE+VE',EE). Because the values bound by "**rec** *vb*" must be function values (see 10(4)), E' is well defined by "tying knots" (Landin).

3.7 Evaluation of type and datatype bindings

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

<<*tyvar_seq*>>*tycon* = *con*$_1$<< **of** *ty*$_1$>> | _ | *con*$_n$<<**of** *ty*$_n$>> : the value environment VE' = <(*con*$_1$,v$_1$), _, (*con*$_n$,v$_n$)> is returned, where v$_i$ is either the constant value *con*$_i$ (if "**of** *ty*$_i$" is absent) or else the function which maps v to *con*$_i$(v). Other effects of this datatype binding are dealt with by the compiler or type-checker, not by evaluation.

db$_1$ **and** _ **and** *db*$_n$: *db*$_1$, _ , *db*$_n$ are evaluated from left to right, returning VE$_1$, _ ,VE$_n$; then VE' = VE$_1$+ _ +VE$_n$ is returned.

3.8 Evaluation of exception bindings

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

exn << =*exn'*>> : EE' = <(*exn*,e)> is returned, where

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

eb$_1$ **and** _ **and** *eb*$_n$: *eb*$_1$, _ ,*eb*$_n$ are evaluated in E from left to right, returning EE$_1$, _ ,EE$_n$; then EE' = EE$_1$+_+EE$_n$ is returned.

3.9 Evaluation of declarations

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

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

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

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

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

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

local dec_1 **in** dec_2 **end** :
dec_1 is evaluated, returning E_1, then dec_2 is evaluated in $E+E_1$, returning E_2; then E' = E_2 is returned.

dec_1<<;>> _ dec_n<<;>> :
each dec_i is evaluated in $E+E_1+$ _ $+E_{i-1}$, returning E_i, for $i = 1,2,$ _ $,n$; then E' = (<>,<>)$+E_1+$ _ $+E_n$ 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 "dec_1 ; _ dec_n ;" takes place in the initial presence of the standard top-level environment ENV_0 containing all the standard bindings (see Section 5). For $i > 0$ the top-level environment ENV_i, present after the evaluation of dec_i in the program,is defined recursively as follows: dec_i is evaluated in ENV_{i-1} returning environment E_i, and then $ENV_i = ENV_{i-1}+E_i$.

4 Directives

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

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

infix<<r>> <<d>> id_1 _ id_n

nonfix id_1 _ id_n

where d is a digit. The **infix** and **infixr** directives introduce infix status for each id_i (as a value variable or constructor), and the **nonfix** directive cancels it. The digit d (default 0) determines the precedence, and an infixed identifier associates to the left if introduced by **infix**, to the right if by **infixr**. Different infixed identifiers of equal precedence associate to the left. As indicated in Table 4, the precedence of infixed application is just weaker than that of application.

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

Several standard functions and constructors have infix status (see Table 6) with precedence; these are all left associative except "::".

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

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

5 Standard bindings

The bindings of this section form the standard top-level environment ENV_0.

5.1 Standard type constructors

The bare language provides the record type "$\{lab_1{:}ty_1, _ , lab_n{:}ty_n\}$" for each $n \geq 0$, and the infixed function-type constructor "->". Otherwise, type constructors are postfixed. The following are standard:

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

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

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

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

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

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

5.2 Standard functions and constants

All standard functions and constants are listed in Table 6. There is not a lavish number; we envisage function libraries provided by each implementation, together with the equivalent ML declaration of each function (though the implementation may be more efficient). In time, some such library functions may accrue to the standard; a likely candidate for this is a group of array-handling functions, grouped in a standard declaration of the unary type constructor "array".

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

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

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

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

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

5. Two multi-typed functions are included as quick debugging aids. The function print :*ty*->*ty* is an identity function, which as a side-effect prints its argument exactly as it would be printed at top-level. The printing caused by "print(*exp*)" will depend upon the type ascribed to this *particular* occurrence of *exp*; thus print is not a normal polymorphic function. The function

makestring :*ty*->string is similar, but instead of printing it returns as a string what print would produce on the screen. Since top-level printing is not fully specified, programs using these two functions should not be ported between implementations.

5.3 Standard exceptions

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

The only other predeclared exception names are

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

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

3 div x **handle** div => 10000

will return 10000 when x = 0.

6 Standard Derived Forms

6.1 Expressions and Patterns

See Table 2.

Each derived form may be implemented more efficiently than its equivalent form, but must be precisely equivalent to it semantically. The type-checking of each derived form is also defined by that of its equivalent form. The precedence among all bare and derived forms is shown in Table 4.

The derived type "ty_1 * _ * ty_n" is called an (n-)tuple type, and the values of this type are called (n-)tuples. The associated derived forms of expressions and patterns give exactly the treatment of tuples in the previous ML proposal [5].

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

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

Table 2: Standard Derived Forms: Expressions and Patterns

DERIVED FORM	EQUIVALENT FORM	
Types :		
ty_1 * _ * ty_n	{ #1:ty_1, _ , #n:ty_n }	
Expressions :		
()	{ }	(no space in "()")
(exp_1, _ , exp_n)	{ #1=exp_1, _ , #n=exp_n }	($n \geq 2$)
raise *exn*	**raise** *exn* **with** ()	
case *exp* **of** *match*	(**fn** *match*) (*exp*)	
if *exp* **then** exp_1 **else** exp_2	**case** *exp* **of** true=>exp_1 | false=>exp_2	
exp **orelse** *exp'*	**if** *exp* **then** true **else** *exp'*	
exp **andalso** *exp'*	**if** *exp* **then** *exp'* **else** false	
(exp_1; _ ; exp_n; *exp*)	**case** exp_1 **of** (_) => _ _ _ _ **case** exp_n **of** (_) => *exp*	($n \geq 1$)
let *dec* **in** exp_1; _ ; exp_n **end**	**let** *dec* **in** (exp_1; _ ; exp_n) **end**	
while *exp* **do** *exp'*	**let val rec** f = **fn** () => **if** *exp* **then** (*exp'*; f()) **else** () **in** f() **end**	
[exp_1 , _ , exp_n]	exp_1:: _ ::exp_n::nil	($n \geq 0$)
Handling rules :		
exn => *exp*	*exn* **with** (_) => *exp*	
Patterns :		
()	{ }	(no space in "()")
(pat_1, _ , pat_n)	{ #1=pat_1, _ , #n=pat_n }	($n \geq 2$)
[pat_1, _ , pat_n]	pat_1:: _ ::pat_n::nil	($n \geq 0$)
{ _, id<<:*ty*>><< as *pat*>>, _ }	{ _, id=id<<:*ty*>><< as *pat*>>, _ }	

6.2 Bindings and Declarations

A new syntax class *fb*, of function bindings, is introduced. Function bindings are a convenient form of value binding for function declarations. The equivalent form of each function binding is an ordinary value binding. These new function bindings must be declared by "**fun**", not by "**val**"; however, the bare form of value binding may still be used to declare functions, using **val** or **val rec** (see Table 3).

Table 3: **Standard Derived Forms: Bindings and Declarations**

DERIVED FORM	EQUIVALENT FORM
Function bindings *fb* :	
var $apat_{11}$ _ $apat_{1n}$<<:*ty*>>= exp_1 | _ _ _ _ | *var* $apat_{m1}$ _ $apat_{mn}$<<:*ty*>>= exp_m	*var* = **fn** x_1 => _ **fn** x_n => **case** (x_1, _ , x_n) **of** ($apat_{11}$, _ , $apat_{1n}$) => exp_1<<: *ty*>> | _ _ _ _ | ($apat_{m1}$, _ , $apat_{mn}$) => exp_m <<:*ty*>> (where the x_i are new, and $m, n \geq 1$)
fb_1 **and** _ **and** fb_n	vb_1 **and** _ **and** vb_n (where vb_i is the equivalent of fb_i)
Declarations :	
fun *fb*	**val rec** *vb* (where *vb* is the equivalent of *fb*)
exp	**val** it = *exp*

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

7 References and equality

7.1 References and assignment

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

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

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

The polymorphic contents function "!" is provided, and is equivalent to the function "**fn** (ref x) => x".

7.2 Equality

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

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

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

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

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

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

<<*tyvar_seq*>>*tycon* = *con*$_1$<<**of** *ty*$_1$>> | _ | *con*$_n$<<**of** *ty*$_n$>>

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

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

8 Exceptions

8.1 Discussion

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

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

raise *exn* **with** *exp*

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

exn **with** *match*

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

? => *exp'* .

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

8.2 Derived forms

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

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

exn => *exp'*

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

raise *exn*

since the only possible excepted value is ().

8.3 An example

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

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

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

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

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

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

|| ? => 50000

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

8.4 Some pathological examples

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

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

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

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

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

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

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

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

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

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

9 Type-checking

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

In a match "*pat*$_1$=>*exp*$_1$ | _ | *pat*$_n$=>*exp*$_n$", the types of all *pat*$_i$ must be the same (*ty* say), and if variable var occurs in *pat*$_i$ then all free occurrences of var in *exp*$_i$ must have the same type as its occurrence in *pat*$_i$. In addition, the types of all the *exp*$_i$ must be the same (*ty'* say). Then *ty*->*ty'* is the type of the match. The type of "**fn** *match*" is the type of the match.

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

A type variable is only explicitly bound (in the sense of variable-binding in lambda-calculus) by its occurrence in the *tyvar_seq* on the left hand side of a simple type or datatype binding "<<*tyvar_seq*>>*tycon* = _", and then its scope is the right hand side. (This means for example that bound uses of 'a in both *tb*$_1$ and *tb*$_2$ in the type binding "*tb*$_1$ **and** *tb*$_2$" bear no relation to each other.) Otherwise, repeated occurrences of a type variable may serve to link explicit type constraints. The scope of such a type variable is determined by its first occurrence (ignoring all occurrences which lie within scopes already thus determined). If this first occurrence is in an **exception** declaration, then it has the same scope as the declared exception(s); otherwise, its scope is the smallest **val** (or **fun**) declaration in which it lies. For example, consider

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

Here the scope of both 'a and 'b is the whole **fun** declaration, while the scope of 'c is just the **val** declaration. Note that this allows "Id" to be used polymorphically after its declaration. Moreover, type-checking must not further constrain a type variable within its scope. Thus for example the declaration

"**fun** Apply(f:'a->'b)(x:'a):'b = x"

– in which "x" has been written in error in place of "f(x)" – will be faulted since it requires 'a and 'b to be equated.

A simple datatype binding "<<*tyvar_seq*>>*tycon* = _ " is *generative*, since a new unique type constructor (denoted by *tycon*) is created by each textual occurrence of such a binding. A simple type binding "<<*tyvar_seq*>>*tycon* = *ty*", on the other hand, is *non-generative*; to take an example, the type binding

" 'a couple = 'a * 'a " merely allows the type expression "*ty* couple" to abbreviate "*ty* * *ty*" (for any *ty*) within its scope. There is no semantic significance in abbreviation; in the Core language it is purely for brevity, though in Modules non-generative type-bindings are essential in matching Signatures. However, the type-checker should take some advantage of non-local type abbreviations in reporting types at top-level; in doing this, it may need to choose sensibly between different possible abbreviations for the same type.

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

10 Syntactic restrictions

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

2. In a match "pat_1=>exp_1 | _ | pat_n=> exp_n", the pattern sequence pat_1, _ , pat_n should be *irredundant* and *exhaustive*. That is, each pat_j must match some value (of the right type) which is not matched by pat_i for any $i < j$, and every value (of the right type) must be matched by some pat_i. The compiler must give warning on violation of this restriction, but should still compile the match. Thus the "match" exception (see Section 3.4) will only be raised for a match which has been flagged by the compiler. The restriction is inherited by derived forms; in particular, this means that in the function binding "*var* $apat_1$ _ $apat_n$ <<:*ty*>> = *exp*" (consisting of one clause only), each separate $apat_i$ should be exhaustive by itself.

3. For each value binding "*pat* = *exp*" the compiler must issue a report (but still compile) if either *pat* is not exhaustive or *pat* contains no variable. This will (on both counts) detect a mistaken declaration like "**val** nil = *exp*" in which the user expects to declare a new variable nil (whereas the language dictates that nil is here a constant pattern, so no variable gets declared). However, these warnings should not be given when the binding is a component of a top-level declaration **val** *vb* ; e.g. "**val** x::l = exp_1 **and** y = exp_2" is not faulted by the compiler at top level, but may of course generate a "bind" exception (see Section 3.6).

4. For each value binding "*pat* = *exp*" within **rec**, *exp* must be of the form

"**fn** *match*". The derived form of value binding given in Section 6.2 necessarily obeys this restriction.

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

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

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

8. If, within the scope of a type constructor *tycon*, a type binding *tb* or datatype binding *db* binds (simultaneously) one or more type constructors $tycon_1, _, tycon_n$ then: (a) if the identifiers $tycon_i$ are all distinct from *tycon*, then their value constructors (if any) must also have identifiers distinct from those (if any) of *tycon*; (b) if any $tycon_i$ is the same identifier as *tycon*, then any value constructor of *tycon* may be re-bound as a value constructor for one of $tycon_1, _, tycon_n$, but is otherwise considered unbound (as a variable or value constructor) within the scope of *tb* or *db* , unless it is bound again therein. This constraint ensures that the scope of a type constructor is identical with the scopes of its associated value constructors, except that in an **abstype** declaration the scope of the value constructors is restricted to the **with** part.

11 Relation between the Core language and Modules

The sister report [6] on ML Modules describes how ML declarations are grouped together into *Structures* which can be compiled separately. Structures, and the *Functors* which generate them, may not be declared locally within ML programs, but only at top-level or local to other Structures and Functors; this means that the Core language is largely unaffected by their nature.

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

$$\mathrm{id}_1._.\mathrm{id}_n.\mathrm{id} \quad (n \geq 1)$$

where $\mathrm{id}_1, _, \mathrm{id}_n$ are Structure names, each id_i is the name of a component Structure of id_{i-1} for $1 < i \leq n$, and id is either a type constructor, a value constructor, a value variable, an exception name or a Structure name declared as a component of Structure id_n. Thus the syntax classes *tycon*, *con*, *var* and *exn* are extended to include qualified names. Further, the declaration

$$\textbf{open}\ \mathrm{id}_1._.\mathrm{id}_n \quad (n \geq 1)$$

(where $\mathrm{id}_1, _, \mathrm{id}_n$ are as above) allows the components of the Structure $\mathrm{id}_1._.\mathrm{id}_n$ to be named without qualification in the scope of the declaration.

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

12 Conclusion

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

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

Table 4: **Syntax : Expressions and Patterns**

(See Section 2.8 for conventions and remarks)

aexp ::=		
<<op>>*var*		(variable)
<<op>>*con*		(constructor)
{ *lab*$_1$=*exp*$_1$, _ , *lab*$_n$=*exp*$_n$ }		(record, $n \geq 0$)
()		(0-tuple)
(*exp*$_1$, _ , *exp*$_n$)		(n-tuple, $n \geq 2$)
[*exp*$_1$, _ , *exp*$_n$]		(list, $n \geq 0$)
(*exp*$_1$; _ ; *exp*$_n$)		(sequence, $n \geq 1$)
exp ::=		
aexp		(atomic)
exp aexp	L	(application)
exp id *exp'*		(infixed application)
exp : *ty*	L	(constraint)
exp **andalso** *exp'*		(conjunction)
exp **orelse** *exp'*		(disjunction)
exp **handle** *handler*	R	(handle exception)
raise *exn* <<**with** *exp*>>		(raise exception)
if *exp* **then** *exp*$_1$ **else** *exp*$_2$		(conditional)
while *exp* **do** *exp'*		(iteration)
let *dec* **in** *exp*$_1$; _ ; *exp*$_n$ **end**		(local declaration, $n \geq 1$)
case *exp* **of** *match*		(case expression)
fn *match*		(function)

match ::=
rule$_1$ | _ | *rule*$_n$ ($n \geq 1$)

handler ::=
hrule$_1$ || _ || *hrule*$_n$ ($n \geq 1$)

rule ::=
pat => *exp*

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

apat ::=		
_		(wildcard)
<<op>>*var*		(variable)
con		(constant)
{ *lab*$_1$=*pat*$_1$, _ , *lab*$_n$=*pat*$_n$ <<, . . .>>}		(record,$n \geq 0$) [a]
()		(0-tuple)
(*pat*$_1$, _ , *pat*$_n$)		(tuple, $n \geq 2$)
[*pat*$_1$, _ , *pat*$_n$]		(list, $n \geq 0$)
(*pat*)		
pat ::=		
apat		(atomic)
<<op>>*con apat*	L	(construction)
pat con pat'		(infixed construction)
pat : *ty*	L	(constraint)
<<op>>*var*<<:*ty*>> **as** *pat*	R	(layered)

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

Table 5: Syntax : Types, Bindings, Declarations

(See Section 2.8 for conventions)

ty ::=		
tyvar		(type variable)
<<*ty_seq*>>*tycon*		(type construction)
{ lab_1:ty_1, _ , lab_n:ty_n }		(record type, $n \geq 0$)
ty_1 * _ * ty_n		(tuple type, $n \geq 2$)
ty_1 -> ty_2	R	(function type)
(*ty*)		
vb ::=		
pat = *exp*		(simple)
vb_1 **and** _ **and** vb_n		(multiple, $n \geq 2$)
rec *vb*		(recursive)
fb ::=		
<<**op**>>*var* $apat_{11}$ _ $apat_{1n}$<<:*ty*>> = exp_1		(clausal function, $m, n \geq 1$) [a]
| _ _		
_ _		
| <<**op**>>*var* $apat_{m1}$ _ $apat_{mn}$<<:*ty*>> = exp_m		
fb_1 **and** _ **and** fb_n		(multiple, $n \geq 2$)
tb ::=		
<<*tyvar_seq*>>*tycon* = *ty*		(simple)
tb_1 **and** _ **and** tb_n		(multiple, $n \geq 2$)
db ::=		
<<*tyvar_seq*>>*tycon* = *constrs*		(simple)
db_1 **and** _ **and** db_n		(multiple, $n \geq 2$)
constrs ::=		
<<**op**>>con_1<<**of** ty_1>> | _		
| <<**op**>>con_n<<**of** ty_n>>		($n \geq 1$)
eb ::=		
exn<<:*ty*>><< =*exn'*>>		(simple)
eb_1 **and** _ **and** eb_n		(multiple, $n \geq 2$)
dec ::=		
val *vb*		(value declaration)
fun *fb*		(function declaration)
type *tb*		(type declaration)
datatype *db*		(datatype declaration)
abstype *db* **with** *dec* **end**		(abstract type declaration)
exception *eb*		(exception declaration)
local *dec* **in** *dec'* **end**		(local declaration)
exp		(top-level only)
dir		(directive)
dec_1<<;>> _ dec_n<<;>>		(declaration sequence, $n \geq 0$)
dir ::=		
infix<<r>> <<d>> id_1_ id_n		(declare infix, $0 \leq d \leq 9$)
nonfix id_1 _ id_n		(cancel infix)

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

Table 6: Predeclared Variables and Constructors

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

nonfix

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

true,false     : bool
not            : bool -> bool

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

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

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

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

infix

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

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

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

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

Special constants: as in Section 2.3.
Notes:

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

   nil true false ref :: and all special constants.

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

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

References

[1] Michael Gordon, Robin Milner, and Christopher Wadsworth, *Edinburgh LCF*, Spring–Verlag Lecture Notes in Computer Science, vol. 78, 1979.

[2] R. Burstall, D. MacQueen, and D. Sannella, *HOPE: An Experimental Applicative Language*, Edinburgh University Internal Report CSR-62-80, 1980.

[3] Luca Cardelli, *ML under UNIX*, AT&T Bell Laboratories, 1984.

[4] Robin Milner, *A Proposal for Standard ML*, Edinburgh University Internal Report CSR-157-83, 1983.

[5] Robin Milner, *The Standard ML Core Language*, Edinburgh University Internal Report CSR-168-84, 1984.

[6] David MacQueen, *Modules for Standard ML*, AT&T Bell Laboratories, May, 1985.

[7] Robert Harper, David MacQueen, and Robin Milner, *Standard ML*, Edinburgh University Internal Report ECS–LFCS–86-2, March, 1986.

Standard ML Input/Output

Robert W. Harper

June 10, 1985

1 Introduction

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

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

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

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

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

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

2 Basic I/O Primitives

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

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

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

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

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

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

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

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

Figure 1: Basic I/O Primitives

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

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

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

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

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

Figure 2: Extended I/O Primitives

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

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

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

3 Extended I/O Primitives

In addition to the basic I/O primitives, provision is made for a some extensions that are likely to be provided by many implementations. The structure `ExtendedIO` consists of an extensible set of operations that are commonly used but are either too complex to be considered primitive or to be implementable on all hosts. Its signature appears in Figure 2.

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

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

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

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

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

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

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

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

Acknowledgements

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

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

References

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

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

PART III

Modules for Standard ML

David MacQueen

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This document describes and motivates the design of a set of constructs to support generic modular programming in Standard ML. The basic constructs are *structures*, which are essentially packaged environments, *signatures*, which are the type specifications of structures, and *functors*, which are mappings from structures to structures. The treatment of inheritance and sharing relationships between structures is the most important and characteristic feature of the design.

October 3, 1985

Contents

1. Introduction

1.1. Language constructs for modularity
1.2. A note on terminology
1.3. Overview

2. The Language

2.1. Structures
2.1.1. Structure expressions
2.1.2. Structure declarations
2.1.3. Accessing structure components
2.1.4. Evaluating structure expressions
2.1.5. Structure equivalence
2.2. Signatures
2.2.1. Type specifications
2.2.2. Signature expressions and declarations
2.2.3. Inferred signatures
2.3. Inheritance
2.3.1. Inherited types and signature closure
2.3.2. Inherited exceptions
2.3.3. Inherited objects
2.3.4. Discussion
2.4. Functors
2.4.1. Functor declarations
2.4.2. Multiple instances
2.5. Type propagation
2.6. Coherence and sharing constraints
2.7. Type abstraction
2.8. Signature and type checking
2.8.1. Signature matching
2.8.2. Typing structure components
2.8.3. Checking sharing constraints
2.9. Miscellaneous notes
2.9.1. Restrictions on module and signature declarations
2.9.2. Sharing constraints in signatures
2.9.3. Recursive modules
2.9.4. Views

3. Conclusions and Future Work

Appendix A: Syntax

Modules for Standard ML

David MacQueen

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

Writing large programs in ML, as in any language, poses difficult problems of organization. Many modern programming languages contain constructs designed to help organize large systems. These constructs, variously known as modules, packages, or clusters, make it possible to partition a large software system into reasonably-sized components and provide secure methods for combining these components to form complete programs. They often support separate compilation of program components and make it possible to assemble libraries of shareable, generic components. They also typically support some form of data abstraction.[1] The use of such constructs to impose structure on large programs is often called "programming in the large" in contrast to "programming in the small", which is concerned with the detailed implementation of algorithms in terms of data structures and control constructs. The purpose of this proposal is to demonstrate that there are natural ways to extend the core ML language to support modular programming.

Besides the practical concerns of programming in the large, there are theoretical problems that motivate us to develop a notion of module for ML. One of these is the need to extend the power of the polymorphic type system by introducing parameterization with respect to types with associated operations. Another is the desire for a more fundamental approach to type abstraction. Finally, there is the problem of treating environments as first-class entities that can be typed, named, and processed.

Fortunately, there is a conceptual framework for dealing with these theoretical problems which also gracefully supports the requirements of programming in the large. This framework is inspired by the theory of dependent types as developed in intuitionistic type theory [ML82, NS84] and the closely related ideas of second-order lambda calculus [REY85, MP85, CW85], both of which can be viewed as generalizations of the basic polymorphic type system of ML. The Pebble language of Burstall and Lampson [BL84, BUR84] has strong parallels with this proposal, but there are some important differences concerning the treatment of types and the critical issue of inheritance and sharing.

1.1. Language constructs for modularity

There are at least two starting points for developing a theory of modularity. One can start from type theory and explore notions like existential types and dependent product types, or one can analyze the large-scale structure of programs more directly by developing a calculus of environments. These approaches are complementary; indeed, notions like existential and dependent types arise naturally when one attempts to provide a type system for describing environments and their relations. Here we begin by thinking about modularity in terms of the creation and manipulation of environments.

---

[1] Here *data abstraction* simply means that one can associate some operations with a given type and then hide the structure of that type so that it can only be used via the designated operations.

In its simplest form, a module is (syntactically) just a collection of declarations viewed as a unit, or (semantically) the environment defined by those declarations. Complications arise when we try to isolate the declarations from their context and then abstract with respect to that context. We want to isolate the declarations in order to make explicit the interfaces between components of a program and to limit and control these interfaces for the sake of information hiding and separate compilation. We want to abstract with respect to the context to achieve flexible, generic program units (parametric modules). Abstraction and its complement, application, as in typed lambda calculus, provide the glue with which we will build programs from their components.

The three central notions of this proposal are *structures*, *signatures*, and *functors*. A *structure* is simply an environment, defined most directly by an encapsulated declaration (think of "structure" as short for "environment structure" if you like). A *signature* represents the type information associated with a structure and serves as an interface specification for the structure; roughly speaking, it indicates what names are bound in the structure, and what their types are. A *functor* is a function that maps structures to structures, and is defined by functional abstraction of a structure expression with respect to structure names that occur free in it. It expresses in a uniform way how one environment is defined in terms of others. Structures and functors are collectively referred to as *modules*.

There is a naive analogy between values, types, and functions on the one hand, and structures, signatures, and functors on the other (see BUR84). There is also an important distinction in kind between structures and values: structure expressions are "big" expressions denoting environments in which "little" expressions denoting values can be evaluated. Furthermore, the environment represented by a structure may be a mixed environment containing type bindings and exception bindings as well as simple value bindings. The question of whether to formally distinguish structures from ordinary values, and signatures from ordinary types, is therefore a critical issue in this design.

Two related systems, SOL [MP85] and Pebble [BL84], both treat their respective analogues of our structures (called *data algebras* and *bindings*, respectively) as just another kind of value, and the corresponding types (*existential types* and *declarations*, respectively) as ordinary types. But they accomplish this in quite different ways: SOL restricts the use of data algebras in such a way that they cannot carry any type information, while Pebble abandons the distinction between values and types, treating types as a kind of value.

In this design, on the other hand, the distinction between values and types is strictly maintained, but structures are allowed to carry type information. In fact, a structure that contains a type binding together with bindings of related functions can be thought of as an *interpreted type*, *i.e.* a type whose behavior is determined by the associated functions rather than by its structure alone. Such structures are hybrid entities inhabiting an area between the domain of values and the domain of simple types. Thus structures are not ordinary values, nor are signatures ordinary types. The extended ML described here is a stratified language, with values and types on one level and structures, functors, and signatures on another. The two levels are related, of course, but they do not mix; functions cannot be applied to structures nor can functors be applied to values. We resist the temptation to merge these two levels, as is done in Pebble, because if we did we would forgo the possibility of *static* type checking.

Another crucial issue for the design of modules is the treatment of dependency. It is clearly useful to define new structures in terms of old ones, and such definitions introduce dependencies between structures. In certain circumstances, such as when a type from the old structure appears in the signature of the new structure, the new structure cannot be

used independently of the old one, and we say that the dependence is *explicit*. Explicit dependencies must be taken into account when we abstract with respect to a structure; in effect, abstraction with respect to a structure often entails abstraction with respect to all other structures on which it explicitly depends. The essence of a parametric module design is the method used to manage the interaction of dependency and abstraction. This is another issue on which this proposal differs fundamentally from SOL, which does not allow explicit dependencies, and Pebble (see [MAC86] for a fuller discussion of these issues and an explanation of the foundations of this design).

1.2. A note on terminology

The term "structure" is meant to be suggestive of the universal algebra notion of a mathematical structure consisting of some carrier sets and a collection of operations over the carriers. "Signature" is a conventional mathematical term for the specification of the type structure of such "structures". "Functor" is a word borrowed from category theory, where it designates a mapping between categories, which for our purposes can be considered collections of mathematical structures like partially ordered sets, groups, etc. Strictly speaking, functors operate not only on the structures themselves, but also on structure-preserving maps (or "morphisms") between structures. Our use of the term functor does not include this morphism mapping aspect, since there is no analogue of the notion of morphism in our theory of modules.

In this treatment, the concept of a structure is taken as primary, with signatures and functors playing secondary roles. However in the informal presentation of Pebble in [BUR84] signatures are called "interfaces" and structures "implementations" (of interfaces), suggesting that interfaces have the central role. Clearly the terminology reflects a judgement about the relative importance of the basic notions.

As indicated above, the terminology in SOL is "data algebra" for what corresponds to a (certain kind of) structure, and "existential type" for signature. The closely related language of [CW85] uses "package" and "existential type" respectively. "Existential type" and "data algebra" are too specialized to apply to structures and signatures as presented here, but "package" seems a reasonable alternative to "structure".

1.3. Overview

The purpose of this document is to present extensions to the ML Core Language [MIL85] for the support of modular, parametric programming. The style is descriptive and tutorial, and the document is not meant to serve as a formal reference manual for the new language facilities. A number of details are left to be resolved later when experience has been gained with preliminary implementations of the design.

Section 2 describes the new constructs and attempts to provide intuitive motivations for them. It also covers the important topics of *inheritance* (§2.3), *type propagation* (§2.5), *sharing* (§2.6), and a new treatment of type abstraction based on structure abstraction (§2.7). Section 3 indicates areas where further development is required and suggests possible extensions and variations. Appendix A provides an informal syntax for the new language constructs. Some familiarity with Standard ML is assumed.

2. The Language

In this section we introduce the proposed extensions to Standard ML and informally describe their semantics and typing rules. An informal syntax in the style used in [MIL85] is provided in Appendix A.

2.1. Structures

As mentioned above, a structure is essentially a heterogeneous environment. It consists of a set of bindings, possibly including type bindings, value bindings, and exception bindings. We also permit the definition of a structure to include structure declarations, yielding a notion of component substructure. In order to limit the ways in which a structure can depend on its context, the definition of a structure can only refer to other structures and functors and pervasive primitive types and values.

2.1.1. *Structure expressions*

There are three forms of structure expressions: encapsulated declarations, functor applications, and structure names (either simple or qualified — see §2.1.3). The basic form of structure expression is the encapsulated declaration, which is simply an ordinary ML declaration surrounded by the keywords "`struct ... end`." For example, the following structure expression represents an implementation of stacks:

```
struct
  datatype 'a stack = nilstack | push of 'a * 'a stack
  exception pop: unit and top: unit
  fun empty(nilstack) = true
    | empty _ = false
  and pop(push(_,s)) = s
    | pop _ = raise pop
  and top(push(x,_)) = x
    | top _ = raise top
end
```

Certain restrictions must be imposed on the declarations that make up the body of an encapsulated declaration. These restrictions are embodied in the following rules.

(1) *Structure closure.* In order to isolate the interface between a structure and its context, a structure expression (in particular an encapsulated declaration) is not allowed to contain global references to types, values or exceptions, except for pervasive primitives of the language like `int`, `nil`, etc. It can, however, contain global references to other structures, signatures, and functors, including qualified names referring to arbitrary components of other structures. The closure rule for signatures given in §2.3.1 will further restrict how such global references may be used.[2]

(2) *No redeclaration.* Type and substructure names should be declared only once at top level within a given encapsulated declaration. Furthermore, the same name should not be used for both a type component and a substructure within a given structure. The first restriction is meant to prevent "orphans" (e.g. value components whose types are not expressible because certain types have been hidden by redeclaration),

[2] Schemes for separate compilation may add more stringent closure requirements for structures (and functors) to be separately compiled. For instance, they might require *no* global references to structures or functors.

and thereby simplify the problem of assigning signatures to structures. The second restriction prevents ambiguity in sharing constraints (§2.6).

(3) *Monomorphic exceptions.* Exception components of structures must have monomorphic types because, like exceptions declared at top level, their scope is of indefinite extent.

Since functors are structure-valued mappings, structures may also be denoted by functor applications such as `F(`*str1*`,`*str2*`)`, where `F` is the name of a functor and *str1* and *str2* are structure expressions. The third form of structure expression consists of a name currently bound to a structure, *i.e.* a structure variable. These include both simple names and qualified names referring to a structure component of a named structure (§2.1.3), *e.g.* `EListEq.Eq` in §2.4. Any of these forms may be used wherever a structure expression is allowed, *i.e.* the right hand side of a structure declaration or an argument of a functor application.

2.1.2. *Structure declarations*

A structure declaration binds a structure to a name, and is written in the usual ML style with the declaration keyword "`structure`", *i.e.*

`structure` *name* = *structure expression*

The structure presented presented above, for instance, could be named "`Stack`" by the following declaration:

```
structure Stack =
    struct
      datatype 'a stack = nilstack | push of 'a * 'a stack
      ...
    end
```

As with other sorts of declarations, one can combine several simultaneous structure bindings separated by "and" in a single structure declaration, as in

```
structure A = struct ... end
      and B = struct ... end
      and C = struct ... end
```

Mutually recursive structure definitions are not allowed, however, for reasons discussed in §2.9.2.

An encapsulated declaration defining a structure may itself include structure declarations, and we call the resulting components *substructures*. Substructures are used to make a structure self-contained; see the discussion of inheritance in §2.3. Structure declarations can appear only at top level, or immediately (*i.e.* at "top level") within an encapsulated declaration. In particular, structure declarations cannot appear inside *let* expressions or *local* declarations. See §2.9.1 for further discussion.

2.1.3. *Accessing structure components*

The bindings making up a structure can be thought of as defining named *components* of the structure, as in a record. To refer to such components we use qualified names, which are formed in the conventional way by appending a period followed by a component name to the name of the structure. For instance, `Stack.empty` refers to the function `empty` defined in the structure `Stack`. If the qualified name designates a substructure of a structure, then it too can be qualified; *e.g.* `A.B.x` denotes the component named `x` of the substructure named `B` of a structure named `A`. We can view a structure

declaration as simultaneously defining the structure name and all the qualified names derived from it. Qualifiers can be attached only to names; they do not apply to other forms of structure expressions. Qualified names are treated as single lexical units; the period is not an infix operator.

Direct access to the bindings of a structure is provided by the "open" declaration, which is analogous to the "with" clause of Pascal. For example, in the scope (determined in the usual way) of the declaration

```
open Stack
```

the names `stack`, `empty`, `pop`, etc. refer to the corresponding components of the `Stack` structure. It is as though the body of the structure definition had been inserted in the program at that point, except that the bindings are not recreated, but are instead simply "borrowed" from the opened structure. *Open* declarations follow the usual rules for visibility, so that if `A` and `B` are two structures containing a binding of `x` (of the same flavor, of course), then after opening both `A` and `B` with the declaration "`open A; open B`", the unqualified identifier `x` will be equivalent to `B.x`. The `x` component of `A` can still be referred to as `A.x`, unless `B` also contains a structure binding of `A`. Note that an *open* declaration can give rise to the same sorts of visibility violations (e.g. redefining a constructor) as would have resulted from the textual substitution of the structure's body, and such violations should be reported by the compiler.

When an infix identifier is used as a qualifier, the qualified name does not inherit its infix status. Thus if "+" is declared to be infix in a structure `A`, `A.+` is not an infix identifier. However, when an identifier is made visible by opening a structure, it retains its infix status, if any.

Several structures can be opened in a single declaration. Thus "`open A B C`" is equivalent to "`open A; open B; open C`."

2.1.4. *Evaluating structure expressions*

The evaluation of a structure expression *str* depends on its form, and assumes a current structure environment *SE* that binds structures and functors to names. Informally, evaluation proceeds as follows:

(1) If *str* is a simple name, then its binding in *SE* is returned. If it is a qualified name, then it is used as an access path starting with *SE* and the designated substructure is returned.

(2) If *str* is an encapsulated declaration, then the body declarations are evaluated relative to SE and the *pervasive* value, exception, and type environments of ML (that is, the environments binding the built-in primitives of the language).[3] The resulting environment is packaged as a structure and returned.

(3) If *str* is a functor application, say $str = M(str_1, \ldots, str_n)$, the parameter structures are evaluated relative to *SE* yielding structures $S_1, \cdots, S_n$ and then the "body" of the definition of *M*, which is a structure expression, is evaluated in a structure environment derived from *SE* by binding $S_1, \cdots, S_n$ to the corresponding formal parameter variables of *M*. In other words, functor applications are evaluated in a conventional call-by-value fashion.

[3] Note that the rules for evaluating ordinary type and value declarations must be modified to use a structure environment for the interpretation of qualified names appearing in type and value expressions.

To evaluate a simple structure declaration one evaluates the defining structure expression in the current structure environment *SE* and returns the binding of the name on the left hand side to the resulting structure. Simultaneous structure declarations are evaluated in sequence, with all structure expressions evaluated in *SE*, and the set of bindings is returned. If evaluation of a structure expression causes an (untrapped) exception, then the declaration has no effect.

2.1.5. *Structure equivalence*

For certain purposes, such as checking sharing constraints (§2.6) we must be able to determine whether two (references to) structures are considered equal or equivalent. Here structures are treated somewhat like datatypes: each evaluation of an encapsulated declaration or functor application creates a distinct new structure, and all references to this structure are considered equal. Thus after the following declarations

```
structure S1 = struct ... end

structure S2 = S1
```

the names `S1` and `S2` refer to the same structure and are "equal." On the other hand, after

```
structure S1 = struct ... end

structure S2 = struct ... end
```

`S1` and `S2` are not equal, even if the right-hand-sides of the two definitions are identical.

2.2. Signatures

For every declaration (in context), the ML type inference mechanism will yield a corresponding type specification, or *signature*. For instance, in the case of the stack structure the signature of the body declaration is

```
datatype 'a stack = nilstack | push of 'a * 'a stack
exception pop: unit and top: unit
val empty: 'a stack -> bool
and pop: 'a stack -> 'a stack
and top: 'a stack -> 'a
```

The signature captures the essence of the declaration, in the sense that it contains all the information that is required to compile any code lying in the scope of that declaration.

2.2.1. *Type specifications*

Note that in this example the *specification* of the type constructor `stack` is identical to its *declaration*. This is because the interface information required to use a datatype directly, *e.g.* to do pattern matching over the type, consists of the name and arity of the type constructor and the names and types of the data constructors, and thus essentially reiterates all the information in the declaration. It is important to realize, however, that while the declaration creates a unique type, the corresponding specification can be satisfied by any number of distinct, though structurally isomorphic, types.[4] A similar comment applies to exception specifications, which also may be identical to the corresponding declaration.

[4] This redundancy between declarations and specifications of datatypes is not really related to the generative nature of the datatype declarations. Even if unions were purely structural, or if the type

An alternate type specification for `stack` would omit the data constructors and leave the internal structure of the type unspecified:

```
type 'a stack . . .
```

A client program using this weaker signature (*e.g.* a functor having a parameter of this signature) would not be able to operate directly on stack values using pattern matching because the constructors for `stack` would not be visible to it, but it would be able to use the explicitly provided operations on stacks, which we think of as the *interpreting* functions for the type. Later we shall see that such partial specifications in conjunction with structure parameters yield a natural form of type abstraction.

2.2.2. *Signature expressions and declarations*

Just as we need type expressions to specify constraints on variable bindings and function parameters, we need signature expressions to explicitly constrain structure bindings and structure parameters of functors. We represent a signature either by an explicit *signature expression* such as

```
sig
  type elem
  val eq: elem * elem -> bool
end
```

or by a name bound to a signature by a *signature declaration* such as

```
signature EQ =
  sig
    type elem
    val eq: elem * elem -> bool
  end
```

We can use a signature as a constraint when declaring a structure, as in

```
structure E: EQ = struct ... end
```

As usual, such constraints provide a measure of redundancy because the declarations in the body of the structure expression must agree with the explicitly specified signature. However, the signature matching rules (§2.8.1) allow a kind of forgetful coercion from a structure to a restriction or subset of that structure, so the definiens may be richer than the specified signature, in the sense of having extra components. In this case, the signature constraint serves an editing function, allowing us to discard some components of the defining structure. When used in this way, the signature constraint acts as an "export" specification.

Another critical use of signature constraints is to specify the "type" of parameters of functors, where they provide information necessary for the type checking and compilation of the functor body.

The order of specifications in a signature is significant, in the sense that a structure will match a signature only if it binds its component names in the same order as they are specified in the signature. Furthermore, any well-formed signature must obey the

---

declaration was a simple type identity like "`type t = (int*int->bool)*string`", it would still be tempting in certain circumstances to let the definition of the type reside in the signature, so that it could be shared by all clients *and* implementors of that signature. This idea of putting shared type definitions into an interface is used in C/Mesa [MMS79].

following rules:

(1) *Unique specification.* Within each category (structure, type, value, and exception) a name can be specified at most once. A name cannot be specified as both a structure and a type.

(2) *Specification before use.* A name must be specified (as a structure or type) before it can appear in other specifications, except in the case of mutually recursive datatype specifications, where the names of the recursive types may appear in one another's constructor specifications.

(3) *Signature closure.* Signatures may not contain free names other than pervasive primitives (*e.g.* `int` and `bool`) and the names of previously declared signatures.

(4) *Monomorphic exceptions.* Exception specifications must have monomorphic type constraints.

The *unique specification* rule reflects the fact that a signature specifies the type of an environment, and an environment can have at most one binding for a name in each category. The *specification before use* rule reflects the usual requirement that names must be defined before they are used, except in the case of mutually recursive definitions. The *signature closure* rule is a way of insuring that structures will have self-contained, independent interfaces. It is discussed further in §2.3.1. The *monomorphic exception* rule follows from the corresponding rule for structure expressions.

2.2.3. *Inferred signatures*

It is convenient to omit the explicit signature constraint from a structure declaration under certain circumstances, *e.g.* when the structure is used in a very limited context and its signature is "obvious". In many such cases, a signature for the structure may be inferred from its definition using the specifications generated by the ML type inference mechanism. However, for various reasons a structure expression (an encapsulated declaration in particular) may not have a unique signature. For example, the structure expression

```
struct
  type t = int
  fun f x = x+1        { :int -> int }
end
```

would match any of the following signatures, depending on which occurrences of `int` are "abbreviated" to `t`:

```
sig type t  val f: int->int end
sig type t  val f: t->int end
sig type t  val f: int->t end
sig type t  val f: t->t end
```

Given that `t` denotes `int`, the four different specifications for `f` express the same typing, so the inferred type of `f` does not distinguish between them.

On the other hand, given

```
structure A =
  struct
    datatype 'a D = d of 'a
  end
```

the structure expression

```
struct
  type t = int D
  val f(d(x)) = d(x+1)
end
```

has the following potential signatures, only the last of which satisfies the signature closure rule (the others contain free references to the structure A):

```
sig type t  val f: A.D->A.D end
sig type t  val f: t->A.D end
sig type t  val f: A.D->t end
sig type t  val f: t->t end
```

In general, the algorithm for inferring a well-formed signature for a structure expression (or determining that it has none) is rather complex, and may not be fully implemented by a given compiler. Furthermore, there will not always be a unique well-formed signature. Hence it is advisable to provide an explicit signature when in doubt.

2.3. Inheritance

It will of course be commonplace to define new structures in terms of old ones. In such cases the new structure will be called the *derived* structure and the older structures on which it is based will be know as its *antecedents*.[5] Antecedent structures are "imported" into the definition of the derived structure simply by being referred to in that definition. There is no special "import" declaration.

Clearly a derived structure "depends" on its antecedents, but the nature of that dependence varies according to how the antecedents are used in the definition. The principal distinction is between antecedents which are relevant only to the *implementation* of the derived structure, and those which are relevant to its *use* as well.

As an example where the antecedent is relevant only to the implementation, imagine a symbol table structure SymTab having the following signature

```
signature SYMTAB =
  sig
    type 'a table
    val nilst: 'a table
    and extend: 'a table -> 'a table
    and contract: 'a table -> 'a table
    and putst: string * 'a * 'a table -> 'a table
    and access: string * 'a table -> 'a
    exception access: string
    and putst: string
  end
```

Suppose that SymTab implements its type constructor table in terms of stacks as defined in the Stack structure of §2.1.2. The signature SYMTAB does not mention stacks, and we can assume that it is irrelevant to the user of SymTab whether stacks of any sort were used in its implementation. In this case, the fact that SymTab depends on

[5] An *antecedent* structure is simply one whose name appears in the definition of the derived structure. The relation of being an antecedent is not transitive, but we might use the term *ancestor* for its transitive closure. Note that the antecedent relation is not the same as the substructure relation.

its antecedent `Stack` can remain implicit because it is only relevant to the internal makeup of `SymTab`.

The principal reasons why an antecedent structure might be relevant to the *use* of a derived structure are:

- The derived structure uses types inherited from an antecedent in an "external" or "visible" way, i.e. in such a way that the inherited types appear in the signature of the derived structure (§2.3.1).
- Exceptions inherited from the antecedent structure are raised within functions of the derived structure (§2.3.2).
- An antecedent structure with state is used as a medium of communication with the derived structure (*i.e.* structures as *objects*) (§2.3.3).

We say that a derived structure *depends explicitly* on an antecedent when the antecedent is relevant to the use of the derived structure for any of these reasons, and in this case we call the antecedent structure a *prerequisite* of the derived structure.

The manner in which we deal with explicit dependence is one of the most characteristic features of this language proposal. The dependency may be (and in certain circumstances, *must* be) acknowledged by incorporating prerequisite structures in the derived structure as substructures. This makes structures self-contained, in the sense that whenever we have access to a structure, we also have access, through its substructures, to any antecedents that can help us make use of the structure. Achieving this kind of self-sufficiency was the basic motivation for introducing the notion of substructure in the first place. Technically, the combination of the derived structure with its antecedents corresponds to forming an element of a dependent product; see [MAC86] for further details.

2.3.1. *Inherited types and signature closure*

Here we consider the first form of essential dependency: the visible inheritance of types. We will see that one side effect of the inclusion of prerequisite structures is that signatures become closed, context-independent expressions.

We will investigate the phenomenon of type inheritance in terms of a fairly realistic example. First we define a structure `Exp` that implements a data type of expressions with associated operations.

```
signature EXP =
  sig
    datatype id = Id of string
    datatype exp = Var of id
                 | App of id * (exp list)
    val oper: exp -> id
    and ntharg: exp * int -> exp
    and varsof: exp -> id list
    ...
  end

structure Exp: EXP =
  struct
    datatype id = Id of string
    datatype exp = Var of id
                 | App of id * (exp list)
```

```
    fun oper(App(x,_)) = x
    and ntharg(App(_,args), n) = nth(args,n)
    and varsof(Var i) = i
      | varsof(App(_,args)) = concat(map(varsof, args))
    ...
  end
```

Now suppose we want to define another structure Subst that implements substitutions for these expressions. We might (improperly) define this structure as follows:

```
structure Subst: SUBST =
  struct
    type subst = (Exp.id * Exp.exp) list
    fun lookup(id,nil) = Exp.Var id
      | lookup(id,(id',e)::l) = if id=id' then e else lookup(id,l)
    fun substitute s (Exp.Var id) = lookup(id, s)
      | substitute s (Exp.App(id,args)) =
              Exp.App(id, map(substitute s, args))
  end
```

where

```
signature SUBST =
  sig
    type subst
      val lookup: Exp.id * subst -> Exp.exp
      and substitute: subst -> Exp.exp -> Exp.exp
    end
```

The derived structure Subst depends essentially, and *overtly* on its antecedent Exp, because it contains the functions lookup and substitute that operate on types inherited from Exp. Clearly it will not be possible to make use of Subst except in a context where Exp is also available. In fact, we could view Subst as being merely an *enrichment* of the basic structure Exp.

Note also that if we define Subst directly in terms of Exp as above we get an inferred signature that contains global references to Exp and is therefore context dependent. We regard this as an unfortunate side effect that we would like to avoid.[6]

As indicated above, we will require the structure Subst to be "closed" by incorporating Exp as a substructure and defining the rest of its components in terms of that substructure rather than directly in terms of Exp. The definition of Subst still refers to Exp of course, but only in the definition of the substructure; essentially, we will have "internalized" Exp. The new definition of Subst and its signature are

```
structure Subst: SUBST =
  struct
    structure E = Exp
    type subst = (E.id * E.exp) list
    fun lookup(id,nil) = E.Var id
      | lookup(id,(id',e)::l) = if id=id' then e else lookup(id,l)
```

[6] It is possible to think of signatures as functions of structures under certain circumstances. See the discussion of dependent function types in [MAC86].

```
    fun substitute s E.Var(id) = lookup(id, s)
      | substitute s E.App(id,args) =
              E.App(id, map(substitute s, args))
  end

signature SUBST =
  sig
    structure E: EXP
    type subst
    val lookup: E.id * subst -> E.exp
    and substitute: subst -> E.exp -> E.exp
  end
```

Note that the signature of the new `Subst` is completely closed (except for the reference to signature `EXP`), since the external references to `Exp` have been replaced by references to the locally bound structure variable `E`.

We make it mandatory that structures which visibly inherit types from antecedent structures should be closed in this way, by imposing the *signature closure* rule of §2.2.2. This rule strengthens the original structure closure rule of §2.1.1 by limiting the role played by global structure and functor references. It also uncouples the signature environment from the structure environment, insuring that the meaning of a signature will not depend on the structure environment.

2.3.2. *Inherited exceptions*

Although an inherited type is the most obvious reason why an antecedent structure may be relevant to the *use* of the derived structure, there are other less obvious reasons. The first has to do with exceptions. Suppose that we define a structure `StackUser` using the structure `Stack` from §2.1.2 and a function `f` defined in `StackUser` calls `Stack.pop`:

```
structure StackUser =
  struct
    ...
    fun f(...) = ...Stack.pop(s)...
    ...
  end
```

Assuming that the type `Stack.stack` does not appear in the signature of `StackUser`, the dependence is not overt but it is still essential. Whenever we call the function `StackUser.f` it may cause the exception `Stack.pop` to be raised, and if we wish to trap that particular exception the antecedent structure `Stack` must also be available.

The idea of incorporating relevant antecedents as substructures also provides a way of explicitly acknowledging dependence based on inheritied exceptions such as `Stack.pop`. If we were to change the definition of `StackUser` to

```
structure StackUser =
  struct
    structure MyStack = Stack
    ...
    fun f(...) = ...MyStack.pop(s)...
    ...
  end
```

then we would be assured that wherever `StackUser` was used one would have access to the exception `Stack.pop` as `StackUser.MyStack.pop`. However, in this case the inclusion of `Stack` as a substructure is not required to close the signature of `StackUser`, so it is not enforced by the language. We might say that `Exp` was a mandatory prerequisite of `Subst` but `Stack` is only an optional prerequisite of `StackUser`. If `Stack` is not included, we will not be able to selectively trap the `pop` exception unless independent access to `Stack` has somehow been provided.

2.3.3. *Inherited objects*

The other circumstance where it is useful or necessary to know about the antecedent occurs when that antecedent maintains some private state variables that may be used for indirect communication with the derived structure. Consider, for example, a structure `Buf` that implements a buffer with operations `put` and `get`, and another derived structure `BufUser` that gets values from `Buf` and processes them:

```
structure Buf =
  struct
    local
      val buffer = array(...)
      ...
    in
      fun get() = ...
      and put(x) = ...
    end
  end

structure BufUser =
  struct
    ...
    fun f(...) = ...Buf.get()...
  end
```

Clearly `BufUser` does not inherit any types from `Buf`, and we may assume that functions in `BufUser` do not raise exceptions defined in `Buf`. But `Buf` is still relevant to the use of `BufUser` because it can be used to communicate values to `BufUser`. Once again we can acknowledge this essential dependence by including `Buf` as a substructure of `BufUser`:

```
structure BufUser =
  struct
    structure MyBuf = Buf
    ...
    fun f(...) = ...MyBuf.get()...
  end
```

As in the case of exception inheritance this inclusion is optional.

2.3.4. *Discussion*

(1) As indicated above, only visible inheritance of types forces a prerequisite structure to be included as structure. The other cases are optional, but if we abstract with respect to a structure to form a functor (see §2.4) it is convenient for that structure to include all its prerequisites. Otherwise it may be necessary to provide those prerequisites

as separate parameters.

On the other hand, substructures may be included even when there is no dependency involved. For instance, a structure might include several substructures simply as a means of packaging together a collection of loosely related utilities. Also remember that an arbitrary structure expression can be used to define a substructure, so a substructure may be a newly derived structure possibly including antecedents as its substructures.

(2) It is possible to avoid including whole substructures by causing only individual types, exceptions, or values to be inherited in the derived structure. For instance, `Subst` could have been defined as

```
structure Subst: SUBST =
  struct
    type my_id = Exp.id
     and my_exp = Exp.exp
     and subst = (my_id * my_exp) list
     ...
  end
```

with the closed signature

```
signature SUBST =
  sig
    type my_id
     and my_exp
     and subst
    val lookup: my_id * subst -> my_exp
    and substitute: subst -> my_exp -> my_exp
  end
```

Similar *ad hoc* declarations could be used to incorporate the `Stack.pop` exception in `StackUser` and the `Buf.put` function in `BufUser`.

To some extent the question of whether whole structures or individual types and values should be inherited is one of taste, but this proposal tacitly assumes that well designed structures will form the most natural units of inheritance.

(3) In some cases, an antecedent structure as a whole is not essential, but one of its substructures is. In such cases, it is possible to include only the relevant substructure of the antecedent, or even some new structure derived from it. For example, a `Unify` structure defined in terms of `Subst` might only need to inherit `Subst.E`, as in

```
structure Unify =
  struct
    structure Expr = Subst.E
    fun unify(Expr.Var(id), e): Expr.exp = ...
  end
```

2.4. Functors

2.4.1. *Functor declarations*

A functor is defined by abstracting a structure expression with respect to some or all of the free structure variables occurring in it. The result is a kind of function that can be applied to structures to produce new structures.

For example, suppose `E` is a structure of signature `EQ` as defined in §2.2.2. We will assume that `E` consists of a type and an associated equality predicate. We can define a derived structure `EListEq` that implements membership and list equality predicates for lists of elements of type `E.elem` as follows:

```
signature LISTEQ = sig
    structure Eq: EQ
    val member: Eq.elem * Eq.elem list -> bool
    and eqlists: Eq.elem list * Eq.elem list -> bool
end

structure EListEq: LISTEQ =
  struct
    structure Eq = E
    fun member(e,nil) = false
      | member(e,e'::l) = Eq.eq(e,e') orelse member(e,l)
    and eqlists(nil,nil) = true
      | eqlists(nil,_) = false
      | eqlists(_,nil) = false
      | eqlists(e1::l1,e2::l2) = Eq.eq(e1,e2) andalso eqlists(l1,l2)
  end
```

Note that `E` appears free in the body of the definition of `EListEq` and that the definition is uniform in `E: EQ` (i.e. it makes sense independently of the value of `E`). We can therefore abstract with respect to `E` to define a functor `ListEq` that provides list membership and equality predicates over any such structure:

```
functor ListEq (E: EQ): LISTEQ =
  struct
    structure Eq = E
    fun member(e,nil) = false
      | member(e,e'::l) = Eq.eq(e,e') orelse member(e,l)
    and eqlists ...
  end
```

The result signature specification, `LISTEQ` in this example, is optional, but the parameter signature specification is always required. As in the case of ordinary structure declarations, type inference is used to determine an anonymous result signature if one is not explicitly specified.

The body, or right hand side, of a functor declaration need not be an encapsulated declaration; it can be any sort of structure expression. Thus if `F1` and `F2` were functors of appropriate types, we could define another functor in terms of them by declaring

```
functor F3(A: SigA) = F1(A, F2(A))
```

Functors are always declared using this sugared syntax, with the arguments following the functor name on the left hand side, because there are no free-standing functor

expressions corresponding to lambda expressions for ordinary functions. Functors may not be passed as parameters to other functors, they can only be applied to structures. Thus functors are analogous to first-order functions that can only be applied to nonfunctional values. The "types" of functors are also left implicit; that is, we do not provide for the expression of "higher-order" signatures. See [MAC86] for further discussion of this point.[7]

2.4.2. *Multiple instances*

A functor can of course be applied many times to a variety of arguments. Indeed, the basic point of abstraction is to be able to "instantiate" the structure forming the body in the different contexts provided by the actual parameters, thereby creating specialized copies of the abstracted structure.

However, even when a structure has no antecedent structures it is sometimes useful to turn that structure into a nullary functor with an empty parameter list. This occurs typically when the structure defines some state variables and functions that operate on that state; such structures are analogous to *objects* in languages like Simula 67 and Smalltalk. For example, the functor `StackFun` defined below produces a new stack structure each time it is applied ("instantiated").[8]

```
functor StackFun () =
  struct
    local
        val stack: int list ref = ref nil
          {local stack data structure}
    in
        fun push x = (stack := x :: !stack)
        and pop () = (stack := tl(!stack))
        and top () = hd(!stack)
    end
  end

structure Stack1 = StackFun()
      and Stack2 = StackFun()
```

Note that the structures produced by this functor do not contain a stack type; they *are* stacks. The stack functor is a generator of stack objects, each with its own internal data structures representing the state of a stack. It is therefore quite similar to a class in Simula or Smalltalk. The closest thing to a stack type in this implementation is the result signature of the functor, i.e.

```
sig
  val push: int -> unit
  and pop: unit -> unit
  and top: unit -> int
end
```

[7] This restriction might be relaxed in a future version. It would, for instance, be reasonable and probably useful to allow Curried functors that would effectively produce functors as results. The language presented in [MAC86] permits nested abstractions.

[8] Here `ref` constructs updateable references to values, `!` dereferences, and `hd`, `tl`, and `::` (infix cons) are the usual primitive functions for lists.

which gives the interface, or set of functions available for operating on each stack object.

2.5. Type propagation

Ordinary type declarations do not hide the identity or structure of the types they name. Thus within the scope of a declaration such as

```
type t = int * int
```

`t` is simply an abbreviation for the type `int*int` and will match it during type checking. The same is true of type declarations within explicit structure definitions. For instance, after the declaration

```
structure A: sigA =
  struct
    type t = int * int
    ...
  end
```

the qualified name `A.t` is also just an abbreviation for the type `int*int`, and will match `int*int` in type checking. Qualified references via an explicitly defined structure name such as `A` are said to be "transparent", and A is said to "propagate" the identity of its type component `A.t`. Note that type components are *not* made "abstract" simply by being placed inside a structure.

On the other hand, references via bound structure variables (i.e. functor formal parameters) are said to be "opaque", because they cannot be seen as denoting anything beyond themselves. Thus in the body of

```
functor F(X: sigA) =
  struct
    ...X.t...
  end
```

the qualified type name `X.t` has no external referent and in type checking matches only itself. In general, references via "structure constants" (i.e. names that denote a particular structure) are transparent, while references via bound structure variables (functor parameters) are opaque. The use of opaque references to achieve type abstraction is discussed further in §2.7.

The notions of transparent and opaque reference are relative ones, depending on context. Consider the following definition:

```
functor G(X: sigA): sigG =
  struct
    structure B: sigB =
      struct
        type s = X.t list
        ...
      end
    ... B.s ...
  end
```

Within the body of the functor definition `B.s` transparently denotes `X.t list`, while `X.t` is opaque as before. After instantiating the functor body by the definition

```
structure C: sigG = G(A)
```

the qualified reference `C.B.s` transparently denotes `A.t list = (int*int) list`. Note how type components are "propagated" through functor calls.

As a further illustration, consider the following structure with signature `EQ` (defined in §2.2.2):

```
structure IntEq: EQ =
  struct
    type elem = int
    val eq = (op =): int*int -> bool
  end
```

`IntEq` *interprets* the type `int` by associating with it the standard equality predicate for integers. The qualified name `IntEq.elem` is just another name for `int`, and consequently the type of `IntEq.eq` is

```
IntEq.elem * IntEq.elem -> bool  =  int * int -> bool
```

and we are therefore justified in using `IntEq.eq` with ordinary integer expressions, as in `IntEq.eq(3, 2*5)`. The point is that building a structure around a type does not alter or hide the identity or form of the type. In particular, it does not make the type "abstract".

To see how type propagation applies when we make one structure a substructure of another and when a new structure is created by a functor application, consider the declaration

```
structure IntListEq = ListEq(IntEq)
```

By type propagation we have

```
IntListEq: LISTEQ
IntListEq.Eq = IntEq
IntListEq.Eq.elem = int
IntListEq.member: int * int list -> bool
```

Here is a somewhat more involved example involving lexicographic ordering of lists of ordered elements (`/\` in `LexOrd` denotes strict boolean conjunction, assumed to be primitive):

```
signature ORDSET =
  sig
    type s
    val le: s * s -> bool
  end

functor LexOrd (O: ORDSET): ORDSET =
  struct
    type s = O.s list
    fun 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 O.le(x,y)
  end
```

```
structure IntOrd: ORDSET =
  struct
    type s = int
    val le = (op <=)
  end
{ IntOrd: ORDSET
  IntOrd.s = int
  IntOrd.le: int * int -> bool }

structure IntListOrd = LexOrd(IntOrd)
{ IntListOrd: ORDSET
  IntListOrd.s = int list
  IntListOrd.le: int list * int list -> bool }
```

These typings justify applying `IntListOrd.le` to ordinary integer list expressions, as in

```
IntListOrd.le([1, x+1], (2*y)::1)
```

Note that in this example, the result structure of `LexOrd` is simply an `ORDSET`; it does not inherit the parameter `ORDSET`, `O`, as a substructure. Thus although the element type propagates from the parameter to the result, the ordering on elements does not.

The previous examples entail adding an interpretation to an existing type. When a structure defines a type component in terms of a (generative) datatype, however, a new type is created unique to that structure. That type cannot, of course, be operated on by previously defined functions (except in a trivial way by polymorphic functions), but it will propagate as the structure is used to build other structures. The internal constituents of the datatype will be accessible if and only if the associated constructors are exported, so a form of type abstraction can be obtained by defining the representation type as a datatype and failing to export the constructors. The more general approach to abstraction explained in §2.7 relies on the opacity of parameter references instead of the generative nature of datatype definitions is discussed.

2.6. Coherence and sharing constraints

Complex programs will be built as layered hierarchies of structures, with higher level structures defined in terms of facilities provided by lower level structures, and this hierarchical organization will be (partially) reflected in the substructure relationships. It will be common for several higher level structures to share the same lower level antecedents, so the substructure dependency graph will be an acyclic directed graph in general, rather than a simple tree. In fact, such shared antecedents are essential for communication and cooperation between the higher level structures. Informally, we say that structures are *coherent* when they can cooperate by virtue of shared antecedents.

In the absence of parameterization, coherence comes about naturally "by construction". That is, higher level structures are simply defined in terms of the same antecedent structures. We can illustrate this point in terms of the following simplified outline of a set of structures implementing a parser.

```
structure Symbol =
  struct
    datatype symbol = mksymbol of string * ...
    val mksymbol: string -> symbol = ...
    and eqsymbol: symbol * symbol -> bool = ...
  end

structure AbstSyntax =
  struct
    structure Symb = Symbol
    type term = ...
    val Idname: term -> Symb.symbol = ...
    ...
  end

structure SymTable =
  struct
    structure Symb = Symbol
    type info = ...
     and table = ...
    val mktable: unit -> table = ...
    and lookup: Symb.symbol * table -> info = ...
    ...
  end

structure Parser =
  struct
    structure AS = AbstSyntax
    structure ST = SymTable
    val symtable = ST.mktable()
    fun parse(...) =
    ... ST.lookup(AS.Idname(t), symtable) ...
  end
```

The functions `ST.lookup` and `AS.Idname` can be composed in the definition of `Parser` because both of their types involve the type `Symbol.symbol` inherited from the structure Symbol. Because of the sharing relationship (Figure 1),

```
Parser.AS.Symb = Symbol = SymTable.Symb
```

the actual types of the functions are

```
ST.lookup: Symbol.symbol * SymTable.table -> SymTable.info
AS.Idname: AbstSyntax.term -> Symbol.symbol
```

and therefore they may be composed.

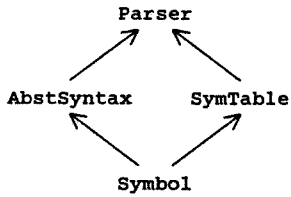

Figure 1. The dependency graph for the parser structures.

Note that because of type propagation, it is not strictly necessary that `AbstSyntax` and `SymTable` share the same substructure `Symbol`. They might be based on two different `Symbol` structures, `Symbol1` and `Symbol2` say, that coincidentally share the same representation type `symbol`, i.e. `AbstSyntax.Symb.symbol = SymTable.Symb.symbol`. Such "accidental" compatibility would probably not be desirable, since the common representation type (e.g. `int`) might be interpreted quite differently in the two structures (e.g. as indices into two different symbol table data structures).

Now suppose we wanted to use the parser mechanism implemented in the structure `Parser` with various abstract syntaxes and various implementations of symbol tables. To do this we would convert `Parser` into a functor by abstracting with respect to both `AbstSyntax` and `SymTable`. The definition of the `Parser` functor would look something like this:

```
functor ParserFun(AbstSyntax: ABSTSYNTAX, Symtable: SYMTABLE) =
  struct
    structure AS = AbstSyntax
    val symtable = SymTable.mktable()
    fun parse(...) =
     ... SymTable.lookup(AS.Idname(t), symtable) ...
  end
```

where `ABSTSYNTAX` and `SYMTABLE` are the signatures of the original `AbstSyntax` and `SymTable` structures.

Now consider the expression "`SymTable.lookup(AS.Idname(t), symtable)`" in the body of the functor `ParserFun`. In the structure `Parser` this had type checked correctly because it was known "manifestly" that the operations shared the same type `symbol`. But now `AbstSyntax` and `SymTable` no longer refer to particular, known structures; they are formal parameters representing arbitrary structures of the specified signatures. These signatures insure that each parameter has a substructure `Symb` of signature `SYMBOL` (the signature of the structure `Symbol`), but they do not insure that these two substructures are the same. We can infer that

```
AbstSyntax.Idname: AbstSyntax.term -> AbstSyntax.Symb.symbol

SymTable.lookup: SymTable.Symb.symbol * SymTable.table
                    -> SymTable.info
```

but we cannot presume that

```
AbstSyntax.Symb.symbol = SymTable.Symb.symbol          (*)
```

and therefore the expression in question does not type check.

The problem is that the functor parameter specification does not insure the required "coherence", or overlapping heritage, between the two parameter structures. The solution is to add to the parameter specification of `ParserFun` a *sharing constraint* that asserts the desired sharing of antecedent structures between the parameters. The modified declaration is

```
functor ParserFun (AbstSyntax: ABSTSYNTAX, SymTable: SYMTABLE
        sharing AbstSyntax.Symb = SymTable.Symb) = . . .
```

The sharing constraint could have specified only that the type `symbol` must be shared, as in (*), but as pointed out before, requiring that the entire `Symb` substructures be shared guarantees that they will be interpreted in the same way by the associated operations.

In general, the parameter sharing specification for a module can include several multi-equations of the form $path_1 = \ldots = path_n$, each relating a set of parameter substructures of the same signature or a set of type components. The equations involving types can contain at most one "absolute" type term (i.e. a type term involving only pervasive primitives). This rule precludes constraints like "`int = bool`."

There is another kind of sharing involved in the definition of functors, namely the sharing between parameter structures and the result structure. This sort of sharing is important because it determines how the result depends on the parameters, and this dependence information is essential for performing static checking of other sharing constraints. In some formulations functional dependencies of this sort are expressed using dependent function types (see [MAC86]). But in this design, functional dependencies are not explicitly declared but are inferred from the definitions of functors. It may be useful to make these inferred dependency specifications available to the user in some form.

2.7. Type abstraction

The present form of abstract type declaration in Standard ML depends on the generative nature of the datatype construct and uses the device of restricting the scope of the associated constructors to a body in which interpretive functions for the type are defined. Outside of that body, one cannot use the constructors to access the internal structure (representation) of the type, so it is considered to be "abstract".

Another, less artificial notion of type abstraction is provided by structure abstraction as used in the formation of functors. If a parameter signature contains a simple type specification, such as

```
type complex
```

then so far as the body of that functor is concerned the type `complex` must be treated as abstract, because it is essentially a formal parameter whose potential bindings are unknown. Thus, as suggested by Reynolds [Rey74], type abstraction can be reduced to a kind of lambda abstraction over types and their associated interpretive functions (which in his case were nested or curried abstractions, but in our case have been uncurried by packaging the type and its functions together in a structure parameter). The structures to which the functor is later applied supply implementations of the abstraction specified by the parameter signature. Thus structure abstraction enforces the separation between implementation and use of the abstraction, while functor application makes the connection between the abstraction and its implementation(s).

We can employ this view of type abstraction as functional abstraction over structures

to provide a new form of abstract type declaration. The idea is to regard the scope of the declaration (e.g. at top level the rest of the program, i.e. all further top level declarations) as implicitly forming the body of a functor that is abstracted with respect to a structure of the signature specifying the abstract type and its interpreting functions. The implementation of the abstract type is regarded as the argument to which this functor is applied. The definition and use of an abstraction with signature `ABST` and implementation structure `Impl` therefore correspond roughly to the following declarations:

```
functor User (A: ABST) =
  struct
    ... A ...          {scope of abstraction}
  end

User(Impl)
```

But as usual we sugar the combined abstraction and application as a declaration construct, in this case using the keyword "`abstraction`" (in place of the usual keyword "`structure`"):

```
abstraction A: ABST = Impl

 ... A ...
```

In the scope of this declaration, references via `A` are treated as opaque, as though `A` were a functor formal parameter. As with ordinary structure declarations, the open declaration, "open `A`", makes it possible to refer directly to the types and operations of `A` without using qualified names, but these abbreviated forms of reference remain opaque.

As noted in §2.5, the normal structure declaration does not provide type abstractions because the rules of type propagation make references to type components transparent. Consider the following program fragment:

```
signature POINT =
  sig
    type point
    val x_coord: point -> int
    ...
  end

structure P: POINT =
  struct
    type point = int * int
    fun x_coord(x,_) = x
    ...
  end

 ...P.x_coord(3,4)...
```

The last expression is legal because `P.point` is recognized as equivalent to `int*int`, but if we declared P as an abstraction, as in

```
abstraction P: POINT =
  struct
    type point = int * int
    fun x_coord(x,_) = x
    ...
  end
```

then the expression "`P.x_coord(3,4)`" would fail to type check. In practice, the hiding of the types in an abstraction can be accomplished by "freezing" the type bindings so that they are not expanded in the process of type matching (i.e. they match "by name").

The distinction between "`structure`" and "`abstraction`" declarations is reminiscent of the distinction between let and lambda bindings with regard to polymorphism. The type propagation rules are designed to let us take advantage of our knowledge of the explicit definition of a structure, including its type components; when we define an abstraction, we choose not to take advantage of this knowledge.

2.8. Signature and type checking

2.8.1. *Signature matching*

The type checking of structure and functor definitions involves one-way matching of a *candidate* signature against a *target* signature. When checking a structure expression, including a functor body, against its declared signature, the declared signature is the target and the inferred signature of the expression is the candidate. When checking a functor application, the declared parameter signature is the target and the signature of the corresponding actual parameter is the candidate. The matching relationship is written as "*candidate* matches *target*".

The matching process uses the fact that polymorphic types in value specifications are considered generic, and will therefore match any instance. Thus the specification

```
val x: 'a list -> bool
```

in a candidate signature will match the corresponding specification

```
val x: int list -> bool
```

in the target signature, but not the other way around.

We also permit a candidate signature to match a smaller target signature, i.e. one which specifies only a subset of the bindings specified in the candidate signature. This provides an implicit *forgetful* coercion from the "richer" candidate signature to the "poorer" target signature. For example, the candidate signature

```
sig
  datatype 'a stack = nilstack | push of 'a * 'a stack
  exception pop: unit and top: unit
  val empty: 'a stack -> bool
  and pop: 'a stack -> 'a stack
  and top: 'a stack -> 'a
end
```

will match the following target signature

```
sig
  type 'b stack
  val push: 'b * 'b stack -> 'b stack
  and empty: 'b stack -> bool
  and pop: 'b stack -> 'b stack
end
```

Note that a datatype specification in the candidate signature may match a simple type specification in the target, as in the case of the `stack` type in this example. Furthermore, when this occurs, a constructor from the candidate type specification, such as `push` in this case, may match a value specification in the target. As specified in the target signature, `push` is an ordinary function and cannot be used as a constructor in pattern matching. When the target signature contains a datatype specification, the candidate must also specify a datatype and the two specifications must match exactly (modulo type variable names and the order of constructors).

Note also that the order of specifications in a signature is significant in signature matching. It would be possible to extend the automatic forgetful coercion to also deal with reorderings that did not violate the *specification before use* rule.

The process of pure signature matching proceeds as follows:

1. Match corresponding substructures:
   For each substructure spec

   ```
   structure X: SIGX
   ```

   in the target, there must be a corresponding substructure spec with the same name

   ```
   structure X: SIGX'
   ```

   in the candidate, and `SIGX'` must match `SIGX`. Note that forgetful coercions can be used in matching substructure signatures, recursively to any depth.
2. Match corresponding types:
   a. For each simple type spec in the target there must be a corresponding type spec (simple or datatype) in the candidate with the same arity.
   b. For each datatype spec

      ```
      datatype ('a1,..,'am) tycon = con_1 of ty1 | ...
                                  | con_n of tyn
      ```

      in the target there must be a corresponding type spec

      ```
      datatype ('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 the types `tyi` and `tyi'` must agree assuming the identity of corresponding substructures and types in the candidate and target signatures.
3. Match corresponding value and exception bindings:
   a. 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'`, assuming the identity of corresponding substructures and types of the candidate and target signatures.

b. Specifications of exceptions in the target and candidate must match exactly (they are both monotypes).

When matching the signature of an actual structure against a declaration signature constraint or a functor parameter specification, we need to modify the above procedure to take into account the type bindings in the structure itself. For example, consider the declarations

```
signature POINT =
  sig
    type point
    val origin: point
  end

structure P: POINT =
  struct
    type point = int * int
    val origin = (0,0)
  end
```

The inferred signature of the structure expression in the declaration of `P` is

```
sig
  type point
  val origin: int*int
end
```

which does not match the signature `POINT` according to the above procedure because `point` and `int*int` will not match in step 3.b. If we modify that step to take into account the type bindings in the actual structure (`P` in this case), then these types will match. (This is a sort of inverse process to type propagation.)

2.8.2. *Typing structure components*

The type propagation behavior discussed in §2.5 is implemented by the rules for typing references to structure components. The type specs for values and exceptions in a signature are specifications relative to the actual type components of structures having that signature. Therefore, to get the true type of a value or exception component we instantiate the type spec in the signature using the type bindings in the structure itself.

Consider, for instance, the structure `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 `s*s->bool`, where `s` is the type component of the signature, so to get the type of `IntOrd.le` we instantiate that type with the type bound to `s` in `IntOrd`, namely `int`, yielding the type `int*int->bool`.

If the type of a structure value component is polymorphic, it can always be used generically, even if the structure is a formal parameter of a functor. This is because the signature matching rules insure that the corresponding value component of the actual parameter will indeed be as polymorphic as the specification in the parameter signature.

2.8.3. *Checking sharing constraints*

Sharing constraints can be checked statically based on two sources of information. The first source is the current structure environment, which contains information on all substructure relationships among existing structures. If functor `M` is defined by

```
functor M(X: SIGX, Y: SIGY sharing X.U = Y.V)
```

and we want to check the application `M(A,B)` where `A` and `B` are actual structures, then we simply check that `A.U` and `B.V` are the same substructure or type.

When the parameters of a functor application are formal structure variables or compound structure expressions we rely on a combination of declared sharing constraints among the formal variables and statically inferred sharing propagation specifications for the functors involved. Each functor's sharing propagation specifications are inferred and recorded implicitly by the compiler. See [MAC86] for further discussion of type specifications for functors.

2.9. Miscellaneous notes

2.9.1. *Restrictions on module and signature declarations*

Signatures and functors may only be declared at top level, while structures may only be declared at top level or as components of other structures (including the bodies of functor definitions). In particular, structures and functors cannot be declared within functions or conditional expressions.

These restrictions are intended to prevent dynamic and conditional creation of structures, and thus insure that the whole hierarchy of structures is statically evident to the compiler. This makes it possible for the compiler to statically interpret the indirections involved in qualified names and produce code for efficient dynamic reference to structure components even when these lie several layers deep in the hierarchy.

The restrictions should not be bothersome for those structures that implement interpreted types; after all, types are rarely declared within functions or conditional expressions. For those structures implementing "objects", the restriction may indeed have a constraining effect, but the use of structures as objects is already limited by the fact that structures are not values and may not be manipulated by functions.

2.9.2. *Sharing constraints in signatures*

Strictly speaking, sharing constraints are only required in functor parameter specifications. But when sharing constraints specify sharing between substructures of a single structure, it would be reasonable to allow them to be included in the signature of that structure. Such "intrastructure" sharing constraints could reduce the number of constraints required in functor parameter specifications, which would concentrate on "interstructure" constraints. For example, the result signature for the `Parser` functor (§2.6) might be defined as

```
signature PARSER =
  sig
    structure AS: ABSTSYNTAX
          and ST: SYMTABLE
      sharing AS.Symb = ST.Symb
    ...
  end
```

A related extension would be to allow type identities as type specifications in signatures, as in the following two examples

```
sig
  type t = int list
  val f: t -> int
end

sig
  structure A: SIGA
  type t = A.s * A.s
  ...
end
```

Both of these extensions would presumably continue to obey the signature closure rule, *i.e.* the sharing equations and type identities would refer only to locally specified types and structures.

2.9.3. *Recursive modules*

There are two problems that would have to be solved before mutually recursive structure definitions could be admitted. First, it would have to be possible to detect all indirect circularities among the type and value declarations of the mutually recursive structures and make sure that these conformed to the usual rules: circular type definitions could involve only datatypes and circular value definitions could involve only functions defined explicitly by lambda (i.e. `fun`) expressions. This requirement could probably be satisfied by insisting that all the recursive structures are defined by encapsulated declarations (i.e. "`struct` ... `end`" forms) and that none of the structures is used as an argument of a functor within any of the definitions.

The more serious problem is that mutually recursive structure definitions would generally involve cyclical substructure relationships. Consider the following declaration

```
structure rec A = struct
                    structure Ab = B
                    datatype a = a1 | a2 of B.b
                  end
          and B = struct
                    structure Ba = A
                    datatype b = b1 | b2 of A.a
                  end
```

What signatures should be inferred for `A` and `B`? It is clear that their signatures would also have to involve mutual recursion, such as

```
signature rec SIGA = sig
                       structure Ab: SIGB
                       datatype a = a1 | a2 of B.b
                     end
          and SIGB = sig
                       structure Ba: SIGA
                       datatype b = b1 | b2 of A.a
                     end
```

Circularities in the substructure relationship and mutually recursive signatures do not

appear to be inherently problematic or infeasible, but there consequences have not yet been explored in sufficient depth for them to be incorporated in the current proposal.

Mutually recursive functors seem to present even greater difficulties. Consider, for instance,

```
functor rec A () = struct
                     structure Ab = B()
                   end
        and B () = struct
                     structure Ba = A()
                   end
```

An instantiation of A or B would not terminate, or if it did terminate by some trick of lazy evaluation the resulting substructure graph would not be well founded. This situation is not atypical, so it does not appear to be possible to make sensible use of recursive functors.

2.9.4. *Views*

Sometimes one wants to make an structure X of signature SIG1 masquerade as an structure of some other, presumably simpler signature SIG2, so that X can be passed to a functor requiring a SIG2 parameter. If SIG2 is a simple subset of SIG1 then the forgetful coercion discussed in §2.8 will do the job. But in general a more serious transformation involving a change of names will be required. This process can be thought of as creating a new "view" of the structure [GOG83], or as applying a "signature morphism" to the structure. Such a signature transformation can easily be expressed as a functor, or as an *ad hoc* definition of a new derived structure.

For example, suppose the signature SET is defined as follows

```
signature SET =
  sig
    structure E: EQ
    type set
    val singleton: E.elem -> set
    and union: set * set -> set
    and member: E.elem * set -> bool
    and subset: set * set -> bool
    and eqsets: set * set -> bool
  end
```

Now suppose we wanted to consider an SET structure as an ORDSET ordered by the subset relation, so that we could lexicographically order lists of sets. The functor that expresses this view uniformly on all SET structures is (see 5 for definition of ORDSET)

```
functor SetOrd (S: SET) : ORDSET =
  struct
    type s = S.set
    val le = S.subset
  end
```

If IntSet: SET is a structure implementing sets of integers we can get an ordering of lists of sets of integers by

```
structure IntSetListOrd = LexOrd(SetOrd(IntSet))
```

Alternatively, we could create an *ad hoc* view of `IntSet` as an `ORDSET` by using a structure expression as follows.

```
structure IntSetListOrd = LexOrd(struct
                                    type elem = IntSet.set
                                    val le = IntSet.subset
                                 end)
```

Note, however, that these new views are new structures, though they do share with the old structures, *e.g.* `IntSetListOrd.s = IntSet.set`.

3. Conclusions and Future Work

It should be emphasized that this design is tentative and to a large extent untested. It needs to be proved through (1) implementation, (2) extensive use in real programming, and (3) thorough formalization and theoretical analysis. The proposal has been influenced by a mixture of theoretical and practical influences, and a number of compromises have been made that may not turn out to be optimal.

One of the major goals of the design is to support separate compilation of structures and modules. But a good deal of work remains to be done to actually implement separate compilation. The main task is to devise a scheme for representing and maintaining persistent environments of signatures, structures, and modules. These environments might be called "libraries" or "systems". On top of this facility one should be able to build tools for version maintenance analogous to Unix's *make* command and makefiles.

There are also a number of directions in which the design could be extended or modified such as

(1) Pebble-like dependent function types (as in the language described in [MAC86]).

(2) Type declarations residing in signatures, where they can be shared by both users and implementors of the signature.

(3) User-supplied pervasive bindings, *i.e.* letting the user declare bindings that will pervade the whole system like the built-in primitives do.

(4) Conditional structure expressions: it appears possible to differentiate between opaque (or abstract) structures and transparent ones, and it seems possible to use conditional expressions to define opaque structures.

I hope that this design, or something close to it, will not only help to improve the structure of large ML programs and streamline the development process, but will in fact lead to new programming styles and metaphors. A great deal of work lies ahead of us before this hope is justified.

References

[BOE80] H. Boehm, A. Demers, and J. Donahue, *An informal description of Russell*, Technical Report TR 80-430, Computer Science Dept., Cornell Univ., October 1980.

[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.

[BL84] R. M. Burstall and B. Lampson, *A kernel language for abstract data types and modules*, Semantics of Data Types, G. Kahn, D. B. MacQueen, and G. Plotkin, eds., Lecture Notes in Computer Science, Vol 173, Springer-Verlag, Berlin, 1984.

[BUR84] R. M. Burstall, *Programming with modules as typed functional programming*, Int'l Conf. on 5th Generation Computing Systems, Tokyo, Nov. 1984.

[CAR83] L. Cardelli, *ML under Unix*, Polymorphism, I.3, December 1983.

[CW85] L. Cardelli and P. Wegner, *On understanding types, data abstraction, and Polymorphism*, in preparation.

[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.

[MAC86] D. B. MacQueen, *Using dependent types to express modular structure: experience with Pebble and ML*, 13th Annual ACM Symp. on Principles of Programming Languages, January 1986 (to appear).

[ML82] P. Martin-Löf, *Constructive mathematics and computer programming*, in Logic, Methodology and Philosophy of Science, VI (Proceedings of the 6th Int. Cong., Hanover, 1979), North-Holland, Amsterdam, 1982, pp. 153-175.

[MIL78] R. Milner, *A theory of type polymorphism in programming*, JCSS, 17(3), December 1978, pp. 348-375.

[MIL85] R. Milner, *The Standard ML Core Language*, Polymorphism II.2, October 1985.

[MMS79] J. G. Mitchell, W. Maybury, and R. Sweet, *Mesa Language Manual*, Xerox PARC Research Report, CSL-79-3, April 1979.

[MP85] J. C. Mitchell and G. D. Plotkin, *Abstract types have existential types*, 12th ACM Symp. on Principles of Programming Languages, New Orleans, Jan. 1985, pp. 37-51.

[Rey74] J. C. Reynolds, *Towards a theory of type structure*, in Colloquium sur la Programmationt, Lecture Notes in Computer Science, Vol. 19, Springer Verlag, Berlin, 1974, pp. 408-423.

[Rey85] J. C. Reynolds, *Three approaches to type structure*, Mathematical Foundations of Software Development, Lecture Notes in Computer Science, Vol. 185, Springer, Verlag, Berlin, 1985, pp. 97-138.

Appendix A: Syntax

The following syntax specifications follow the conventions of § 2.8 of [MIL85], except that keywords are identified by `this font` while syntactic metavariables are in italics. The metavariables *sig_id*, *str_id*, and *fnc_id* range over ordinary identifiers used as names for signatures, structures, and functors, respectively.

The syntax of the core language must be extended to admit the following qualified forms of names at any applied occurrence within an expression or type. However, defining occurrences are still restricted to the simple, unqualified form.

QUALIFIED NAMES

str_path ::=
  str_id_1 . __ .str_id_n ($n \geq 1$)

var′ ::=
  var
  str_path.var

con′ ::=
  con
  str_path.con

exn′ ::=
  exn
  str_path.exn

tycon′ ::=
  tycon
  str_path.tycon

SIGNATURES *sig*

sig ::=
  sig_id (signature name)
  `sig` $spec_1$ __ $spec_n$ `end` (signature specification)

spec ::=
  `datatype` *db*
  `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
  str_spec

str_spec ::=
  `structure` str_id_1 : sig_1 `and` __ `and` id_n : sig_n

SIGNATURE DECLARATIONS *sigb*

sigb ::= (signature binding)
  sig_id = *sig*
  $sigb_1$ and __ and $sigb_n$ (multiple, n ≥ 2)

dec ::=
  `signature` *sigb* (signature declaration)

STRUCTURES *str*

str ::=
  str_path
  `struct` *dec* `end` (basic structure expression)
  mod_id (*str_seq*) (module application)

STRUCTURE DECLARATIONS *strb*

strb ::=
  str_id <<: *sig*>> = *str*
  $strb_1$ and __ and $strb_n$ (multiple, $n \geq 2$)

dec ::=
  `structure` *strb*

FUNCTOR DECLARATIONS

path ::=
  tycon
  str_path<<.*tycon*>>

path_eq ::=
  $path_1$ = __ = $path_n$

share_spec ::=
  `sharing` $path_eq_1$ and __ and $path_eq_n$

param_spec ::=
  str_id_1 : sig_1, __, str_id_n : sig_n <<*share_spec*>>

fncb ::=
  fnc_id (<<*param_spec*>>) <<: *sig*>> = *str*
  $fncb_1$ and __ and $fncb_n$

dec ::=
  `functor` *fncb*

OPEN

dec ::=
  open str_id_1 __ str_id_n $(n \geq 1)$

Restrictions

(1) Functor and signature declarations can appear only at top level. Structure declarations may appear at top level or immediately within a basic structure expression.