ML under Unix

Luca Cardelli

ABSTRACT

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

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

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

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

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

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

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 The system keeps track of module versions to detect compiled modules which are out of date.

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

Contents

August 14, 1983

1. Introduction

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

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

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

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

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

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

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

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

1.1. Expressions

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

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

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

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

result:         16 : int
```

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

```
expression:     "this is it";

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

Pairs of values are built by an infix pairing operator ‘,’ (comma, which associates to the right); the type of a pair is given by the infix type operator #, which denotes cartesian products.

```
expression:     3,4;

result:         3,4 : int # int

expression:     3,4,5;

result:         3,(4,5) : int # (int # int)
```

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

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

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

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

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

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

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

result:         3 : int

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

result:         false : bool
```

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

1.2. Declarations

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

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

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

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

expression:     (a + b) div c;

result:         4 : int
```

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

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

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

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

binding:            val f : int -> int

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

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

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

result:         4,6 : int # int
```

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

The function f has one argument x. The result of a function is just the value of its body, in this case x+1. Arguments to functions do not need to be parenthesized in general (both in definitions and applications): the simple juxtaposition of two expressions is interpreted as a function application. Function application is treated as an invisible infix operator, and it is the strongest-binding infix operator; expressions like f 3 + 4 are parsed like (f 3) + 4 and not like f (3 + 4).

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

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

```
expression:     f, f 3;
```

```
result:          fun,4 : (int -> int) # int

declaration:     val h = g;

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

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

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

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

binding:               val f : int -> int

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

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

expression:      f 1;

result:          4 : int
```

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

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

1.3. Local scopes

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

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

result:          4 : int
```

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

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

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

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

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

result:         13,3 : int
```

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

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

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

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

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

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

result:         3,6: int # int
```

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

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

result:         3,6: int # int
```

1.4. Functions

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

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

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

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

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

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

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

```
declaration:    val plus x = (fst x) + (snd x);
```

where fst and snd are primitive functions which extract the first and second element of a pair.

What we did in the previous definition of plus was to use a *pattern* (a,b) for x, which implicitly associated a to fst x and b to snd x. These patterns come in many forms. For example a pattern [a;b;c] matches a list of exactly three elements, which are bound to a, b and c; a pattern first::rest matches a non-empty list whose first element is associated to first, and the rest to rest; similarly first::second::rest matches a list of at least two elements; etc. The most common patterns are of course tuples, like (a,b,c), but more complicated patterns can be constructed by nesting, like ([a;b],c,(d,e)::$). The special pattern $ matches any value without estabilishing any binding. Patterns can conveniently replace the use of selectors like fst and snd for pairs, and the analogous hd and tl for lists.

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

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

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

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

In the above example we have a function f returning three values, and the pattern a,b,c is used to unpack the result of f[1;2;3] into its components. A moderate use of patterns is considered good programming style, and also happens to be more efficient than data selectors in unpacking data structures.

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

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

binding:                val factorial : int −> int
```

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

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

---

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

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

1.5. Polymorphism

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

```
declaration:    val rec length list =
                        if null list
                        then 0
                        else 1 + length(tl l);

binding:                val length : 'a list -> int

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

result:         3,4 : int#int
```

where null is a test for empty list, and tl (tail) returns a list without its first element.

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

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

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

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

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

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

result:         "abc",[ ] : string # 'a list
```

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

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

Several primitive functions are polymorphic; for example we have already encountered the pair selectors fst and snd:

```
fst     : ('a # 'b) -> 'a
snd     : ('a # 'b) -> 'b
```

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

```
expression:     [];

result:         [] : 'a list

expression:     fst;

result:         fun : ('a # 'b) -> 'a
```

The primitive operators on lists are also polymorphic. Note that if this were not the case, we would need different primitive operators for all possible types of list elements!

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

We have already seen null and tl at work. hd (head) extracts the first element of a list. :: (cons, which is infix) appends a new element to the beginning of a list; note that the 'a shared by its two arguments prevents any attempt to build lists containing objects of different types. @ (append, also infix) concatenates two lists. Here are some examples:

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

1.6. Higher-order functions

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

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

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

expression:     times 3 4;

result:         12 : int

declaration:    val twice = times 2;

binding:            val twice : int -> int

expression:     twice 4;

result:         8 : int
```

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

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

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

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

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

binding:            val fourtimes : int -> int

expression:     fourtimes 5;

result:         20 : int
```

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

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

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

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

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

declaration:    val curryplus = curry plus;

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

declaration:    val successor = curryplus 1;

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

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

1.7. Failures

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

```
expression:     1 div 0;

failure:            Failure: div
```

Another common case of failure is trying to extract the head of an empty list:

```
expression:     hd [];

failure:            Failure: hd
```

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

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

```
expression:     hd [] ? 3;

result:         3 : int
```

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

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

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

result:         3 : int

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

failure:            Failure: div
```

The user can generate failures by the failwith keyword followed by a string, which is the failure reason. Moreover fail is an abbreviation for failwith "fail":

```
expression:     hd [] ? failwith "emptylist";

failure:            Failure: emptylist

expression:     fail;

failure:            Failure: fail
```

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

1.8. Types

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

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

On the other hand, ML types can express structural dependencies inside objects, for example that the right part of a pair must have the same type (whatever that type is) as the left part of the pair, or that a function

must return an object of the same type of its argument (whatever that type may be).

Types like *bool*, *int*, and *string*, which do not show any internal structure, are called *basic*. Compound types are built by *type operators* like # (cartesian product), list (list) and −> (function space). Type operators are usually infix or suffix: int # int, int list and int −> int are the types of integer pairs, lists and functions. Strictly speaking, basic types are type operators which take no arguments. Type operators can be arbitrarily nested: e.g. ((int −> int) list) list is the type of lists of lists of integer functions.

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

Every type denotes a *domain*, which is a set of objects, all of the given type; for example int # int is (i.e. denotes) the domain of integer pairs, and 'a −> 'b is the domain of all functions. An object can have several types, i.e. it can belong to several domains. For example the identity function fun x. x has type 'a −> 'a as it maps any object (of type 'a) to itself (of type 'a), but it is also a function and has a type 'a −> 'b mapping any object (of type 'a) to some other object (of type 'b, in general). It is clear that 'a −> 'a gives more information that 'a −> 'b, because the former is a smaller domain that the latter. Hence 'a −> 'a is a 'better' type for fun x. x, although 'a −> 'b is not wrong. The ML typechecker always determines the 'best' type for an expression, i.e. the one which corresponds to the smallest domain, given the information contained in that expression.

Types can be wrapped around any expression, in order to *specify* the type of that expression:

```
expression:     3 : int;

result:         3 : int

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

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

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

```
expression:     3 : bool;

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

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

```
expression:     [] : int list;

result:         [] : int list

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

result:         fun : int −> int
```

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

Type specifications can be used in bindings (parentheses are needed around the outmost type specifications of non-functions):

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

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

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

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

New type operators can be introduced as abbreviations of more complex type expressions:

```
declaration:    type 'a pair = 'a # 'a;

binding:            type 'a pair = 'a # 'a

expression:     (3,4) : int pair;

result:         3,4 : int pair
```

The type 'a pair is now completely equivalent to 'a # 'a. The system tries to use **t** pair to print out the type of a value declared to be a pair, but in some circumstances it will print the type as **t # t**.

Type definitions cannot be recursive (only abstract types can be recursive). In a type declaration, the type variables on the right of an = must all appear on the left. The left hand side must be a list of distinct type variables followed by a type identifier (infix and prefix type operators can also be defined: see section "Lexical declarations"). Here are type operators with zero, one and two type parameters:

```
declaration:    type point = int # int
                and 'a predicate = 'a −> bool
                and ('a,'b) leftprojection = ('a # 'b) −> 'a;

bindings:           type point = int # int
                    type 'a predicate = 'a −> bool
                    type ('a,'b) leftprojection = ('a # 'b) −> 'a
```

1.9. Abstract data types

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

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

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

```
declaration:    type color <=> int # int # int
                with val white  = abscolor(0,0,0)
                and red = abscolor(15,0,0)
```

```
                and blue = abscolor(0,15,0)
                and yellow = abscolor(0,0,15)
                and mix (parts: int, color: color, parts': int, color': color) : color =
                        if parts < 0 or parts' < 0 then failwith "mix"
                        else    let val red,blue,yellow = repcolor color
                                and red',blue',yellow' = repcolor color'
                                and totalparts = parts + parts'
                                in abscolor(
                                        (parts*red + parts'*red') div totalparts,
                                        (parts*blue + parts'*blue') div totalparts,
                                        (parts*yellow + parts'*yellow') div totalparts))
                                end
                end;

bindings:               type color = −
                        val white = − : color
                        val red = − : color
                        val yellow = − : color
                        val mix : (int # color # int # color) −> color
```

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

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

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

The sign <=>, read *isomorphism*, (instead of =) is characteristic of abstract type definitions. An abstract type is a type isomorphic to some concrete data type (in this case int # int # int). The isomorphism is given by two functions abscolor : int # int # int −> color and repcolor : color −> int # int # int which are respectively the basic constructor and destructor of colors from and to their concrete representation. In general, for an abstract type **T**, abs**T** and rep**T** are available in the with part of the definition, where the abstract type operations are defined. However abs**T** and rep**T** are not known outside the abstract type definition, thus protecting the privacy of the concrete representation.

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

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

```
expression:     fst pink

type error:             Type Clash in:      fst pink
                        Looking  for :      'a # 'b
                        I have found :color
```

The correct thing to do is fst(repcolor pink), but only where repcolor is available.

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

1.10. Interacting with the ML system

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

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

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

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

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

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

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

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

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

− a + 1;
   4 : int
```

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

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

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

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

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

```
− 1 + 1;
  2 : int

− it;
  2 : int

− it + 1;
  3 : int

− val a = 5;
> val a = 5 : int

− it;
  3 : int

− it + z;
Unbound Identifier: z

− it;
  3 : int

− 1 div 0;
Failure: div

− it,it;
  3,3 : int # int
```

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

1.11. Errors

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

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

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

```
− noway;
Unbound Identifier: noway

− 3 : noway;
```

```
Unbound Type Identifier: noway

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

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

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

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

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

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

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

2. Lexical matters

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

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

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

Integers are sequences of digits.

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

Type variables are only accepted in type expressions. They are identifiers starting with, and possibly also containing, the character ‘’’. For example: ’, ’’’, ’1 ’tyvar, etc.

Delimiters are one-character punctuation marks like ‘.’ and ‘;’ which never stick to any other character. No space is ever needed around them.

Left parentheses are (and [; right parentheses are) and]. Moreover, some characters stick to the inner side of parentheses to form compound parentheses like (| and |). See appendix "Syntax of lexical entities" for details.

Comment are enclosed in curly brackets ‘{’ and ‘}’, and can be nested.

3. Expressions

Expressions denote values, and have a type which is statically determined. Expressions can be constants, identifiers, pairs, lists, records, variants, conditionals, fun-expressions, function applications, operator applications (prefix, infix or suffix), scope blocks, while expressions, case expressions, failure and trap expressions, and type specifications. All these different kinds of expressions are described in this section.

3.1. Unit

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

```
()          : unit
```

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

3.2. Booleans

The predefined identifiers

```
true, false          : bool
```

denote the respective boolean values. Boolean operators are

```
not                  : bool −> bool                (prefix)
  &, or              : (bool # bool) −> bool       (infix)
```

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

3.3. Integers

The integer operators are:

```
~                          : int −> int                    (prefix)
+, −, *, div, mod          : (int # int) −> int            (infix)
>, >=, <=, <               : (int # int) −> bool           (infix)
```

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

3.4. Pairs

The operations on pairs are:

```
,          : (’a # ’b) −> (’a # ’b)          (infix)
fst        : (’a # ’b) −> ’a
snd        : (’a # ’b) −> ’b
```

where for example 3,4: int#int is a pair of two numbers whose fst is 3 and snd is 4.

3.5. Lists

A list of elements $\mathbf{e_i}$ is written:

syntax: [$\mathbf{e_1}$; .. ; $\mathbf{e_n}$] n≥0

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

The basic operations on lists are:

```
null        : 'a list −> bool
hd          : 'a list −> 'a
tl          : 'a list −> 'a list
::          : ('a # 'a list) −> 'a list          (infix)
```

• null tests whether a list is empty.
• hd (head) takes the first element of a list and tl (tail) returns a list where the first element has been removed; hd and tl fail when applied to an empty lits, respectively with strings "hd" and "tl".
• :: (cons) adds an element to a list (e.g. 1::[2;3;4] is the same as [1;2;3;4]).

Some other frequent operations on list are predefined in the system. They can all be defined in terms of the four basic operations above and []:

```
length          : ('a list) −> int
@               : ('a list # 'a list) -> 'a list          (infix)
rev             : ('a list) −> ('a list)
map             : ('a −> 'b) −> (('a list) −> ('b list))
fold, revfold   : (('a # 'b) −> 'b) −> (('a list) −> ('b −> 'b))
```

• length returns the length of a list.
• @ (append) appends two lists (e.g. [1;2]@[3;4] is the same as [1;2;3;4]).
• rev returns the reverse of a list.
• map takes a function and then a list, and applies the function to all the elements of the list, returning the list of the results (i.e. map **f** [$\mathbf{e_1}$; .. ; $\mathbf{e_n}$] is [**f** ($\mathbf{e_1}$); .. ; **f** ($\mathbf{e_n}$)]).
• fold maps a binary function to an n-ary function over a list (e.g. it can map + to the function computing the sum of the elements of a list), it takes first a binary function, then the list to accumulate, then the value to return on empty lists (i.e. fold **f** [$\mathbf{e_1}$; .. ; $\mathbf{e_n}$] **e** is **f** ($\mathbf{e_1}$, .. **f** ($\mathbf{e_n}$,**e**)..)).
• revfold is just like fold but accumulates in the opposite direction (i.e. revfold **f** [$\mathbf{e_1}$; .. ; $\mathbf{e_n}$] **e** is **f** ($\mathbf{e_n}$, .. **f** ($\mathbf{e_1}$,**e**)..)).

3.6. Strings

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

Operation on strings are:

```
size            : string −> int
extract         : (string # int # int) −> string
explode         : string −> string list
implode         : string list −> string
explodeascii    : string −> int list
implodeascii    : int list −> string
intofstring     : string −> int
stringofint     : int −> string
```

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

The escape character for strings and is \; the conventions used to introduce quotes and non-printable characters inside strings are described in appendix "Escape sequences for strings". For Ascii characters we intend here full 8-bit codes in the range 0..255.

3.7. Disjoint unions

The strong typing rules of ML do not directly allow one to write functions which return different kinds of values, e.g. integers and booleans. Similarly, lists have to be homogeneous and cannot contain, say, functions and pairs at the same time. These restrictions can be relaxed by using objects of union types; for example an object of type int + bool can ‘contain’ (as opposed to ‘be’) an integer or a boolean.

An object of type int + bool is essentially an integer or a boolean with a *tag* ‘left’ or ‘right’ which determines whether the object is in the left part of the sum (in which case it must be an integer) of in the right part of the sum (in which case it must be a boolean).

Operations on disjoint sums are:

```
inl          : 'a -> ('a + 'b)
inr          : 'b -> ('a + 'b)
outl         : ('a + 'b) -> 'a
outr         : ('a + 'b) -> 'b
isl, isr     : ('a + 'b) -> bool
```

• Objects of union types are built by applying the operators inl (in-left, mapping a value to the left part of a disjoint union) and inr (in-right, mapping a value to the right part of a disjoint union). We say that inl produces left injections, and inr produces right injections.
• isl (is-left) and isr (is-right) can be used to test whether an element of a union type is a left or a right injection.
• outl (out-left) projects a left injection to its basic value; it fails with string "outl" when applied to a right injection. outr (out-right) projects a right injection to its basic value; it fails with string "outr" when applied to a left injection.

Disjoint sums provide the basic capabilities for union types, and are fairly convenient for sums of two or three types; however they become awkward for larger unions, for which it is more convenient to use the labeled n-ary union types described in the "Variants" section.

3.8. Records

A record is very similar to a tuple $(\mathbf{x_1},..,\mathbf{x_n})$ where each $\mathbf{x_i}$ is associated to a different label; then the order of the $\mathbf{x_i}$ is not important because every component can be identified by its label. Labels are not values (they cannot be computed) and are not identifiers (they do not conflict with variable or type names). Here is a

record with three components, labeled a, b and c:

expression: (|a = 3; b = true; c = "1"|)

The special brackets (| and |) are used to delimit records and record types. In general, a record is a collection of unordered named fields:

syntax: (|$\mathbf{a_1}$ = $\mathbf{e_1}$; .. ; $\mathbf{a_n}$ = $\mathbf{e_n}$|) n≥0

where the identifiers $\mathbf{a_i}$ are field names and the terms $\mathbf{e_i}$ are field values (expressions). The type of a record is a labeled unordered product:

syntax: (|$\mathbf{a_1}$:$\mathbf{t_1}$; .. ; $\mathbf{a_n}$:$\mathbf{t_n}$|)

typing rule: if $\mathbf{e_i}$: $\mathbf{t_i}$ (1≤i≤n) then (|$\mathbf{a_1}$ = $\mathbf{e_1}$; .. ; $\mathbf{a_n}$ = $\mathbf{e_n}$|) : (|$\mathbf{a_1}$:$\mathbf{t_1}$; .. ; $\mathbf{a_n}$:$\mathbf{t_n}$|)

The only operation defined on records is field selection. A record field can be selected by its field name, using the *dot notation*:

syntax: **e**.**a**

where (|$\mathbf{a_1}$ = $\mathbf{e_1}$; .. ; $\mathbf{a_n}$ = $\mathbf{e_n}$|).$\mathbf{a_i}$ = $\mathbf{e_i}$ 1≤i≤n

The exact type of every record must be known; the expression fun r. r.a will fail to typecheck in isolation because not all fields of r are known (we only know it has an a field). However fun r. r.a is accepted when the context provides enough information about r:

expression: fun r. r.a;

error: Unresolvable Record Selection of type: (|a: 'a|)
Field selector is: a

expression: (fun r. r.a) (|a = 3; b = true|);

result: 3 : int

Application of a function to a record gives a sort of call-by-keyword facility, because records are unordered:

```
expression:  let val f (|a = x; b = y|) = x,y
             in f (|b = 2; a = 1|) end;

result:      1,2 : int # int
```

When a record is used in a pattern, the effect of a Pascal with statement is obtained.

3.9. Variants

Variants and records can be considered as complementary concepts; a record type is a labeled product of types, while a variant type is a labeled sum (disjoint union) of types. An object of a variant type can belong to one of several types; these different cases are distinguished by the label attached to every variant object. Hence testing the label of a variant object is like testing its type out of a finite set of possibilities.

declaration: type OneOfTwo = [|Int: int; Bool: bool|];

```
binding:        type OneOfTwo = [|Bool: bool; Int: int|]
```

The special brackets [| and |] are used to delimit variants and variant types.

The labels Bool and Int are rearranged in alphabetical order (they should not be confused with the types bool and int which follow ':'). Like in records, labels are not values, variables or types.

Two basic operations are defined on variants: is tests the label of a variant object, and as returns the object contained in the variant. These two operations are followed by a label (just like '.' on records): we should think of is l and as l, for every label l, as composite operators, i.e. not as the application of is or as to l.

```
declaration:    val v = [|Int = 3|] : OneOfTwo;

binding:        val v = [|Int = 3|] : OneOfTwo

expression:     v is Int, v is Bool;

result:         (true,false) : bool # bool

expression:     v as Int;

result:         3 : int

expression:     v as Bool;

failure:        Failure: as
```

We must specify ': OneOfTwo' in the definition of v because v might also belong to some different variant containing a case Int: int. This type specification is not needed in general if the context gives enough information about v. Here is what happens if we forget to specify the type:

```
expression:     [|Int = 3|];

error:          Unresolvable Variant Type:      [|Int: int|]
```

A very useful abbreviation concerning variant types is the following: whenever we have a variant type with a field a: unit we can abbreviate that field specification to a, and whenever we have a variant object [|a = ()|] we can abbreviate it to [|a|]. This is very convenient in defining *enumeration types*, which are variant types where only the labels are relevant and the types associated with them are not used. For example, colors and fruits are enumeration types:

```
declaration:    type color = [|red; orange; yellow|]
                and fruit = [|apple; orange; banana|];

bindings:       type color = [|orange; red; yellow|]
                type fruit = [|apple; banana; orange|]

declaration:    val fruitcolor (fruit: fruit): color =
                    if fruit is apple then [|red|]
                    else if fruit is banana then [|yellow|]
                    else if fruit is orange then [|orange|]
                    else fail;

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

else fail is actually never executed; a compile-time error message is given when attempting to pass an argument which is not a fruit.

Note that [|orange|] is both a color and a fruit; it is possible to disambiguate the occurrence above because of the type declarations in the definition of fruitcolor.

In general, a variant type is an unordered disjoint union of types:

syntax: [|$\mathbf{a_1}$:$\mathbf{t_1}$; .. ; $\mathbf{a_n}$:$\mathbf{t_n}$|] n≥0

an object of this type embodies exactly one of the possible alternatives offered by the type:

syntax: [|**a** = **e**|]

typing rule: if **e** : **t** then [|**a** = **e**|] : [| .. ; **a** : **t**; .. |]

Variants can be used in patterns, for example:

(fun [|a = x|]. x) is equivalent to (fun v. v as a)

As we have seen, we cannot write something like [|blue|] unless the type of [|blue|] is further specified by the context. This is because the typechecker must be able to infer exactly the name and type of all the variants contained in the type of every variant-object (i.e. every variant-object must have, at the end of typechecking, a *rigid* type, as explained below). This restriction is only due to efficiency considerations so that the compiler can generate efficient code for the case statement (i.e. a jump table as opposed to a sequence of if-then-else).

Partially specified records and variants are called *flexible,* while fully specified ones are *rigid,* in as we might call int + bool a rigid binary disjoint sum and int + bool + 'a a flexible binary disjoint sum. There is no way to express this distinction syntactically, but it is sometimes useful to know the default actions taken by the typechecker. Two rigid types will typecheck only if they have the same named fields, while they will typecheck to the 'union' of their fields if both of them are flexible. If only one is flexible, the fields of the flexible type must be included in the fields of the rigid one.

Here are the rules: a record constant has a rigid type; a variant constant has a flexible type; x.a, x is a and x as a assign a flexible type to x; type definitions type r = (|..|) and v = [|..|] and type specifications r: (|..|), v: [|..|] assign rigid types both to r and to v.

3.10. Updatable references

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

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

ref	: 'a −> 'a ref	
!	: 'a ref −> 'a	(prefix)
:=	: 'a ref # 'a −> unit	(infix)

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

declaration: val a = ref 3;

```
binding:        val a = (ref 3) : int ref

expression:     a := 5;

result:         () : unit

expression:     ! a;

result:          5 : int
```

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

```
declaration:    type 'a refpair = 'a ref # 'a ref;

binding:        type 'a refpair = 'a ref # 'a ref

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

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

expression:     r := 5;

result:         () : unit

expression:     p;

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

3.11. Arrays

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

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

The array primitives are:

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

• array makes a constant array (all items equal to the third argument) of size n≥0 (second argument) from a lowerbound (first argument). It fails with string "array" if the size is negative.
• arrayoflist makes an array out of a list, given a lower bound for indexing.

• lowerbound returns the lower bound of an array.
• arraysize returns the size of an array.
• sub extracts the i-th item of an array. It fails with string "sub" if the index is not in range.
• update updates the i-th element of an array with a new value. It fails with string "update" if the index is not in range.
• arraytolist converts an array into the list of its elements.

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

```
infix sub;
infix 2000 update 2000;

export

        type array
        val array arrayoflist lowerbound arraysize sub update arraytolist

from

        type 'a array <=> (|lb: int; size: int; table: 'a ref list|);

        val rec length (l: 'a list) : int =
        if null l then 0 else 1 + length(tl l);

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

        val array (lb: int, length: int, item: 'a) : 'a array =
                if length<0 then failwith "array"
                else absarray(|lb=lb; size=length; table=list length|)
                        where rec list n =
                                if n=0 then [ ] else (ref item) :: list(n-1)
                        end

    and arrayoflist (lb: int, list: 'a list) : 'a array =
                absarray (|lb=lb; size=length list; table=map ref list|)

        and lowerbound (array: 'a array) : int =
            (reparray array).lb

        and arraysize (array: 'a array) : int =
                (reparray array).size

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

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

        and arraytolist (array: 'a array) : 'a list =
                map (!) ((reparray array).table)

end;
```

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

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

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

3.12. Functions

Functions are usually defined using the syntax:

syntax: val f **pattern** = **body**

but they can also be introduced in isolation by the use of *fun-notation*. The above definition is actually an abbreviation for

syntax: val f = fun **pattern**. **body**

which associates the function fun **pattern**. **body** to the identifier f in the same way that simple values are associated to identifiers.

A *fun-expression*, like the one above, is an expression denoting an unnamed function; it has a *pattern*, preceded by fun, and a *body*, preceded by '.'. The pattern is the formal parameter of the function. The body is any expression, and its value is the result of the function. The body of a fun-expression extends syntactically to the right as far as possible, so it not uncommon to enclose fun-expressions in parentheses to delimit their scope.

As an abbreviation, nested fun-expressions of the form

syntax: fun **pattern**$_1$. .. fun **pattern**$_n$. **body**

e.g. fun a. fun b. fun c. a + b + c

may be written:

syntax: fun **pattern$_1$** .. **pattern$_n$**. **body**

e.g. fun a b c. a + b + c

This is consistent with the notation for partial application in declarations, where one can write val f a b c = .. for val f = fun a b c. ...

The type of every expression and pattern can be specified by suffixing a colon and a type expression, (note that parentheses are needed around type specifications in patterns, when they are not part of larger patterns). To completely specify the type of a fun-expression we may use either of the two forms:

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

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

result: fun : bool −> ((int # int) −> int)

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

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

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

binding: val f : bool −> ((int # int) −> int)

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

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

The type of a lambda expression is determined by the type of its binder and the type of its body:

typing rule: if **x** : **t** and **e** : **t’** then (fun **x**. **e**) : **t** −> **t’**

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

3.13. Application

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

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

syntax: **f a**

The expression **a** can also be parenthesized (as any expression can), obtaining the standard application syntax **f**(**a**). It is common *not* to use parentheses when the argument is a simple variable, a string (**f** "abc"), a list (**f** [1;2;3]), a record (**f (|r=3; s=4|)**) or a variant (**f** [|v="c"|]). It is necessary to use parentheses when the argument is an infix expression like a tuple or an arithmetic operation (**f** (**a**,**b**,**c**), **f** (**a**+**b**)) because the precedence of application is normally greater than the precedence of the infix operators (**f a**,**b** and **f a**+**b** are interpreted as (**f a**),**b** and (**f a**)+**b**). Function application binds stronger than any predefined operator, except '!' (dereferencing); see the appendix "Precedence of operators and type operators" for details. In case of partial application, the form **f g a**, is interpreted as (**f g**) **a**, and not as **f** (**g a**); note that they are both equally meaningful.

Infix identifiers can be applied in two ways:

syntax: **a f b**

syntax: **f** (**a**,**b**)

The first form is the expected one; an infix identifier has always a type matching ('a # 'b) −> 'c. The second possibility derives from the fact that infix operators in non-infix positions are treated as nonfixes; for example for infix **f** and **g** we can write the list [**f**; **g**] without incurring syntax errors. In the second form above, **f** is found in a non-infix position, and is applied as a nonfix to the pair (**a**,**b**).

Similarly, suffix operators have two application forms:

syntax: **a f**

syntax: **f a**

The backward form is characteristic of suffixes, so that we can write expressions like 3 square. The second form is legal because the suffix **f** is found in a non-suffix position, and is treated like a nonfix.

Prefix operators have only one application form

syntax: **f a**

They are only interesting because of the way they interact with other operators; by adjusting their binding power we can, for example, make them bind stronger than application on one side only. This happens for the predefined prefix operator ! (dereferencing), so that g !a is interpreted as **g** (! **a**) (instead of the standard (**g** !) **a**), while ! **f a** is interpreted as ! (**f a**), which is what one would expect in most situations.

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

typing rule: if **f** : **t** −> **t'** and **a** : **t** then (**f a**) : **t'**

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

3.14. Conditional

The syntactic form for conditional expressions is:

syntax: if $\mathbf{e_1}$ then $\mathbf{e_2}$ else $\mathbf{e_3}$

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

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

The else branch must always be present.

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

typing rule: if $\mathbf{e_1}$: bool and $\mathbf{e_2}$: **t** and $\mathbf{e_3}$: **t**
then (if $\mathbf{e_1}$ then $\mathbf{e_2}$ else $\mathbf{e_3}$) : **t**

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

3.15. Sequencing

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

syntax: ($\mathbf{e_1}$; .. ; $\mathbf{e_n}$) n≥2

(the parentheses are needed), which evaluates $\mathbf{e_1}$ .. $\mathbf{e_n}$ in turn and returns the value of $\mathbf{e_n}$. The type of a sequencing expression is the type of $\mathbf{e_n}$. Example:

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

result:       2 : int
```

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

typing rule: if **e** : **t** then ($\mathbf{e_1}$; .. ; $\mathbf{e_n}$; **e**) : **t** n ≥2

3.16. While

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

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

syntax: while $\mathbf{e_1}$ do $\mathbf{e_2}$

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

typing rule: if $\mathbf{e_1}$: bool and $\mathbf{e_2}$: unit
then (while $\mathbf{e_1}$ do $\mathbf{e_2}$) : unit

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

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

3.17. Case

The case expression provides a systematic way of structuring programs based on variant types. It is very common for a function to take an argument of some variant type, and to do different things according to the variant label. In the "Variants" section we have seen a way of doing this by the is and as operations; here is an example taken from that section and rewritten with a case expression:

```
declaration:    type color = [|red; orange; yellow|];
                and  fruit = [|apple; orange; banana|];

bindings:           type color = [|orange; red; yellow|]
                    type fruit = [|apple; banana; orange|]

declaration:    val fruitcolor (fruit: fruit): color =
                    case fruit of
                        [|apple.  [|red|];
                          banana. [|yellow|];
                          orange. [|orange|]
                        |];

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

The case construct is a convenient form of saying if fruit is apple then [|red|] if fruit is banana then [|yellow|] if fruit is orange then [|orange|] else fail (else fail never arises; a compile-time error message is given if some case is missing).

The general form of a case expression is:

syntax: case **x** of [|$\mathbf{a_1}$ = $\mathbf{p_1}$. $\mathbf{e_1}$; .. ; $\mathbf{a_n}$ = $\mathbf{p_n}$. $\mathbf{e_n}$|]

where **x** has type [|$\mathbf{a_1}$:$\mathbf{t_1}$; .. ; $\mathbf{a_n}$:$\mathbf{t_n}$|], $\mathbf{a_1}$..$\mathbf{a_1}$ are variant labels, $\mathbf{p_1}$..$\mathbf{p_n}$ are patterns, and $\mathbf{e_1}$..$\mathbf{e_n}$ are all expressions of type **t**, which is also the type of the whole case expression.

The object **x** must be a variant object of the form [|$\mathbf{a_i}$ = **y**|] (the typechecker guarantees this) which is matched to the pattern $\mathbf{p_i}$ corresponding to the label $\mathbf{a_i}$. A set of variable bindings is established by this matching process, according to the rules described in section "Patterns". These variables can be used in the expression $\mathbf{e_i}$ which determines the resulting value of the case expression.

For example, let us redefine the type fruit so that it has a color and an origin as attributes:

```
declaration:    type country =
                    [|Israel; Britain; Morocco|];

binding:        type country = [|Britain; Israel; Morocco|];

declaration:    type attribute =
                    (|color: color; origin: country|);

binding:        type attribute = (|color: color; origin: country|);

declaration:    type fruit =
                    [|apple: attribute;
                      orange: attribute;
                      banana: attribute
                    |];

binding:        type fruit = [|apple: attribute; banana: attribute; orange: attribute|]

declaration:    val fruitcolor (fruit: fruit): color =
                    case fruit of
                        [|apple = (|color; origin|).  color;
                          banana = (|color; origin|).  color;
                          orange = (|color; origin|).  color
                        |];

binding:        val fruitcolor : fruit −> color
```

Note that the pattern (|color; origin|) is an abbreviation for (|color = color; origin = origin|) (see section "Patterns"), where the first color is a record label, and the second color is a variable which is bound to a fruit color and then used in the case branches.

If we already know what the colors of the different fruits must be, we can write them in the pattern so that fruit parameters will be matched against them:

```
declaration:    val origin (fruit: fruit): country =
                    case fruit of
                        [|apple = (|color = [|red|]; origin|).  origin;
                          banana = (|color = [|yellow|]; origin|).  origin;
                          orange = (|color = [|orange|]; origin|).  origin
                        |];

binding:        val origin : fruit −> country
```

This will produces a run-time pattern failure if the color of a fruit is not what is expected.

The typing rule for case is:

typing rule: if **x** : [|$\mathbf{a_1}$: $\mathbf{t_1}$; .. ; $\mathbf{a_n}$: $\mathbf{t_n}$|]
and $\mathbf{p_1}$: $\mathbf{t_1}$ and .. and $\mathbf{p_n}$: $\mathbf{t_n}$
and $\mathbf{e_1}$: **t** and .. and $\mathbf{e_n}$: **t**
then (case **x** of [|$\mathbf{a_1}$ = $\mathbf{p_1}$. $\mathbf{e_1}$; .. ; $\mathbf{a_n}$ = $\mathbf{p_n}$. $\mathbf{e_n}$|]) : **t**

This says that all the possible cases of **x** must be considered in the case expression, and that the values returned for each case must have the same type.

3.18. Scope blocks

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

syntax: let **declaration** in **expression** end

syntax: **expression** where **declaration** end

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

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

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

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

3.19. Exceptions and traps

An exception can be raised by a system primitive or by a user program. When an exception is raised, the execution of the current expression is abandoned, and a *failure string* is propagated outward, tracing back along the history of function calls and expression evaluations which led to the exception. If the exception is allowed to propagate up to the top level, a message is printed:

failure: Failure: **reason**

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

User exceptions can be raised by the failwith construct (fail is an abbreviation of failwith "fail"):

syntax: fail

syntax: failwith **expression**

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

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

The result of the evaluation of $\mathbf{e_1}$? $\mathbf{e_2}$ is normally the result of $\mathbf{e_1}$, unless $\mathbf{e_1}$ fails, in which case it is the result of $\mathbf{e_2}$.

The result of the evaluation of $\mathbf{e_1}$?? $\mathbf{e_2}$ $\mathbf{e_3}$ (where $\mathbf{e_2}$ evaluates to a string list **sl**) is normally $\mathbf{e_1}$, unless $\mathbf{e_1}$ fails, in which case it is the result of $\mathbf{e_3}$ whenever the failure string is one of the strings in **sl**; otherwise the failure is propagated.

The result of the evaluation of $\mathbf{e_1}$?\ **v** $\mathbf{e_2}$ is normally the result of $\mathbf{e_1}$, unless $\mathbf{e_1}$ fails, in which case it is the result of $\mathbf{e_2}$, where the variable **v** is associated to the failure string and can be used in $\mathbf{e_2}$.

syntax: e_1 ? e_2

syntax: e_1 ?? e_2 e_3

syntax: e_1 ?\ v e_2

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

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

typing rules: (failwith **e**) : 'a

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

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

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

3.20. Type semantics and type specification

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

$$V = Bool + Int + .. + (V + V) + (V \# V) + (V \rightarrow V) \qquad [1]$$

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

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

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

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

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

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

Second, 'a −> 'a, as a domain, is smaller than any of its instances, for example int −> int. In fact any function returning an 'a when given an 'a, must return an int when given an int (i.e. 'a −> 'a ⊆ int −> int), but an int −> int function is not required to return a boolean when given a boolean (i.e. int −> int ⊃ 'a −> 'a). Hence a function like (fun x. x+1) : int −> int is not an 'a −> 'a, although it is an 'a −> 'b.

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

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

syntax: **expression** : **type**

the effect is to take the *join* of the type of the expression inferred by the system and of the type following ':' specified by the user (if this join exists) as the type of the expression. The join of two domains is the smallest domain which contains both.

This implicit join operation explains why the specifications:

expression: 3 : 'a;

result: 3 : int

expression: (fun x. x) : int −> int;

result: fun : int −> int

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

result: fun : int −> int

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

3.21. Equality

It is impossible to use the mathematical definition of equality, as this is in general undecidable. Hence equality is considered to be a predefined overloaded operator, i.e. an infinite collection of decidable equality operators, whose types are instances of ('a # 'a) −> bool.

This still leaves a great deal of freedom in the choice of the semantics of = because we can overload it to an arbitrary degree. In the present version, equality only works on monotypes which do not contain function spaces or isomorphic domains.

If only monotypes are involved, then the compiler can produce code which does not need to perform run-time type checking (consider the polymorphic equality fun a. a=a: what code should we produce for it?). Note that in the absence of run-time typechecking not even monomorphic equality can be compiled as a subroutine; the compiler must produce specialized in-line code for every occurrence of = or <> (inequality). However a general equality subroutine can be written which uses the limited run-time type information used by the garbage collector.

Extensional (i.e. mathematical) equality on functions is undecidable. Intensional (i.e. compiled-code) equality on functions is too arbitrary. Hence equality on functions is disallowed; where needed, functions can be marked by data having a decidable equality, but then the user must write a special purpose equality routine.

Equality over isomorphic types might be implemented as equality over the corresponding concrete representations. However in this case we would have troubles in the use of isomorphisms in abstract types; for example equality would not be transparent to a change of representation (e.g. from an int list to an

int−>int). Also, if we implement abstract sets by concrete multisets, equality could distinguish between sets which should be equal, giving dangerous insights on the chosen representation, as well as not corresponding to set equality.

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

Right	Wrong
() = ();	
true = false;	
3 = 3;	
"a" = "";	
(2,"a") = (2,"a");	
[] = ([]: int list);	[] = [];
[1;2] = [3];	
inl 3 = (inl 3: int+unit);	inl 3 = inl 3;
inl 3 = inr ();	
fun a,b. a=b : string#string −> bool;	fun a,b. a=b;
	fun a. a = fun a. a;
	absset[3] = absset[3];

4. Type expressions

A type expression is either a *type variable*, a *basic type*, or the application of a *type operator* to a sequence of type expressions. Type operators are usually suffix, except for the infix operators # and −> and for the outfix operators (| .. |) and [| .. |]. The user can define prefix, infix or suffix type operators (see section "Lexical declarations").

4.1. Type variables

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

4.2. Type operators

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

5. Declarations

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

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

5.1. Value bindings

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

5.1.1. Patterns

The left hand side of an = in a value definition can be a simple variable, or a more complex pattern involving several distinct variables. A pattern is said to *match* a value on the right hand side of the =; the variables in the pattern are associated to the corresponding parts of the value. The matching rules are given recursively on the structure of the pattern. The letters **a** and **b** denote variables, **u** and **v** values, and **p** and **q** patterns.

Pattern	Value	Match(Pattern,Value)	
$	v	∅	
()	()	∅	
a	**v**	{**a** = **v**}	
p,**q**	**u**,**v**	Match(**p**,**u**) ∪ Match(**q**,**v**)	
p :: **q**	**u** :: **v**	Match(**p**,**u**) ∪ Match(**q**,**v**)	
[$\mathbf{p_1}$;..;$\mathbf{p_n}$]	[$\mathbf{v_1}$;..;$\mathbf{v_n}$]	Match($\mathbf{p_1}$,$\mathbf{v_1}$) ∪ .. ∪ Match($\mathbf{p_n}$,$\mathbf{v_n}$)	n ≥ 0
[|**l**=**p**|]	[|**l**=**v**|]	Match(**p**,**v**)	
(|$\mathbf{l_1}$=$\mathbf{p_1}$;..;$\mathbf{l_n}$=$\mathbf{p_n}$|)	(|$\mathbf{l_1}$=$\mathbf{v_1}$;..;$\mathbf{l_n}$=$\mathbf{v_n}$|)	Match($\mathbf{p_1}$,$\mathbf{v_1}$) ∪ .. ∪ Match($\mathbf{p_n}$,$\mathbf{v_n}$)	n ≥ 0
ref **p**	ref **v**	Match(**p**,**v**)	

The pattern **p** :: **q** fails with "destcons" if it is associated to an empty list. The pattern [$\mathbf{p_1}$; .. ; $\mathbf{p_n}$] fails with "destcons" if it is associated to a list shorter than n, and fails with "destnil" if it is associated to a list longer than n. The pattern [|**l**=**p**|] fails with "destvariant" if it is associated to a variant [|**l'**=**v**|] with **l** ≠ **l'**. All other mismatchings are detected at compile time as type errors.

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

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

The variables in a single pattern must all be distinct.

A record pattern of the form (|..; a=a; ..|) can be abbreviated as (|..; a; ..|). A variant pattern of the form [|a=()|] can be abbreviated as [|a|].

5.1.2. Simple bindings

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

syntax: **pattern** = **expression**

syntax: **ide pattern**$_1$ .. **pattern**$_n$ = **expression**

This declaration imports the variables used in expression and exports the variables defined on the left of '='. Here are some examples:

```
declaration:    val x = 2 * 3;

binding:            val x = 6 : int

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

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

declaration: val f x = x;

binding: val f : 'a −> 'a

5.1.3. Parallel bindings

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

val $\mathbf{x_1}$, .. ,$\mathbf{x_n}$ = $\mathbf{e_1}$, .. ,$\mathbf{e_n}$;

val $\mathbf{x_1}$ = $\mathbf{e_1}$ and .. and $\mathbf{x_n}$ = $\mathbf{e_n}$;

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

syntax: **value_binding**$_1$ and **value_binding**$_2$

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

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

declaration: val a = 10 and b = 5;

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

declaration: val a = b and b = a;

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

5.2. Recursive bindings

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

syntax: rec **value_binding**

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

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

A recursive value declaration can only define functions; simple value bindings are not allowed to be recursive. Only the environment operators and and rec are admitted within a rec.

5.3. Type bindings

Type bindings define type abbreviations and type isomorphisms, and are prefixed by the keyword type. After type there can be a definition, a parallel binding or a recursive binding. Type definitions can be simple abbreviations for complex type expressions, or totally new types built by isomorphism from existing

types.

5.3.1. Simple bindings

The simplest form of type binding, called a *type abbreviation*, introduces a single, possibly parametric, type identifier:

syntax: **type_ide** = **type_exp**

syntax: **type_var type_ide** = **type_exp**

syntax: (**type_var**$_1$, .. ,**type_var**$_n$) **type_ide** = **type_exp** n > 0

A type defined by abbreviation denotes some complex type expression, and it is totally equivalent to that expression in any context.

```
declaration:    type intpair = int # int;

binding:            type intpair = int # int

declaration:    type 'a pair = 'a # 'a;

binding:            type 'a pair = 'a # 'a

declaration:    type ('a, 'b) pairpair = ('a # 'b) # ('a # 'b);

binding:            type ('a, 'b) pairpair = ('a # 'b) # ('a # 'b)
```

Type abbreviations cannot be recursive, and all the type variables appearing on the right of the defining = sign must appear on the left as parameters of the type being defined. The list of parameters on the left of the = sign must be a sequence of distinct type variables.

5.3.2. Isomorphism bindings

A type defined by isomorphism denotes a new type which is isomorphic, but not equal, to its base type. The symbol <=> is used instead of = in the definition of isomorphic types. The isomorphism is given by two functions which are defined together with the type. The name of these functions is obtained by prefixing abs and rep to the name of the type. Here are examples of isomorphic types with zero, one and two type parameters:

```
declaration:    type intpair <=> int # int;

bindings:           type intpair = −
                    val absintpair : (int # int) −> intpair
                    val repintpair : intpair −> (int # int)

declaration:    type 'a pair <=> 'a # 'a;

bindings:           type 'a pair = −
                    val abspair : ('a # 'a) −> 'a pair
                    val reppair : 'a pair −> ('a # 'a)

declaration:    type ('a, 'b) pairpair <=> ('a # 'b) # ('a # 'b);

bindings:           type ('a, 'b) pairpair = −
```

```
val abspairpair : ('a # 'b) # ('a # 'b) -> ('a, 'b) pairpair
val reppairpair : ('a, 'b) pairpair -> ('a # 'b) # ('a # 'b)
```

The types intpair and int # int are different, and any attempt to use an intpair as an int # int will result in a type error. An intpair is known to be isomorphic to int # int, but its internal structure is hidden; note that it is printed as '−'.

Type isomorphism in conjunction with the environment operators with or export can be used to define abstract data types (see below, and section "Abstract data types").

5.3.3. Parallel bindings

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

syntax: **type_binding**$_1$ and **type_binding**$_2$

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

5.3.4. Recursive bindings

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

syntax: rec **type_binding**

declaration: type rec 'a tree <=> unit + ('a tree) # ('a tree);

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

Only the operators and and rec are admitted within a rec.

5.4. Sequential declarations

Cascaded, or 'sequential' declarations are provided by the environment operator ';', which makes earlier declarations available inside later declarations, and otherwise has the same effect as and.

syntax: **d**$_1$; **d**$_2$

In **d**$_1$; **d**$_2$, the variables exported by **d**$_1$ are imported into **d**$_2$, but not vice versa. The variables exported by the whole ';' construct are the ones exported by **d**$_2$, plus the ones exported by **d**$_1$ which are not redefined in **d**$_2$.

declaration: val a = 10; val b = a + 5;

bindings:
```
val a = 10 : int
val b = 15 : int
```

5.5. Private declarations

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

```
syntax:        export
                   type t1 .. tn
                   val v1 .. vn
               from d end
```

The type identifiers $\mathbf{t_i}$ and the value identifiers $\mathbf{v_i}$ (which should be declared in the declaration **d**), are exported to the outside, while the other type identifiers and value identifiers defined in **d** are hidden. Both the type and val sections can be omitted, and there can be several of them intermixed.

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

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

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

bindings:              val increment : 'a -> unit
                       val fetch : 'b -> int

declaration:   export type pair
                       val pair left right
               from   (type pair <=> int # int;
                       val pair(a,b) = abspair(a,b)
                       and left p = fst(reppair p)
                       and right p = snd(reppair p))
               end;

bindings:              type pair = -
                       val pair : (int # int) -> pair
                       val left : pair -> int
                       val right : pair -> int
```

In the first example, the variable b is unknown at the top level. The second example shows how a variable can be made private to a function or a group of functions: only the functions increment and fetch have access to count, so that nobody can corrupt count. The third example is an abstract data type pair: note that the functions abspair and reppair have been hidden, so that the implementation of pairs is protected.

The with environment operator is commonly used to define abstract data types, and it just an abbreviation for an export declaration:

```
syntax:        d1 with d2 end
```

The with operator behaves like ';', except that the values defined in $\mathbf{d_1}$ are not known outside $\mathbf{d_1}$ with $\mathbf{d_2}$ (hence the values and types defined in $\mathbf{d_1}$ are known in $\mathbf{d_2}$, and the types defined in $\mathbf{d_1}$ are known outside $\mathbf{d_1}$ with $\mathbf{d_2}$). This means that, if $\mathbf{d_1}$ defines an isomorphism type **A**, then the type **A** will be exported, while the functions abs**A** and rep**A** will only be known inside $\mathbf{d_2}$. The pair example above can be written more conveniently as:

```
declaration:    type pair <=> int # int
                with
                    val pair(a,b) = abspair(a,b)
                    and left p = fst(reppair p)
                    and right p = snd(reppair p)
                end;

bindings:           type pair = −
                    val pair : (int # int) −> pair
                    val left : pair −> int
                    val right : pair −> int
```

5.6. Module declarations

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

syntax: import $\mathbf{id_1}$ .. $\mathbf{id_n}$

```
declaration:    import minmax;

bindings:           val min : (int # int) −> int
                    val max : (int # int) −> int
```

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

5.7. Lexical declarations

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

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

```
declarations:
                    prefix half ;
                    infix  <−−> ;
                    suffix square ;
                    nonfix + ;
                    prefix type set ;
                    infix type ## ;
                    suffix type bag ;
                    nonfix type −> ;
```

This example declares a prefix operator half an infix operator <−−>, a suffix operator square and puts the operator + back to a status of normal non-infix identifier. Similarly for type fixes. These operators can now be uses as fixes in expressons and declarations.

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

Type operator and value operator fixity are independent; for example + can be both an infix function and a prefix type operator.

Left and right precedence can be specified by numbers in the appropriate positions:

```
declaration:
                    prefix half 200 ;
                    infix 100 <−−> 101 ;
                    suffix 50 square ;
                    nonfix + ;
```

In the case of infix operators, if the left precedence is less or equal to the right precedence, then the operator is left associative.

5.8. Parenthesized declarations

Complex declarations can be bracketed by (and). When parentheses are not used, the and operator binds stronger than with, and with binds stronger than ';'. All the infix operators are right associative.

6. Abstract data types

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

```
declaration:    type rec 'a tree <=> 'a + ('a tree # 'a tree)
                with
                        val mkleaf a = abstree(inl a)
                        and mknode(t,t') = abstree(inr(t,t'))
                        and isleaf t = isl(reptree t)
                        and left t = fst(outr(reptree t))
                        and right t = snd(outr(reptree t))
                        and leaf t = outl(reptree t)
                end;

bindings:               type 'a tree = −
                        val mkleaf : 'a −> 'a tree
                        val mknode : ('a tree # 'a tree) −> 'a tree
                        val isleaf : 'a tree −> bool
                        val left : 'a tree −> 'a tree
                        val right : 'a tree −> 'a tree
                        val leaf : 'a tree −> 'a
```

This definition uses the isomorphism (<=>) type definition and the with environment operator. The functions abstree and reptree defined by <=> are only known during the definition of the basic operations on trees. All users of the tree abstract type will be unable to take advantage of the concrete representation of trees given by 'a + ('a tree # 'a tree), thus making this type 'abstract'.

Note that the with operator can be preceded by any arbitrary declaration; this allows one to define mutually recursive abstract types:

```
declaration:    type rec a <=> .. b ..
                    and b <=> .. a ..
                with .. end;
```

In the declaration following with, the isomorphism functions are accessible as absa, absb, repa and repb.

Recursive types must use <=> instead of =. This fact sometimes forces one to use abstract types even when they are not conceptually necessary.

7. I/O streams

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

```
getstream    : string −> stream
putstream    : (string # stream) −> unit
newstream    : unit −> stream
channel      : string −> stream
instring     : (stream # int) −> string
outstring    : (stream # string) −> unit
lookahead    : (stream # int # int) −> string
emptystream  : stream −> bool
terminal     : stream
```

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

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

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

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

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

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

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

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

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

8. Modules

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

```
module:     module Pair
            body
                type 'a pair = 'a # 'a;
                val left (p: 'a pair) : 'a =   fst p
                and right (p: 'a pair) : 'a =   snd p
            end;
```

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

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

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

```
declaration:    import Pair;

bindings:           module /usr/lib/ml/lib
                    type 'a pair = 'a # 'a
                    val left : 'a pair −> 'a
                    val right : 'a pair −> 'a
```

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

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

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

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

8.1. Module hierarchies

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

A module can import other modules for several reasons.

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

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

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

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

Sometimes an imported module B *must* be reexported from an importing module A. This happens when a type identifier t defined in B is contained in the (types of the) bindings exported by A. Exporting those binders without exporting B may potentially generate objects of an unknown (in the current scope) type t, on which no operations are available. Hence the following restriction applies: whenever a type identifier is involved in the exports of a module, its type definition must also be exported.

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

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

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

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

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

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

declaration:   import A;

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

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

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

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

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

declaration:   import B;

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

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

```
declaration:   import B;

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

8.2. Module sharing

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

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

```
module:         module A
                body
                        type T <=> int;
                        val a = ref 3
                end;

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

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

module:         module D
                includes B C
                body
                end;

declaration:    import D;

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

                                type T = −
                                val CabsT : T −> int
```

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

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

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

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

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

8.3. Module versions

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

```
declaration:    import A;

failure:        Module A must be compiled
                Failure: link
```

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

A new version of a module is generated whenever that module is recompiled and its external interface (i.e. the types of the exported bindings) changes.

9. The ML system

9.1. Entering the system

The ML system is entered under Unix by typing ml as a shell command. A library of standard functions is then loaded. This library is installation dependent and is meant to contain the functions which are most used locally. This is a typical screen, just after loading ml:

```
$ ml
> val length : ('a list)  ->  int
|  val @ : (('a list) # ('a list))   ->  ('a list)
|  val rev : ('a list)  ->  ('a list)
|  val map  : ('a -> 'b) -> (('a list) -> ('b list))
|  val fold : ('a -> ('b -> 'b))   -> (('a list)  ->  ('b -> 'b))
|  val revfold : ('a -> ('b -> 'b))   -> (('a list)  ->  ('b -> 'b))
|  val curry : (('a # 'b)  -> 'c)  ->  ('a ->  ('b -> 'c))

_
```

You can now start typing expressions or definitions, or loading source files.

9.2. Loading source files

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

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

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

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

There is no way to inhibit printing while loading a file.

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

9.3. Exiting the system

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

9.4. Error messages

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

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

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

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

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

```
− let    type A <=> int;
         val oldabsA = absA;
         type A <=> int
  in repA(oldabsA 3) end;
Type Clash in:        (repA (oldabsA 3))
Looking  for :        A
I have found :A
Clash between two different types having the same name: A
```

Some type errors are peculiar to records and variants; they should be corrected by introducing more explicit type information in the program (see sections "Records" and "Variants"):

```
− val f r = r.a, r.b;
Unresolvable Record Selection of type:    (|a:’a; b:’b|)
```

```
Field selector is:                          a

− [|a=3|];
Unresolvable Variant Type:          [|a:int|]

− val f v = v is a, v is b;
Unresolvable Variant Selection of type:   [|a:'a; b:'b|]
Case selector is:                          a
```

Equality also has its own type errors:

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

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

− let type A <=> int
  in absA 3 = absA 3 end;
Invalid type of args to "=" or "<>":        A
I cannot compare abstract types
```

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

```
− val a = 3;
>  val a = 3 : int

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

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

```
− import A;

Module A must be compiled
Failure: link
```

9.5. Monitoring the compiler

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

```
− monitor;
"y" for Yes; <CR> for No.
ParseTree?
Types?
```

```
TypeVariables?
StackCode?
OptimizerOff?
AssemblerCode?
ObjectCode?
Environment?
StopBeforeExec?
CheckHeap?
MemoryAllocation?
Watch? hex:
Timing?

_
```

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

• *ParseTree* prints the parse tree of every subsequent top level phrase. This can be useful to see how the precedence of operators works.
• *Types* prints the type of every subsequent top level phrase, before executing it. It is useful only in debugging the system, as the type is the same as the type of the result which is printed after evaluation.
• *TypeVariables* prints all the hidden attributes of type variables. The following is for typechecking wizards only. 'a[i] means that 'a has an occurrence level i (i.e. it is used in a fun-nesting of depth i). Generic type variables are written 'a[I] (I for infinity). Weak type variables are written _'a[i], and non-generic type variables are written !'a[i].
• *StackCode* for every subsequent top-level phrase, prints the intermediate Functional Abstract Machine code produced by the compiler [Cardelli 83], after peephole optimization.
• *OptimizerOff* switches off the peephole optimizer for the abstract machine code. The optimizer works mainly on multiple jumps, function return, partial application, and tail recursion.
• *AssemblerCode* for every subsequent top-level phrase, prints the VAX assembler instructions produced from the abstract machine code.
• *ObjectCode* for every subsequent top-level phrase, prints the final hexadecimal result of the compilation.
• *Environment* for every subsequent top-level phrase, prints all the the types currently defined with their definition, and all the variables currently defined with their value and type.
• *StopBeforeExec* for every subsequent top-level phrase, stops immediately before executing the result of compilation, asking for a confirmation to proceed.
• *CheckHeap* for every subsequent top-level phrase, makes a complete consistency check of the data in the heap, and reports problems.
• *MemoryAllocation* reports every Unix memory allocation or deallocation, except the ones caused by Pascal.
• *Watch* is used for internal system debugging.
• *Timing* at the end of every evaluation, gives the parsing time (Pars), typechecking time (Anal), translation to abstract machine code and optimization time (Comp), translation to VAX code time (Assm), total compilation time (Total) and run time (Run). All in milliseconds.

9.6. Garbage collection and storage allocation

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

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

garbage collection is less frequent.

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

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

Appendix A. Lexical classes

Ascii characters are classified into the following categories:

- Illegal — Unacceptable in a source program. These characters are ignored, and a warning message is printed.
- Eof — End-of-file character. It terminates an interactive session, or stops the process of reading a source ml file. A top level control-D is interpreted as Eof, thereby exiting the system.
- Blank — Characters which are interpreted as a space character " ".
- Digit — Digits.
- Letter — Letters and other characters which can be used to form identifiers.
- Symbol0 — A class of character which can be used to form operators (see appendix "Syntax of lexical entities").
- SymbolA — A class of character which can be used to form operators (see appendix "Syntax of lexical entities").
- SymbolB — A class of character which can be used to form operators (see appendix "Syntax of lexical entities").
- Escape — Escape character in strings. It behaves as a Symbol0 outside strings.
- TypVar — Character starting type variables.
- Quote — String quotation character.
- Delim — One-character punctuation marks.
- LftPar — Left parenthesis character (see "Lexical Matters")
- RhtPar — Right parenthesis character (see "Lexical Matters")
- Star — Initial character of a type variable.

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

```
nul  Eof;       soh  Illegal;   stx  Illegal;
etx  Illegal;   eot  Illegal;   enq  Illegal;
ack  Illegal;   bel  Illegal;   bs   Illegal;
ht   Blank;     lf   Blank;     vt   Blank;
ff   Blank;     cf   Blank;     so   Illegal;
si   Illegal;   dle  Illegal;   dc1  Illegal;
dc2  Illegal;   dc3  Illegal;   dc4  Illegal;
nak  Illegal;   syn  Illegal;   etb  Illegal;
can  Illegal;   em   Illegal;   sub  Illegal;
esc  Illegal;   fs   Illegal;   gs   Illegal;
rs   Illegal;   us   Illegal;   ' '  Blank;
'!'  SymbolB;   '"'  Quote;     '#'  SymbolA;
'$'  SymbolB;   '%'  Synmbol0;  '&'  Symbol0;
'''  TypVar;    '('  LftPar;    ')'  RhtPar;
'*'  SymbolB;   '+'  SymbolA;   ','  Delim;
'-'  SymbolA;   '.'  Delim;     '/'  Symbol0;
'0'  Digit;     '1'  Digit;     '2'  Digit;
'3'  Digit;     '4'  Digit;     '5'  Digit;
'6'  Digit;     '7'  Digit;     '8'  Digit;
'9'  Digit;     ':'  SymbolA;   ';'  Delim;
'<'  SymbolA;   '='  SymbolA;   '>'  SymbolA;
'?'  SymbolB;   '@'  SymbolB;   'A'  Letter;
```

```
'B'   Letter;     'C'   Letter;     'D'   Letter;
'E'   Letter;     'F'   Letter;     'G'   Letter;
'H'   Letter;     'I'   Letter;     'J'   Letter;
'K'   Letter;     'L'   Letter;     'M'   Letter;
'N'   Letter;     'O'   Letter;     'P'   Letter;
'Q'   Letter;     'R'   Letter;     'S'   Letter;
'T'   Letter;     'U'   Letter;     'V'   Letter;
'W'   Letter;     'X'   Letter;     'Y'   Letter;
'Z'   Letter;     '['   LftPar;                 '\'   Escape;
']'   RhtPar;                '^'   Symbol0;              '_'   Letter;
'`'   Symbol0;              'a'   Letter;               'b'   Letter;
'c'   Letter;     'd'   Letter;     'e'   Letter;
'f'   Letter;     'g'   Letter;     'h'   Letter;
'i'   Letter;     'j'   Letter;     'k'   Letter;
'l'   Letter;     'm'   Letter;     'n'   Letter;
'o'   Letter;     'p'   Letter;     'q'   Letter;
'r'   Letter;     's'   Letter;     't'   Letter;
'u'   Letter;     'v'   Letter;     'w'   Letter;
'x'   Letter;     'y'   Letter;     'z'   Letter;
'{'   LftPar;                '|'   Symbol0;              '}'   RhtPar;
'~'   SymbolB;              DEL   Illegal;
```

Appendix B. Keywords

and
as
body
case
do
else
end
export
fail
failwith
from
fun
if
import
in
includes
infix
is
let
module
nonfix
of
prefix
rec
suffix
then
type
val
where
while
with
:
?
??
?\
<=>

Appendix C. Predefined identifiers

Identifier	Meaning	Fixity
true	Logic true	Nonfix
false	Logic false	Nonfix
not	Logic not	Prefix
&	Logic and	Infix
or	Logic or	Infix
~	Complement	Prefix
+	Plus	Infix
−	Difference	Infix
*	Times	Infix
div	Divide	Infix
mod	Modulo	Infix
=	Equal	Infix
<>	Different	Infix
>	Greater than	Infix
<	Less than	Infix
>=	Greater-eq	Infix
<=	Less-eq	Infix
size	String length	Nonfix
extract	Substring extraction	Nonfix
explode	String explosion	Nonfix
implode	String implosion	Nonfix
explodeascii	String to Ascii conv.	Nonfix
implodeascii	Ascii to string conv.	Nonfix
intofstring	String to int conv.	Nonfix
stringofint	Int to string conv.	Nonfix
fst	Pair first	Nonfix
snd	Pair second	Nonfix
::	List cons	Infix
hd	List head	Nonfix
tl	List tail	Nonfix
null	List null	Nonfix
length	List length	Nonfix
@	List append	Infix
map	List map	Nonfix
rev	List reverse	Nonfix
fold	List folding	Nonfix
revfold	List rev folding	Nonfix
inl	Left inject	Nonfix
inr	Right inject	Nonfix
outl	Left projection	Nonfix
outr	Right projection	Nonfix
isl	LeftInj test	Nonfix
isr	RightInj test	Nonfix
ref	New reference	Prefix
!	Dereferencing	Prefix
:=	Assignment	Infix
array	New array	Nonfix
arrayoflist	List to array	Nonfix
lowerbound	Array lower bound	Nonfix
arraysize	Array size	Nonfix

sub	Array indexing	Infix
update	Array update	Infix
arraytolist	Array to list	Nonfix
o	Function comp	Infix
getstream	Stream from file	Nonfix
putstream	Stream to file	Nonfix
newstream	Empty stream	Nonfix
channel	External channel	Nonfix
instring	Read from stream	Nonfix
outstring	Write to stream	Nonfix
lookahead	Peek from stream	Nonfix
emptystream	Test for empty stream	Nonfix
terminal	Terminal stream	Nonfix

Appendix D. Predefined type identifiers

unit	Unit Type		Nonfix
bool	Boolean Type		Nonfix
int	Integer Type	Nonfix	
string	String Type	Nonfix	
#	Cartesian Product	Infix	
+	Disjoint Union		Infix
−>	Function Space		Infix
list	List Type		Suffix
ref	Reference Type		Suffix
array	Array Type		Suffix
(| |)	Record Type	Outfix	
[| |]	Variant Type	Outfix	

Appendix E. Precedence of operators and type operators

(see section "Lexical declarations").

```
prefix                  not             600   ;
prefix                  ~               1100  ;
prefix                  !               1100  ;

infix       −100        ?               −100  ;
infix       −100        ?? exp          −100  ;       { in "exp ?? exp exp" }
infix       −100        ?\ ide          −100  ;       { in "exp ?\ ide exp" }
infix       101         ,               100   ;
infix       201         ::              200   ;
infix       170         :=              170   ;
infix       170         update          170   ;
infix       170         sub             170   ;
infix       400         or              400   ;
infix       500         &               500   ;
infix       700         =               700   ;
infix       700         <>              700   ;
infix       700         >               700   ;
infix       700         <               700   ;
infix       700         >=              700   ;
infix       700         <=              700   ;
infix       800         +               800   ;
infix       800         −               800   ;
infix       900         *               900   ;
infix       900         /               900   ;
infix       900         mod             900   ;
infix       1400        o               1400  ;
infix       1500                        1500  ;       { function application }

suffix      150         : type                ;       { in "exp : type" }
suffix      1300        . ide                 ;       { in "exp . ide" }
suffix      1300        as ide          ;     { in "exp as ide" }
suffix      1300        is ide                ;       { in "exp is ide" }

type infix  1           ,               0     ;       { argument pairing }
type infix  101         −>              100   ;
type infix  201         +               200   ;
type infix  301         #               300   ;
type infix  500                         500   ;       { parameter application }

type suffix 400         list                  ;
type suffix 400         ref                   ;
type suffix 400         array                 ;
```

Appendix F. Metasyntax

Strings between quotes ‘"’ are terminals.

Identifiers are non-terminals.

Juxtaposition is syntactic concatenation.

‘|’ is syntactic alternative.

‘[ ]’ is the empty string.

‘[..]’ is zero or one times (i.e. optionally) ‘ .. ’.

‘{ .. }n’ is n or more times ‘ .. ’ (default n=0).

‘{ .. / -- }n’ means n (default 0) or more times ‘..’ separated by ‘--’.

Parentheses ‘(..)’ are used for precedence.

Appendix G. Syntax of lexical entities

```
Letter ::=
  "a" | .. | "z" | "A" | .. | "Z" | "'" | "_".

Digit ::=
  "0" | .. | "9".

Symbol ::=
  "!" | "#" | "$" | "&" | "+" | "-" | "/" | ":" | "<" | "=" | ">" |
  "?" | "@" | "\" | "^" | "~" | "|" | "*".

Character ::= .. (see appendix "Lexical classes" for a list of legal characters)

Ide ::=
  Letter {Letter | Digit} |
  {Symbol}1.

Integer ::=
  {Digit}1.

String ::=
  """ {Character} """.

TypeIde ::=
  Letter {Letter | Digit} |
  {Symbol}1.

TypeVar ::=
  {"'"}1 {Letter | Digit}.

LeftPar ::=
  ("(" | "[") {Symbol}.

RightPar ::=
  {Symbol} (")" | "]").
```

Note1: Not all the sequences of Symbol characters are valid. Symbol characters are grouped into three classes:

```
0:  $ ^ | & /
A:  − < + : = # >
B:  * ? ~ @ ! \
```

In forming an identifier, every adjacent pair of Symbol characters must ‘stick’; two characters stick if they belong to the same class or if at least one is of class 0, i.e. they will not stick only if one is of class A and the other one of class B. The effect of these complex rules is to allow to use operators like "−−>", and to avoid having to insert too many blanks to separate lexical entities.

Note2: Only identifiers starting (or ending) with special characters of class 0 stick to the inner side of parentheses to form compound left (or right) parentheses like (|, |), [^\, /^], etc.

Appendix H. Syntax

```
TopTerm ::=
  [Term | SimpleDecl | Module | Use | Monitor] ";".

SimpleDecl ::=
  "val" ValDecl |
  "type" TypeDecl |
  "export" ExportList "from" Decl "end"|
  Decl "with" Decl "end" |
  "import" Decl |
  LexicalDecl.

Module ::=
  "module" ModuleName ["includes" {ModuleName}1] "body" Decl "end".

Use ::=
  "use" String.

Monitor ::=
  "monitor".

Term ::=
  Ide |
  "(" [Term] ")" |
  Integer |
  String |
  Term "," Term |
  "[" {Term / ";"} "]" |
  "(|" {Ide [ "=" Term ] / ";"} "|)" |
  "[|" Ide [ "=" Term ] "|]" |
  {"if" Term "then" Term}1 "else" Term |
  "while" Term "do" Term |
  "fun" Bind "." Term |
  Term Term |
  "let" Decl "in" Term "end" |
  Term "where" Decl "end" |
  Term ("." | "is" | "as") Ide |
  "case" Term "of" "[|" {{Ide / ","}1 [ "=" Bind ] "." Term / ";"} "|]" |
  Term ":" Type |
  PrefixOp Term |
  Term InfixOp Term |
  Term SuffixOp |
  "fail" |
  "failwith" String |
  Term "?" Term |
  Term "??" Term Term |
  Term "?\" Ide "." Term.

Decl ::=
  SimpleDecl |
  Decl ";" Decl |
  "(" Decl ")".
```

```
ValDecl ::=
  (Bind | FunBind [":" Type]) "=" Term |
  ValDecl "and" ValDecl |
  "rec" ValDecl.

ExportList ::=
  {"type" {Ide / ","}1 | "val" {Ide / ","}1}.

FunBind ::=
  Ide CurryBind |
  PrefixOp TopBind [CurryBind] |
  TopBind InfixOp TopBind [CurryBind] |
  TopBind SuffixOp [CurryBind].

CurryBind ::=
  {TopBind}1.

TopBind ::=
  Ide |
  "(" [Bind] ")" |
  TopBind "," TopBind |
  "[" {Bind / ";"} "]" |
  TopBind "::" TopBind |
  "(|" {Ide ["=" Bind] / ";"} "|)" |
  "[|" Ide ["=" Bind] "|]" |
  "ref" Bind.

Bind ::=
  Ide |
  "(" [Bind] ")" |
  Bind "," Bind |
  "[" {Bind / ";"} "]" |
  Bind "::" Bind |
  "(|" {Ide ["=" Bind] / ";"} "|)" |
  "[|" Ide ["=" Bind] "|]" |
  "ref" Bind |
  Bind ":" Type.

Type ::=
  TypeVar |
  [TypeArgs] TypeIde |
  PrefixTypeOp TypeArgs |
  TypeArgs InfixTypeOp TypeArgs |
  TypeArgs SuffixTypeOp |
  "(|" {{Ide / ","}1 [":" [Type]] / ";"} "|)" |
  "[|" {{Ide / ","}1 [":" [Type]] / ";"} "|]" |
  "(" Type ")".

TypeArgs ::=
  Type |
  "(" {Type / ","}1 ")".

TypeDecl ::=
  TypeBind ("=" | "<=>") Type |
```

```
  TypeDecl "and" TypeDecl |
  "rec" TypeDecl.

TypeBind ::=
  [TypeParams] TypeIde |
  PrefixTypeOp TypeParams |
  TypeParams InfixTypeOp TypeParams |
  TypeParams SuffixTypeOp.

TypeParams ::=
  TypeVar |
  "(" {TypeVar / ","}1 ")".

LexicalDecl ::=
  "nonfix" ["type"] Ide |
  "prefix" ["type"] Ide [Integer] |
  "infix"  ["type"] [Integer] Ide [Integer] |
  "suffix" ["type"] [Integer] Ide.

PrefixOp ::= Ide.

InfixOp  ::= Ide.

SuffixOp ::= Ide.

PrefixTypeOp ::= TypeIde.

InfixTypeOp  ::= TypeIde.

SuffixTypeOp ::= TypeIde.

ModuleName ::= Ide | String.
```

Notes:

- The top-level command monitor provides a menu of check-prints for monitoring the inner workings of the compiler.
- The top-level command use **filename** loads a file of ML definitions and expressions.
- Not all the sequences of special characters are legal identifiers (see appendix "Syntax of lexical entities").
- No error is given because of a misplaced fix operator: when a fix operator is out of place it is taken as a nonfix. Examples:

```
a+b;    +(a,b);    +;    [−; +; *; /];   ,,,;

val =(a,b) = a+b and ,(a,b) = a−b;

infix $;
val a $ b = ..
val $(a,b) = ..
val $ = fun a b. ..
val $ a b = ..
val (a,b) $ c = fun d. ..
val (a,b) $ c d = ..
```

- The fixity of term-identifiers is independent of the fixity of type-identifiers, e.g. ++ can be a prefix function and an infix type operator.
- The fixity of the identifiers '=', ',' and '::' cannot be changed, but their precedence can be redefined. Moreover '=' and ',' cannot be used as type identifiers.

Appendix I. Escape sequences for strings

The escape character for strings is \; it introduces characters according to the following code:

\1 .. \9	One to nine spaces
\0	Ten spaces
\R	Carriage return
\L	Line feed
\T	Horizontal tabulation
\B	Backspace
\E	Escape
\N	Null (Ascii 0)
\D	Del (Ascii 127)
\^c	Ascii control char ^c (c any appropriate char)
\c	c (for any other char c different from #)
\#**s**	128 + **s** (for any of the previous sequences **s**)

Strings are printed at the top level in the same form in which they are read; that is surrounded by quotes, with all the \ and with no non-printable characters. Output operations instead print strings in their crude form.

Appendix J. System installation

The ML system usually comes on a tape containing a single directory mlkit (and sometimes a backup copy of it mlkit.BACKUP). The tape can be read by a tar -x mlkit command, which creates an mlkit subdirectory in the current directory; mlkit can be kept in any user or system directory.

The mlkit directory contains several files and subdirectories. README repeates the information contained in this section. VERSION contains the system version. doc contains this manual in troff format, a paper on the ML abstract machine, and a list of known ML sites. src contains the source programs. progs contains some example ml programs. pub contains the ML object code, libraries and shell scripts.

To install the system enter the pub subdirectory, and read the install script to make sure that its execution is not going to cause any damage on your system. To run install you problably need write permission on /usr/lib and /usr/bin. When you are sure that everything is ok, type install.

Apart from install, pub contains an usr-bin-ml script and an usr-lib-ml subdirectory. The install procedure simply moves usr-bin-ml to /usr/bin/ml, and usr-lib-ml to /usr/lib/ml. usr-lib-ml contains the real ML executable code, called mlsys, a file which is loaded every time the system is entered, called lib, a precompiled library module, called lib.sp and lib.im, and the library module source, called lib.ml.

After install, the ML system can be run from any directory by typing ml. If you instead wish to call the system FORTRAN, say, just rename /usr/bin/ml to /usr/bin/FORTRAN.

If for some reason you need to recompile the system, you should proceed as follows. Most of the work is done by the Makefile file supplied with the source programs (say: make in mlkit/src). This generates an mlsys executable file which should be stripped (say: strip mlsys) and moved to /usr/lib/ml/mlsys

The new ML system is now available, but the ML library has to be recompiled. When loading the new system for the first time, an error message ‘Module /usr/lib/ml/lib must be compiled’ will appear. Immediately after, type ‘use "/usr/lib/ml/lib.ml";’: this will generate a new library for all the future uses of the new system. If you wish to work in the system at this moment, you should also type ‘use "/usr/lib/ml/lib";’ to actually load the library (this will happen automatically from now on).

The library module can be changed at will. It contains some basic ML functions, and can be extended with the locally most used utilities. After editing the file /usr/lib/ml/lib.ml, enter ML and type ‘use "usr/lib/ml/lib.ml";’ to compile and install the new library. You should also type ‘use "/usr/lib/ml/lib";’ to actually load the new library in the current session.

Appendix K. Introducing new primitive operators

This appendix describes how to add to the ML system a new function "foo: t" that cannot be otherwise defined in ML. It has to be defined as a new primitive operation in the Functional Abstract Machine [Cardelli 83] and implemented in C, Pascal, Assembler, or any other language which respects VAX procedure call standards.

• File mlglob.h

- Add "OpFoo" to enumeration type "SMOperationT" IN ASCII ALPHABETIC ORDER!.
- Add "OpFoo: ();" to the case list of type "SMCodeT".
- Add "AtomFoo: AtomRefT" in section "{Parser Vars}".
- If foo is prefix (infix, etc.) add "SynOpFoo" to the enumeration type "SynPrefixOpClassT" ("SynInfixOpClassT", etc.).

• File mlscan.p

- Add "AtomFoo := TableInsertAtom(3,'foo');" in procedure "SetupTable" (where "3" is the length of "foo").
- Add "PushSynRoleIde(AtomFoo);" in the same procedure (assuming that foo is nonfix, else use PushSynRoleInfix, etc. as appropriate, with second argument "SynOpFoo").

• File mlanal.p

- Add "Predefined(AtomFoo,OpFoo,n,t);" to the procedure "SetupEnvironment", where foo: t (you can understand how to express t from the code of SetupEnvironment), and n is the arity of foo (i.e. the number of arguments it expects on the stack).

• File mldebg.p

- Add "SetupMon(EmitSimpleOp(OpFoo));" to the procedure "SetupEnvValues" (if foo is not monadic, use SetupBin etc. Monadic, diadic, etc means that foo takes 1,2, etc. arguments on the top of the stack and returns a result there, after popping the arguments). It is essential that the order of setups in this procedure matches the order of definitions in SetupEnvironment (in mlanal.p).

• File mlconv.p

- Add "OpFoo: Operand:=0{nil};" to the case list in function "ConvertOperand".

• File amglob.h

- Add "#define OpFoo n" IN ASCII ALPHABETIC ORDER in section "Fam OpCodes"; make sure that all the opcodes are SEQUENTIALLY NUMBERED.
- Add "extern Address DoFoo();" in section "Externals".
- If foo may fail with string "foo", declare "extern Address *StrFoo;" in section "FailStrings".

• File amevalop.c

- Add "OpFoo: CallOp((Address)DoFoo,n); break;" to the case list in procedure "Assem", where n is the arity of Foo (i.e. how many arguments it expects on the stack).

• File ammall.c

- If foo may fail with string "foo", declare "Address *StrFoo;" and add "StrFoo = PushGCBox(StringFromC("foo"));" to the procedure "AllocFailStrings".

• File amdebg.c

- Add "case OpFoo: printf("Foo"); break;" to the case list of procedure "AMStatPrint".

• File amfooop.c

- Create this file, containing the C function "Address DoFoo(argn, .. ,arg1)", implementing the desired behavior. The order of arguments must be the inverse of the order in which "foo(arg1, .. ,argn)" is called from ML. WARNING: do not attempt to allocate or change ML data structures in this function without deeply understanding how the garbage collector works. It is safe to inspect (read only) the ML data structures passed as parameters (see appendix "Vax data formats"), and to return them

unchanged. The result of this function must match the ML type declared in mlanal.p. If DoFoo needs to produce an ML failure, it can do so by calling "DoFailwith(*StrFoo)"

• File Makefile

- Add a line "FOOOP = amglob.h amfooop.c".
- Add "amfooop.o" to the definition of "MLSYS".
- Add a line "amfooop.o: $(FOOOP); CC amfooop.c".

Appendix L. System limitations

At the moment there is a limit on the size of any single ML object, due to storage allocation problems. This limit is determined by the constant "PageSize" in the file "amglob.h". All the limitations mentioned on this section are further constrained by this restriction.

The semantic limit on the length of strings is 64K chars.

The syntactic limit on the length of string quotations in a source program is also 64K chars.

Several memory areas inside the system have fixed size. When one of these is exceeded, compilation fails and a message is printed. The only way to fix this is to search for the point in the source program where the error message is generated, increase the corresponding area (usually by redefining a constant) and recompile the system.

Appendix M. VAX data formats

Each segment "+−−+" in the pictures is one byte. The symbol "^" below a data structure represents the location pointed to by pointers to that structure; fields preceding "^" are only used during garbage collection and are inaccessible to the ML operations (and hence to the user). Unboxed data is kept on the stack, or in the place of pointers in other data structures; unboxed data does not require storage allocation. Pointers can be distinguished from unboxed data as the former are > 64K. There are automatic conversions between SmallIntegers and BigIntegers, so that the ML operations only see the type int.

Unit

0 (unity) (unboxed)

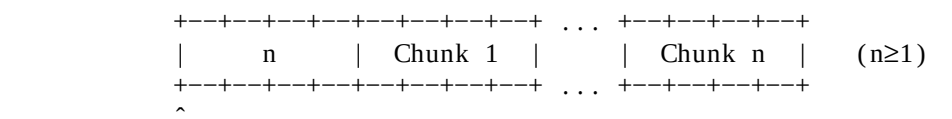
Boolean

0 (false) (unboxed)
1 (true) (unboxed)

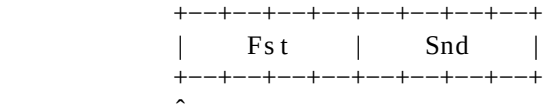
SmallInteger

−32768 .. +32767 (unboxed)

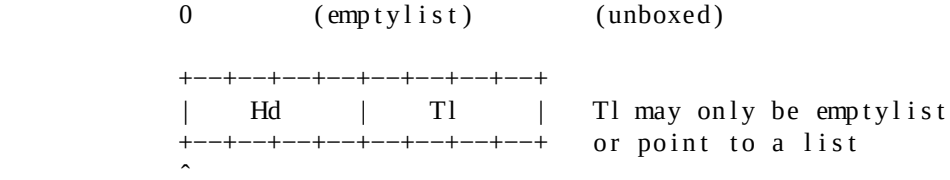
BigInteger



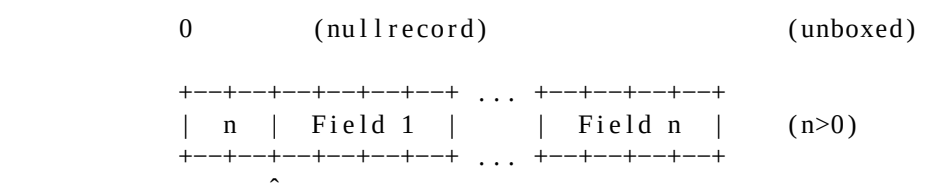
Pair



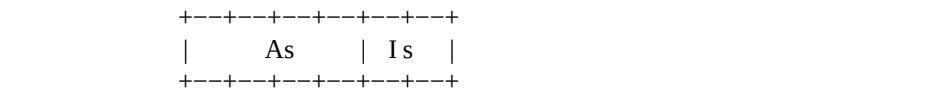
List



Record

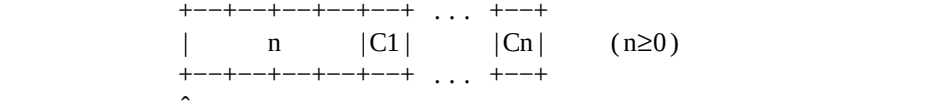


Variant

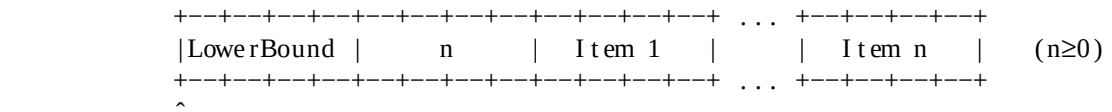


Reference

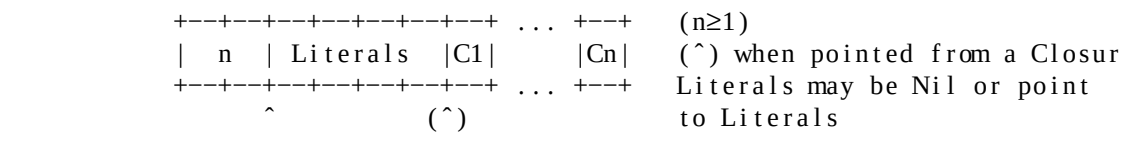
String



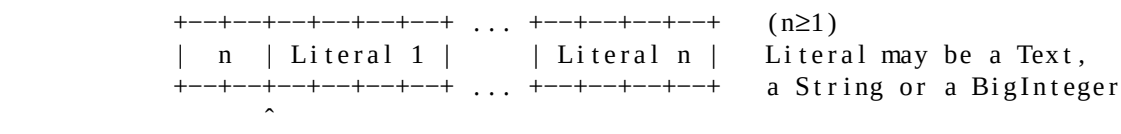
Array



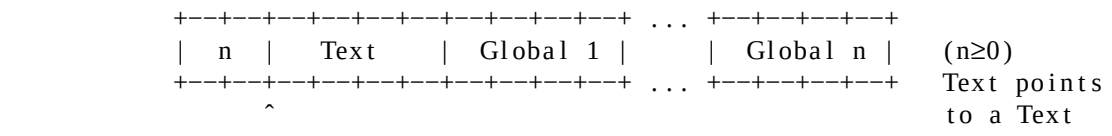
Text



Literals



Closure



Appendix N. Ascii codes

0 nul	1 soh	2 stx	3 etx	4 eot	5 enq	6 ack	7 bel	
8 bs	9 ht	10 nl	11 vt	12 np	13 cr	14 so	15 si	
16 dle	17 dc1	18 dc2	19 dc3	20 dc4	21 nak	22 syn	23 etb	
24 can	25 em	26 sub	27 esc	28 fs	29 gs	30 rs	31 us	
32 sp	33 !	34 "	35 #	36 $	37 %	38 &	39 ’	
40 (	41)	42 *	43 +	44 ,	45 −	46 .	47 /	
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?	
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O	
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	
88 X	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _	
96 ‘	97 a	98 b	99 c	100 d	101 e	102 f	103 g	
104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o	
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	
120 x	121 y	122 z	123 {	124 |	125 }	126 ~	127 del	

Appendix O. Transient

Several minor problems still afflict modules. These will be fixed gradually in the short and medium term. Meanwhile the following things should be noted.

With the exception of the library functions in /usr/lib/ml/lib.ml, modules must not use predefined functions as argumentless functional objects, e.g. + in fold (+) [1;2;3;4] 0. At the moment this will produce and 'Unbound identifier' error. However, it is possible to replace (+) by (fun(x,y). x+y).

Under some complicated circumstances the sharing of types will fail. The scenario is: module A privately imports module B, and then exports a type T defined in B by an export export list; then module C imports A and B and makes T-from-A interact with T-from-B. This will very likely never happen to you. However, if you get an unexplained type conflict (probably two different abstract types having the same name), this might be it. You should be able to fix this by making sure that that type is exported by includes declarations, as opposed to export export lists.

If a module A is recompiled, all the modules depending on it must be recompiled, even if the interface of A has not changed.

Sometimes, when a module B which imports A has to be recompiled, instead of the message Module B must be compiled, the less precise message Module importing A must be compiled is produced.

There is no check on the restriction that modules exporting a type identifier (maybe as part of the type of an exported binder), must also export the type definition. However, nothing bad can of from this.

On some occasions, precompiled modules can give undefined type errors when importing them, or when compiling modules which import them. This can happen if an export declaration rearranges the order of types and values in the body of a module. To fix this, put the export lists back in an appropriate dependency order. You can look at the .sp file for that module to see what is happening. Nothing bad can come from this.

Consider the following scenario: module A is defined; module B, which imports A, is defined; module B is imported; module A is redefined. At this point module B is obsolete, and one would expect a version error in importing B again. This would happen if module B had not already be imported; however an import B at this point will share the already imported module B, without any complaint. At this point we have a B including the old A, while we might expect the new A to be in action. To get the new A into play, you must recompile B and reimport it. Nothing bad can come of this, except confusion.

Input/Output. The current plans are to implement an efficient buffering with in-core cache for internal streams, and to use unbuffered 'raw' I/O on external streams. Internal streams would still look unbuffered to the user, but all the disk I/O would be done in chunks. The current situation is fluid.

At the moment a limited number of streams can be open in an ML session and they cannot be closed. When this limit is exceeded, the operations NewStream, GetStream and CopyStream fail. This problem will be solved in a future version by an automatic cache of open streams.

The operations implemented so far are getstream, putstream, newstream, emptystream, instring, outstring and terminal (not yet working: lookahead and channel).

The failure "corruption" is produced when something goes very wrong (e.g. after a very deep recursion which overruns the stack); the system is currently unsafe after a corruption. You should rarely be thrown out of the system, but on extreme corruptions you may get a Run out of memory message and a core dump.

On end of stream, inchar waits forever; type **DEL** to exit input waits and infinite loops (a trappable failure "interrupt" is produced) (ignore the longjmp botch and System Crash! messages which are sometimes generated).

Bignums are not yet implemented. Integers currently range between ˜32768 and 32767.

Deep recursions producing stack overflow will badly corrupt the system; (this is the most likely cause of crash). The ml failure "corruption", or the message System Crash! may be generated. If you come across this problem, try to rewrite your programs in a linear recursive style, so that the compiler can optimise them to iterations, or directly in iterative style using while-do. This problem cannot be solved easily, given the current Unix memory management primitives.

References

[1] L.Cardelli. "The Functional Abstract Machine", Bell Labs Technical Memorandum TM-83-11271-1, Bell Labs Technical Report TR-107, 1983.

[2] M.Gordon, R.Milner, C.Wadsworth. "Edinburgh LCF", Lecture Notes in Computer Science, n.78, Springer-Verlag, 1979.

[3] R.Milner. "A theory of type polymorphism in programming", Journal of Computer and System Science 17, 348-375, 1978.