Introduction to Classic ML

Christoph Kreitz and Vincent Rahli

Department of Computer Science, Cornell-University
Ithaca, NY 14853-7501, U.S.A.

1 The History of ML

Several versions of the programming language ML have appeared over the years, between the time it was first designed and implemented by Milner, Morris and Wadsworth at the University of Edinburgh in the early 1970's, and the time it was settled and standardized in the mid-1980's. The original ML, the *meta-language* of the Edinburgh LCF system, is defined in [GMW79].

The ML used in the NUPRL [CAB+86, Kre02, ABC+06] system is fairly close to the original. It is derived from a early version that Huet at INRIA and Paulson at the University of Cambridge were working on in 1981. Todd Knoblock at Cornell made most of the NUPRL specific modifications in the mid-1980's. NUPRL's ML hasn't changed since then and is not compatible with the ML versions that are widely used today.

The ML of Huet and Paulson is described in the preface to '*The* ML *Handbook*' [CHP84]. Huet used this version in the Formel project; and it subsequently evolved into a version of ML called CAML. Paulson also used it, as part of the first version of Cambridge LCF, but switched to Standard ML [MTH90, MTHM97] in the later versions of Cambridge LCF [Pau87].

The CAML language [CH90, WAL+90] is now rarely used. But there is a scaled down version called CAML-Light which is actively used in teaching programming to over 10,000 engineers a year in France. Its object-oriented version OCaml [Ler00] have become quite popular in recent years and has been used in the implementation of the group communication toolkit Ensemble [Hay98, BCH+00]. It is also the implementation language of the COQ [BC04] theorem prover. OCaml has directly influenced the design of the Microsoft language F# [SGC07]. The Standard ML language has also become increasingly popular for implementing theorem provers such as HOL [GM93] or Isabelle [Pau90].

The description of ML that appears in the following Sections is based very closely on 'The ML Handbook' [CHP84]. It was adapted for NUPRL purposes from LaTeX sources provided by the HOL theorem proving group in Cambridge. For completeness (and historical interest), the preface to '*The* ML *Handbook*' and the preface to '*Edinburgh LCF: a Mechanised Logic of Computation*' are reproduced below.

1.1 Preface to 'The ML Handbook'

This handbook is a revised edition of Section 2 of 'Edinburgh LCF', by M. Gordon, R. Milner, and C. Wadsworth, published in 1979 as Springer Verlag Lecture Notes in Computer Science nº 78. ML was originally the meta-language of the LCF system. The ML system was adapted to Maclisp on Multics by Gérard Huet at INRIA in 1981, and a compiler was added. Larry Paulson from the University of Cambridge completely redesigned the LCF proving system, which stabilized in 1984 as Cambridge LCF. Guy Cousineau from the University Paris VII added concrete types in the summer of 1984. Philippe Le Chenadec from INRIA implemented an interface with the Yacc parser generator system, for the versions of ML running under Unix. This permits the user to associate a concrete syntax with a concrete type.

The ML language is still under design. An extended language was implemented on the VAX by Luca Cardelli in 1981. It was then decided to completely re-design the language, in order to accommodate in particular the call by pattern feature of the language HOPE designed by Rod Burstall and David MacQueen. A committee of researchers from the Universities of Edinburgh and Cambridge, the Bell Laboratories and INRIA, headed by Robin Milner, is currently working on the new extended language, called Standard ML. Progress reports appear in the Polymorphism Newsletter, edited by Luca Cardelli and David MacQueen from Bell Laboratories. The design of a core language is now frozen, and its description will appear in a forthcoming report of the University of Edinburgh, as 'The Standard ML Core Language' by Robin Milner.

This handbook is a manual for ML version 6.1, released in December 1984. The language is somewhere in between the original ML from LCF and standard ML, since Guy Cousineau added the constructors and call by patterns. This is a LISP based implementation, compatible for Maclisp on Multics, Franzlisp on VAX under Unix, Zetalisp on Symbolics 3600, and Le_Lisp on 68000, VAX, Multics, Perkin-Elmer, etc... Video interfaces have been implemented by Philippe Le Chenadec on Multics, and by Maurice Migeon on Symbolics 3600. The ML system is maintained and distributed jointly by INRIA and the University of Cambridge.

1.2 Preface to 'Edinburgh LCF'

ML is a general purpose programming language. It is derived in different aspects from ISWIM, POP2 and GEDANKEN, and contains perhaps two new features. First, it has an escape and escape trapping mechanism, well-adapted to programming strategies which may be (in fact usually are) inapplicable to certain goals. Second, it has a polymorphic type discipline which combines the flexibility of programming in a typeless language with the security of compile-time type checking (as in other languages, you may also define your own types, which may be abstract and/or recursive).

For those primarily interested in the design of programming languages, a few remarks here may be helpful both about ML as a candidate for comparison with other recently designed languages, and about the description of ML which we provide. On the first point, although we did not set out with programming language design as a primary aim, we believe that ML does contain features worthy of serious consideration; these are the escape mechanism and the polymorphic type discipline mentioned above, and also the attempt to make programming with functions—including those of higher type—as easy and natural as possible. We are less happy about the imperative aspects of the language, and would wish to give them further thought if we were mainly concerned with language design. In particular, the constructs for controlling iteration both by boolean conditions and by escape-trapping (which we included partly for experiment) are perhaps too complex taken together,

and we are sensitive to the criticism that escape (or failure, as we call it) reports information only in the form of a string. This latter constraint results mainly from our type discipline; we do not know how best to relax the constraint while maintaining the discipline.

Concerning the description of ML, we have tried both to initiate users by examples of programming and to give a precise definition.

2 Introduction and Examples

2.1 Classic ML versus EventML

The EventML language has emerged from mixing Classic ML with a logic called *Logic of Events* [Bic09, BC08, BCG11]. This document presents the ML part of the EventML language and makes explicit when EventML's syntax varies from Classic ML.

2.2 Sessions

When writing ML pieces of code, typically users write global (or top-level) declarations (see Section 2.4 below) in a file and evaluate expressions (see Section 2.3 below) in a ML session. The fact that global declarations can only be written in a file and not in a session comes from using EventML sessions which do not allow global declarations. Note that there are ML sessions other than EventML sessions that allow such global declarations (such as the ones used in Nuprl).

Each example presented below follows the same schema: it is split into two boxes, the left box being a ML session running on the file partially displayed in the right box. Also, each example is self-contained in the sense that each time an example makes use of a global declaration, this declaration is in the right part of the example.

EventML provides ways to use functions defined in a library file (extracted from Nuprl) that is distributed with EventML. One way is to use `import` declaration. Many functions can be imported such as functions to deal with lists (see, e.g., Section 2.7). EventML comes with an Emacs UI that provides a feature to display the available functions (the provided library can also be consulted in any other text editor).

2.3 Expressions

In this tutorial, the ML prompt is `#` (the EventML prompt is `EventML#`), so lines beginning with this contain the user's contribution; all other lines are output by the system.

```
# 2 + 3;;
5 : Int
```

1

ML prompted with `#`, the user then typed `␣2 + 3;;␣` followed by a carriage return ↵; ML then responded with `␣5 : int␣` , a new line, and then prompted again. In general to evaluate an expression e one types e followed by a carriage return; the system then prints e's value and type (the type prefaced by a colon).

2.4 Declarations

The declaration `let` x `=` e evaluates e and binds the resulting value to x.

```
# x;;
6 : Int
```

```
let x = 2 * 3;;
```
2

To bind the variables $x_1, \ldots, x_n$ simultaneously to the values of the expressions $e_1, \ldots, e_n$ one can perform the declaration `let (`x_1`,`x_2`,...,`x_n`) = (`e_1`,`e_2`...,`e_n`)`.

```
# x;;
10 : Int
# y;;
6 : Int
```

```
let x = 2 * 3;;
let y = 10;;
let (x, y) = (y, x);;
```
3

A declaration d can be made *local* to the evaluation of an expression e by evaluating the expression `let` d `in` e (or equivalently e `where` d).

```
# let x = 2 in x * y;;
12 : Int
# (x * y) where x = 2;;
12 : Int
```

```
let x = 2 * 3;;
let y = 10;;
let (x, y) = (y, x);;
```
4

2.5 Functions

To define a function f with formal parameter x and body e one performs the declaration `let` f x `=` e. To apply the function f to an actual parameter e one evaluates the expression f e.

```
# f;;
- : Int -> Int
# f 4;;
8 : Int
```

```
let f x = 2 * x;;
```
5

Functions are printed as a dash, `-`, followed by their type, since a function as such is not printable. Application binds more tightly than anything else in the language; thus, for example, f `3 + 4` means `(`f `3) + 4` not f `(3 + 4)`. Functions of several arguments can be defined:

```
# plus 3 4;;
7 : Int

# f 4;;
7 : Int
```

```
let plus x y = x + y;;

let f = plus 3;;
```
6

Application associates to the left so `plus 3 4` means `(plus 3) 4`. In the expression `plus 3`, the function `plus` is partially applied to `3`; the resulting value is the function of type `int -> int` which adds `3` to its argument. Thus `plus` takes its arguments one at a time. We could have made `plus` take a single argument of the cartesian product type `(int * int)`:

```
# plus(3,4);;
7 : Int

# let z = (3,4) in plus z;;
7 : Int

# plus 3;;
[ERROR]
kind:
type constructor clash between Int and *
slice:
⟨..let plus (⟨..⟩, ⟨..⟩) = ⟨..⟩;;
 ..plus 3;;
 ..⟩

Untypable.
```

7

```
let plus(x,y) = x + y;;
```

As well as taking structured arguments (e.g. `(3,4)`) functions may also return structured results.

```
# sumdiff(3,4);;
(7, ~1) : Int * Int
```

8

```
let sumdiff(x,y) =
   (x + y, x - y)
;;
```

2.6 Recursion

The following is an attempt to define the factorial function:

9

```
let fact n =
  if n=0
  then 1
  else n*fact(n-1);;
```

Given this piece of code the type checker returns the following error message:

```
[ERROR]
kind:  fact is a free identifier
slice: ⟨..fact;;..⟩
```

The problem is that any free variables in the body of a function have the bindings they had just before the function was declared; `fact` is such a free variable in the body of the declaration above, and since it is not defined before its own declaration, an error results. To make things clear consider:

```
# f 3;;
9 : Int
```

10

```
let f n = n + 1;;

let f n =
  if n = 0
  then 1
  else n * f(n - 1);;
```

Here `f 3` results in the evaluation of `3 * f(2)`, but now the first `f` is used so `f(2)` evaluates to `2 + 1 = 3`, hence the expression `f 3` results in `3 * 3 = 9`. To make a function declaration hold within its own body, `letrec` instead of `let` must be used. The correct recursive definition of the factorial function is thus:

```
# fact 3;;
6 : Int
```

11
```
letrec fact n =
  if n = 0
  then 1
  else n * fact(n - 1);;
```

2.7 Lists

If $e_1, \ldots, e_n$ all have type ty then the ML expression `[`e_1`;...;`e_n`]` has type `(`ty `List)`. The standard functions on lists are `hd` (head), `tl` (tail), `null` (which tests whether a list is empty – i.e. is equal to `[]`), and the infixed operators `.` (cons) and `++` (append, or concatenation). The functions `hd`, `tl`, and `null` need to be imported to be usable.

12
```
import tl hd null;;
let m = [1;2;(2 + 1);4];;
```

```
# m;;
m = [1; 2; 3; 4] : Int List

# (hd m , tl m);;
(1, [2; 3; 4]) : Int * (Int List)

# (null m , null []);;
(false, true) : Bool * Bool

# 0.m;;
[0; 1; 2; 3; 4] : Int List

# [1; 2] ++ [3; 4; 5; 6];;
[1; 2; 3; 4; 5; 6] : Int List

# [1;true;2];;
[ERROR]
kind:  type constructor clash between
       Int and Bool
slice: ⟨..[1; true; ⟨..⟩];;..⟩

[ERROR]
kind:  type constructor clash between
       Bool and Int
slice: ⟨..[⟨..⟩; true; 2];;..⟩

Untypable.
```

All the members of a list must have the same type (although this type could be a sum, or disjoint union type—see Section 4).

2.8 Atom

A non-empty sequence of characters enclosed between token backquotes (‘ – i.e. ascii 96) is an *atom* (also called a *token*).

```
# t;;
‘an atom‘ : Atom

# ts;;
[‘an‘; ‘atom‘; ‘list‘] : Atom List

# ts = ‘‘an atom‘‘ ++ [‘list‘];;
true : Bool
```

13
```
let t  = ‘an atom‘;;
let ts = ‘‘an atom list‘‘;;
```

The expression ‘‘$\texttt{atom}_1\texttt{atom}_2 \ldots \texttt{atom}_n$‘‘ is an alternative syntax for [‘$\texttt{atom}_1$‘;‘$\texttt{atom}_2$‘;...;‘$\texttt{atom}_n$‘].

2.9 Polymorphism

The list processing functions hd, tl etc. can be used on all types of lists.

```
# hd [1;2;3];;
1 : Int

# hd [true;false;true];;
true : Bool

# hd [(1,2);(3,4)];;
(1, 2) : Int * Int
```

14
```
import hd;;
```

Thus hd has several types; for example, it is used above with types (Int List) -> Int, (Bool List) -> Bool, and (Int * Int) List -> (Int * Int). In fact if *ty* is *any* type then hd has the type (*ty* List) -> *ty*. Functions, like hd, with many types are called *polymorphic*, and ML uses type variables 'a, 'b, 'c etc. to represent their types[1].

```
# hd;;
- : 'a List -> 'a

# map;;
- : ('a -> 'b) -> ('a List) -> 'b List

# map fact [1;2;3;4];;
[1; 2; 6; 24] : Int List
```

15
```
import null;;

letrec fact n =
  if n = 0
  then 1
  else n * fact(n - 1);;

letrec map f l =
  if null l
  then []
  else f(hd l).map f (tl l);;
```

The ML function map takes a function f (with argument type 'a and result type 'b), and a list l (of elements of type 'a), and returns the list obtained by applying f to each element of l (which is a list of elements of type 'b). The function map can be used at any instance of its type: above, both 'a and 'b were instantiated to Int; below, 'a is instantiated to (Int List) and 'b to Bool. Notice that the instance need not be specified; it is determined by the type checker.

```
# map null [[1;2]; []; [3]; []];;
[false; true; false; true] : Bool List
```

16
```
import null map;;
```

[1] EventML, SML and other such ML-like programming languages use 'a, 'b, 'c, etc. to represent type variables, while Classic ML uses *, **, *** etc.

2.10 Lambda-expressions

The expression `\`x`.`e evaluates to a function with formal parameter x and body e. Thus the declaration `let` f x `=` e is equivalent to `let` f `=` `\`x`.`e. Similarly `let` f`(`x`,`y`)` z `=` e is equivalent to `let` f `=` `\(`x`,`y`).\`z`.`e. The character \ is our notation for a lambda. The expressions `\`x`.`e and `\(`x`,`y`).\`z`.`e are called lambda-expressions. Lambda-expressions bind less tightly than anything else in the language. For example, `\`x`. (\`y`. `y`) 1` means `\`x`. ((\`y`. `y`) 1)` not `(\`x`. (\`y`. `y`)) 1`

```
# plus1;;
- : Int -> Int

# plus1 3;;
4 : Int

# map (\x.x * x) [1;2;3;4];;
[1; 4; 9; 16] : Int List

# doubleup;;
- : ('a List List) -> 'a List List

# doubleup [ [1]; [2;3] ];;
[[1; 1]; [2; 3; 2; 3]] : Int List List

# doubleup [];;
[] : 'a List List
```

17

```
import map;;

let plus1 = \x. x + 1;;

let doubleup = map (\x.x ++ x)
;;
```

2.11 Type constructors

Both `List` and `*` are examples of type constructors; `List` has one argument (hence `'a List`) whereas `*` has two (hence `'a * 'b`). Each type constructor has various primitive operations associated with it, for example `List` has `null`, `hd`, `tl`, ... etc, and `*` has `fst`, `snd`.

```
# z;;
(8, 30) : Int * Int

# fst z;;
8 : Int

# snd z;;
30 : Int
```

18

```
let z = (8,30);;
```

Another standard constructor of two arguments is `+`; `'a + 'b` is the disjoint union of types `'a` and `'b`, and associated with it are the following primitives:

`isl : ('a + 'b) -> bool`	tests membership of left summand
`inl : 'a -> ('a + 'b)`	injects into left summand
`inr : 'a -> ('b + 'a)`	injects into right summand
`outl : ('a + 'b) -> 'a`	projects out of left summand
`outr : ('a + 'b) -> 'b`	projects out of right summand

These are illustrated by:

```
# x;;
inl(1) : Int + 'a

# y;;
inr(2) : 'a + Int

# isl x;;
true : Bool

# isl y;;
false : Bool

# outl x;;
1 : Int

# outl y;;
evaluation failed.

# outr x;;
evaluation failed.

# outr y;;
2 : Int
```

19

```
let x = inl 1;;
let y = inr 2;;
```

3 Syntax of ML

We shall use variables to range over the various constructs of ML as follows:

Variable	Ranges over
var, id	variables
con	constructors
ce	constant expressions
ty	types
d	declarations
b	bindings
p	patterns
e	expressions

Variables and constructors are both represented by identifiers but they are different syntax classes. Identifiers and constant expressions are described in Section 3.2 below. Types and type-bindings are explained in Section 4. Declarations, bindings, patterns and expressions are defined by the following BNF-like syntax equations in which:

1. Each variable ranges over constructs as above.
2. The numbers following the various variables are there merely to distinguish between different occurrences.

3. $\{C\}$ denotes an optional occurrence of C, and for $n>1$ $\{C_1|C_2\ldots|C_n\}$ denotes a choice of exactly one of C_1,C_2,...,C_n.
4. The constructs are listed in order of decreasing binding power.
5. 'L' or 'R' following a construct means that it associates to the left (L) or right (R) when juxtaposed with itself (where this is syntactically admissible).
6. Certain constructs are equivalent to others and this is indicated by 'equiv.' followed by the equivalent construct.

3.1 Syntax equations for ML

Table 1 describes ML declarations, Table 2 bindings, Table 3 patterns, and Table 4 on page 11 describes expressions.

d	::=	`let` b	ordinary variables
	|	`letrec` b	recursive functions
	|	`infix` {{0|1|2|3|4|5|6|7}} id	left associative infix directive
	|	`infixr` {{0|1|2|3|4|5|6|7}} id	right associative infix directive
	|	`import` id_1 ... id_n	declaration import

Table 1: Declarations

b	::=	p`=`e	simple binding
	|	id p_1 p_2 ... p_n {`:`ty} `=` e	function definition

Table 2: Bindings

p	::=	`()`		unit
	|	`_`		empty pattern (widcard)
	|	id		variable
	|	p`:`ty		type constraint
	|	(`(`p_1`,`p_2`)`	R	pairing
	|	(`(`p`)`		equivalent to p

Table 3: Patterns

In the syntax equations constructs are listed in order of decreasing binding power. For example, since e_1e_2 is listed before e_1 `+` e_2 function application binds more tightly than addition and thus e_1e_2 `+` e_3 parses as (`(`e_1e_2`)` `+` e_3. This convention determines only the relative binding power of different constructs. The left or right association of a construct is indicated explicitly by 'L' for left and 'R' for right. For example, as application associates to the left, the expression $e_1e_2e_3$ parses as (`(`e_1e_2`)` e_3.

Only functions can be defined with `letrec`. For example, `letrec x = 2 - x` would cause a syntax error.

All the variables occurring in a pattern must be distinct. On the other hand, a pattern can contain multiple occurrences of the unit pattern `()` or of the wildcard `_`.

e	::=	ce		constant
	|	var		variable
	|	$e_1\ e_2$	L	function application
	|	e`:`ty		type constraint
	|	`!`e		negation
	|	$\sim e$		unary minus
	|	e_1 `*` e_2	L	multiplication
	|	e_1 `/` e_2	L	division
	|	e_1 `+` e_2	L	addition
	|	e_1 `-` e_2	L	subtraction
	|	e_1 `.` e_2	R	list cons
	|	e_1 `++` e_2	R	list append
	|	e_1 `=` e_2	L	equality
	|	e_1 `<` e_2		less than
	|	e_1 `>` e_2		greater than
	|	e_1 `<=` e_2		less or equal to
	|	e_1 `>=` e_2		greater or equal to
	|	e_1 `&` e_2	R	conjunction
	|	e_1 `or` e_2	R	disjunction
	|	e_1 *user-infix* e_2	L	user declared infix identifier
	|	`(`e_1`,`e_2`)`	R	pairing
	|	`if` e_1 `then` e_2 `else` e_3		conditional
	|	`[]`		empty list
	|	`[`e_1`;`e_2 ... `;`e_n`]`		list of n elements
	|	e `where` b	R	equivalent to `let` b `in` e
	|	`let` d `in` e		local declaration
	|	$\backslash\ p . e$		abstraction
	|	`(`e`)`		equivalent to e

Table 4: Expressions

Spaces (ASCII 32), carriage returns (ASCII 13), line feeds (ASCII 10) form feeds (`^L`, ASCII 12) and tabs (`^I`, ASCII 9) can be inserted and deleted arbitrarily without affecting the meaning (as long as obvious ambiguities are not introduced). For example, the space in $\sim\ x$ but not in x `or` y can be omitted. *Comments*, which are arbitrary sequences of characters surrounded by `(*` and `*)`, can be inserted anywhere a space is allowed. Comments can be nested.

3.2 Identifiers and other lexical matters

In this section the lexical structure of ML is defined.

3.2.1 Identifiers

A variable (*var*) or *identifier* is a sequence of alphanumerics starting with a letter, where an alphanumeric is either a letter, a digit, a prime (`'`), a dash (`-`) or an underbar (`_`). ML is case-sensitive: upper and lower case letters are considered to be different.

3.2.2 Constant expressions

The ML constant expressions (*ce*'s) used in Table 4 are:

1. Integers, i.e. sequences of digits `0,1...,9`.
2. Truth values `true` and `false`.
3. Tokens and token-lists:
   (a) Tokens consist of any non-empty sequence of characters surrounded by token backquotes (`‘`), e.g. `‘This is a single token‘`.
   (b) Token-lists consist of any sequence (possible empty) of tokens (separated by spaces, returns, line-feed or tabs) surrounded by double backquotes (`‘‘`). e.g.
   `‘‘this is a token-list containing 7 members‘‘`.
   The token list `‘‘ tok1 tok2 ... tokn‘‘` is equivalent to `[‘tok1‘; ‘tok2‘; ...; ‘tokn‘]`.
4. The expression `()`, called *unit*, which evaluates to the unique object of ML type `Unit`.

3.2.3 Prefixes and infixes

Each infix operator has a precedence rated from 0 to 7 (a infix operator with precedence i binds tighter than an operator with precedence j if $j < i$), and is either left or right associative.

The ML *prefixes* px and *infixes* ix are given by:

```
px  ::=  ! | ~
ix  ::=  * | /                          (L, 7)
      |  + | -                          (L, 6)
      |  . | ++                         (R, 5)
      |  = | < | > | <= | >=            (L, 4)
      |  & | or                         (R, 0)
```

In addition, any identifier (and certain single characters) can be made into an infix.

Except for `&` and `or`, each prefix *px* has correlated with it a special identifier `op` *px* which is bound to the associated function. For example, the identifier `op +` is bound to the addition function, and `op ++` to the list-append function. This is useful for passing functions as arguments; for example, *f*`(op ++)` applies *f* to the append function.

4 ML Types

So far, little mention has been made of types. For ML in its original role as the meta-language for proof in LCF, the importance of strict type checking was principally to ensure that every computed value of the type representing theorems was indeed a theorem.

The same effect could probably have been achieved by run-time type checking, but compile-time type checking was adopted instead, in the design of ML. This was partly for the considerable debugging aid that it provides; partly for efficient execution; and partly to explore the possibility of combining polymorphism with type checking. This last reason is of general interest in programming languages and has nothing to do specifically with proof; the problem is that there are many operations (list mapping functions, functional composition, etc.) which work at an infinity of types, and therefore their types should somehow be parameterized – but it is rather inconvenient to have to mention the particular type intended at each of their uses.

The ML type checking system is implemented in such a way that, although the user may occasionally (either for documentation or as a constraint) ascribe a type to an ML expression or pattern, it is hardly ever necessary to do so. The user of ML will almost always be content with the types ascribed and presented by the type checker, which checks every top-level phrase before it is evaluated. (The type checker may sometimes find a more general type assignment than expected.)

4.1 Types and objects

Every data object in ML possesses a type. Such an object may possess many types, in which case it is said to be *polymorphic* and possesses a *polytype* – i.e. a type containing type variables (for which we use an identifier followed by at least one quote) – and moreover it possesses all types which are *instances* of its polytype, formed by substituting types for zero or more type variables in the polytype. A type containing no type variables is a *monotype.*

We saw several examples of types in Section 2. To understand the following syntax, note that `List` is a postfixed unary (one-argument) type constructor (thereafter abbreviated to *tycon*). The user may introduce new *n*-argument type constructors. A binary type operator `directory`, for example, can be introduced. The following type expressions will then be types of different kinds of `directory`:

- `(Atom, Int) directory`
- `(Int, Int -> Int) directory`

The user may even deal with lists of directories, with the type `(Int, Bool) directory List`

4.1.1 The syntax of types

The syntax of ML types is summarized in Table 5. Two or more arguments must be separated by commas and enclosed by parentheses. The type operator `List` is a predeclared unary type operator; and `*`, `+` and `->` may be regarded as infix forms of three predeclared binary type operators.

Types	*ty*	::=	*sty*		Standard (non-infix) type
		|	*ty* `*` *ty*	R	Cartesian product
		|	*ty* `+` *ty*	R	Disjoint sum
		|	*ty* `->` *ty*	R	Function type
Standard Types	*sty*	::=	`Unit` | `Int`		
		|	`Bool` | `Atom`		Basic types
		|	*vty*		Type variable
		|	`(`*ty*`)`		
Type arguments	*tyarg*	::=	*sty*		Single type argument
		|	`(`*ty*`,...,`*ty*`)`		One or more type arguments
Type variables	*vty*	::=	`'`*id*		

Table 5: ML Type Syntax

For an object to possess[2] a type means the following: For basic types, all integers possess `Int`, both booleans (`true` and `false`) possess `Bool` , etc. The only object possessing `Unit` is that denoted by `()` in ML. For a type abbreviation *tycon*, an object possesses *tycon* (during execution of phrases in the scope of the declaration of *tycon*) if and only if it possesses the type which *tycon* abbreviates. For compound monotypes ,

1. The type *ty* `List` is possessed by any list of objects, all of which possess type *ty* (so that the empty list possesses type *ty* `List` for every *ty*).
2. The type ty_1 `*` ty_2 is possessed by any pair of objects possessing the types ty_1 and ty_2, respectively.
3. The type ty_1 `+` ty_2 is possessed by the left-injection of any object possessing ty_1, and by the right-injection of any object possessing ty_2. These injections are denoted by the ML function identifiers `inl : 'a -> 'a + 'b` and `inr : 'b -> 'a + 'b`.
4. A function possesses type ty_1 `->` ty_2 if, whenever its argument possesses type ty_1, its result (if defined) possesses type ty_2. (This is not an exact description; for example, a function defined in ML with non-local variables may possess this type even though some assumption about the types of the values of these non-local variables is necessary for the above condition to hold. The constraints on programs listed below ensure that the non-locals will always have the right types).

Finally, an object possesses a polytype *ty* if and only if it possesses all monotypes which are substitution instances of *ty*.

4.2 Typing of ML phrases

We now explain the constraints used by the type checker in ascribing types to ML expressions, patterns and declarations.

The significance of expression *e* having type *ty* is that the value of *e* (if evaluation terminates successfully) possesses type *ty*. As consequences of the well-typing constraints listed below, it is impossible for example to apply a non-function to an argument, or to form a list of objects

[2]We shall talk of objects *possessing* types and phrases *having* types, to emphasize the distinction.

of different types, or (as mentioned earlier) to compute an object of the type corresponding to theorems which is not a theorem.

The type ascribed to a phrase depends in general on the entire surrounding ML program. In the case of top-level expressions and declarations, however, the type ascribed depends only on preceding top-level phrases. Thus you know that types ascribed at top-level are not subject to further constraint.

Before each top-level phrase is executed, types are ascribed to all its sub-expressions, sub-declarations and sub-patterns according to the following rules. Most of the rules are fairly natural; those which are less so are discussed later. You are only presented with the types of top-level phrases; the types of sub-phrases will hardly ever concern you.

Before giving the list of constraints, let us discuss an example which illustrates some important points. To map a function over a list we may define the polymorphic function `map` recursively as follows (where we have used an explicit abstraction, rather than `letrec map f l = ...`, to make the typing clearer):

```
letrec map = \f.\l. null l => [] | f(hd l).map f(tl l) ;;
```

From this declaration the type checker will infer a *generic* type for `map`. By 'generic' we mean that each later occurrence of `map` will be ascribed a type which is a substitution instance of the generic type.

Now the free identifiers in this declaration are `null`, `hd`, `tl` and `op .`, which are ML primitives whose *generic* (poly)types are `'a List -> Bool`, `'a List -> 'a`, `'a List -> 'a List`, and `'a * 'a List -> 'a List` respectively. The first constraint used by the type checker is that the occurrences of these identifiers in the declaration are ascribed instances of their generic types. Other constraints which the type checker will use to determine the type of `map` are:

- All occurrences of a lambda-bound variable receive the same type.
- Each arm of a conditional receives the same type, and the condition receives type `Bool`.
- In each application e `=` `(`$e_1 e_2$`)`, if e_2 receives ty and e receives ty' then e_1 receives ty `->` ty'.
- In each abstraction e `=` `\`v`.`e_1, if v receives ty and e_1 receives ty' then e receives ty `->` ty'.
- In a `letrec` declaration, all free occurrences of the declared variable receive the same type.

Now the type checker will ascribe the type `('a -> 'b) -> 'a List -> 'b List` to `map`. This is in fact the most general type consistent with the constraints mentioned. Moreover, it can be shown that any instance of this type also allows the constraints to be satisfied; this is what allows us to claim that the declaration is indeed polymorphic.

In the following constraint list, we say p has ty to indicate that the phrase p is ascribed a type ty which satisfies the stated conditions. We use x, p, e, d to stand for variables, patterns, expressions and declarations respectively.

Constants:

1. `()` has type `Unit`
2. `0` has type `Int` , `1` has type `Int`, ...
3. `true` has type `Bool`, `false` has type `Bool`
4. `` `...` `` has type `Atom`

Variables and constructors: The constraints described here are discussed in Section 4.3 below.

1. If x is a variable bound by \ or `letref`, then x is ascribed the same type as its binding occurrence.
2. If x is bound by `let` or `letrec`, then x has ty, where ty is an instance of the type of the binding occurrence of x (i.e. the *generic* type of x), in which type variables occurring in the types of current lambda-bound are not instantiated.
3. If x is not bound in the program (in which case it must be an ML primitive), then x has ty, where ty is an instance of the type of x has defined in the built-in library or in the library file.

Patterns: Cases for a pattern p:

- `()`: p has `Unit`.
- _: p has ty, where ty is any type.
- p_1`:`ty:
  p_1 and p have an instance of ty.
- `(`p_1`,`p_2`)`: If p_1 has ty_1 and p_2 has ty_2, then p has ty_1 `*` ty_2.

Expressions: Cases for an expression e (not a constant or identifier):

- e_1e_2: If e_2 has ty and e has ty' then e_1 has ty `->` ty'.
- e_1`:`ty: e_1 and e have an instance of ty.
- px e_1: Treated as `(op` px`)`e_1 when px is a prefix. If e is $\sim e_1$, then e and e_1 have `Int`.
- e_1 ix e_2: Treated as `(op` ix`)`e_1e_2. If e is `(`e_1 `&` e_2`)` or `(`e_1 `or` e_2`)` then e, e_1 and e_2 have `Bool`.
- `(`e_1`,`e_2`)`: If e_1 has ty_1 and e_2 has ty_2 then e has ty_1 `*` ty_2.
- `if` e_1 `then` e_2 `else` e_3: Each e_1 has `Bool`, and e_2 and e_3 have ty for some ty.
- `[`e_1`;...;`e_n`]`: For some ty, each e_i has ty and e has ty `List`.
- `let` d `in` e_1: If e_1 has ty then e has ty.
- \p.e_1: If p has ty and e_1 has ty' then e has ty `->`ty'.

Declarations:

1. Each binding x $p_1 \ldots p_n$ `=` e is treated as x `=` \$p_1$. `...`\$p_n$.$e$.
2. If d is `let` p`=`e, then d, p and e all have ty for some ty.Note that e is not in the scope of the declaration.
3. If d is `letrec (`x_1`,...,`x_n`) = (`e_1`,...,`e_n`)`, then, for each $i \in \{1,\ldots,n\}$, x_i and e_i have ty_i, and d has ty_1 `*` `...` `*` ty_n for some ty_i. In addition, each free occurrence of x_i in e_1,...,e_n has ty_i, so that the type of recursive calls of x_i is the same as the declaring type.

4.3 Discussion of type constraints

We give here reasons for our constraints on the types ascribed to occurrences of identifiers. The reader may like to skip this section at first reading.

Consider constraint (1) for lambda-bound identifiers. This constraint implies that the expression `let` x `=` e `in` e' may be well-typed even if the semantically-equivalent (dynamic semantics and not static semantics) expression `let` f x `=` e' `in` f e is not, since in the former expression x may occur in e' with two incompatible types which are both instances of the declaring type. The greater constraint on f is associated with the fact that f may be applied to many different arguments during evaluation. To show the need for the constraint, suppose that it is replaced by the weaker constraint for `let`-bound identifiers, so that for example `let f x = if x then 1 + x else x(1)` is a well-typed declaration of type `'a -> Int`, in which the occurrences of `x` receive types `'a`, `Bool`, `Int`, `Int -> Int` respectively. In the absence of an explicit argument for the abstraction, no constraint exists for the type of the binding occurrence of `x`. But, because `f` is `let`-bound, expressions such as `f true` and `f 'dog'` are admissible in the scope of `f`, although their evaluation should result in either nonsense or *run-time* type-errors; one of the our purposes is to preclude these.

Note that expressions of the form `(\`x`.`e'`)`e, could be treated exactly as `let` x`=`e `in` e' because we know the unique instance of type of the argument x, namely the type of e.

References

[ABC+06] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computational type theory using nuprl. *J. Applied Logic*, 4(4):428–469, 2006.

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development*. SpringerVerlag, 2004.

[BC08] Mark Bickford and Robert L. Constable. Formal foundations of computer security. In *NATO Science for Peace and Security Series, D: Information and Communication Security*, volume 14, pages 29–52. 2008.

[BCG11] Mark Bickford, Robert Constable, and David Guaspari. Generating event logics with higher-order processes as realizers. Technical report, Cornell University, 2011.

[BCH+00] Ken Birman, Robert Constable, Mark Hayden, Jason Hickey, Christoph Kreitz, Robbert van Renesse, Ohad Rodeh, and Werner Vogels. The Horus and Ensemble projects: Accomplishments and limitations. In *In DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 149–160. IEEE Computer Society Press, 2000.

[Bic09] Mark Bickford. Component specification using event classes. In Grace A. Lewis, Iman Poernomo, and Christine Hofmeister, editors, *CBSE*, volume 5582 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 2009.

[CAB+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[CH90] G. Cousineau and Gérard Huet. The CAML primer. Technical Report RT-0122, INRIA, September 1990.

[CHP84] Guy Cousineau, Gérard Huet, and Larry Paulson. *The ML handbook*, 1984.

[GM93] Michael Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher-order logic*. Cambridge University Press, 1993.

[GMW79] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation.*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Hay98] Mark Hayden. *The Ensemble System*. PhD thesis, Cornell University, Department of Computer Science, 1998. Technical Report TR98-1662.

[Kre02] Christoph Kreitz. *The Nuprl Proof Development System, Version 5, Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 2002. `http://www.nuprl.org/html/02cucs-NuprlManual.pdf`.

[Ler00] Xavier Leroy. *The Objective Caml system release 3.00*. Institut National de Recherche en Informatique et en Automatique, 2000.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David Macqueen. *The Definition of Standard ML (Revised)*. MIT Press, Cambridge, MA, USA, 1997.

[Pau87] Lawrence C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Cambridge University Press, New York, NY, USA, 1987.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. *In Piergiorgio Odifreddi, editor, Logic and Computer Science*, pages 361–386, 1990.

[SGC07] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# (Expert's Voice in .Net)*. Apress, 2007.

[WAL+90] Pierre Weis, Maria-Virginia Aponte, Alain Laville, Michel Mauny, and Acsander Suarez. *The CAML reference manual*, September 1990.