ML

Introduction and examples

ML is an interactive language. The system repeatedly prompts for input and reports the results of computations; this interaction is said to happen at the top level of evaluation. At the top level one can evaluate expressions or perform declarations. To give a first impression of the system, we reproduce below a session at a terminal in which simple uses of various ML constructs are illustrated. To make the session easier to follow, it is split into a sequence of sub-sessions displayed in boldface. Each sub-session is accompanied by an explanation; the complete session consists of the concatenation of the boldface areas. A complete description of the syntax of ML is given in ---, and of the semantics in ---.

Expressions

The ML prompt is '- ', and so lines beginning with this contain the user's contribution; all other lines are output by the system.

```
- 2+3;
  5 : int

- it;
  5 : int
```

ML prompted with '- '; the user then typed '2+3;' followed by a return; ML then responded with ' 5 : int', a new line, and then prompted again. The user then typed 'it;' followed by a return, and the system responded by typing ' 5 : int' again. In general to evaluate an expression e one types 'e;' followed by a return; the system then prints e's value and type. The value of the last expression evaluated at top level is remembered in the identifier 'it'.

Declarations

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

```
- let x = 2*3;
> x = 6 : int

- it = x;
  false : bool
```

The prefix '> ' indicates that a new declaration is taking place, as opposed to an evaluation which is prefixed by ' '. Notice that declarations do not effect 'it'. To bind $x_1,...,x_n$ simultaneously to the values of $e_1,...,e_n$ one can perform either the declaration 'let $x_1=e_1$ and $x_2=e_2$ ... and $x_n=e_n$' or, equivalently, the declaration 'let $x_1,...,x_n = e_1,...,e_n$'. In the first case we use a environment operator (or declaration operator) 'and', while in the second case we use a structured variable (or varstruct) '$x_1,...,x_n$'.

```
- let y = 10 and z = x;
> y = 10 : int
| z = 6 : int

- let x,y = y,x;
> x = 10 : int
| y = 6 : int
```

Note that the declaration prefix '> ' converts to '| ' after the first definition. Cascaded declarations are obtained by the environment operator 'enc' (enclose) which makes earlier declarations available in later declarations, and has otherwise the same effect as 'and'.

```
- let x = 10 enc y = x+5;
> x = 10 : int
| y = 15 : int

- let x = 5 and y = x+5;
> x = 5 : int
| y = 15 : int
```

Private declarations are obtained by the environment operator 'ins' (inside) which makes a declaration available inside another declaration, but not anywhere else.

```
- let x = 7 ins y = x+5;
> y = 12 : int

- x;
  5 : int
```

Complex declarations can be bracketed by '{' and '}'; otherwise the 'and' operator binds stronger than 'enc' and 'ins' (which have the same binding power) and all three operators are right associative. In the following example declaration brackets make a difference.

```
- let x = 10 and {z = 5 ins y = 10+z};
> x = 10 : int
| y = 15 : int
```

A declaration d can be made local to the evaluation of an expression e by evaluating 'let d in e'. The expression 'e where d' is equivalent to 'let d in e'.

```
- let x = 2 in x*y;
  30 : int

- x;
  10 : int

- x*y where x = 2;
  30 : int
```

Functions

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

```
- let f x = 2*x;
> f = \ : int -> int

- f 4;
  8 : int
```

Functions are printed as a '\' followed by their type ('\' is chosen as an ascii approximation of the greek letter lambda). Application binds tighter than anything else in the language; thus, for example, 'f 3 + 4' means '(f 3)+4' not 'f(3+4)'. Functions of several arguments can be defined:

```
- let add x y = x+y;
> add = \ : int -> (int -> int)

- add 3 4;
  7 : int

- let f = add 3;
> f = \ : int -> int

- f 4;
  7 : int
```

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

```
- let add(x,y) = x+y;
> add = \ : (int # int) -> int

- add(3,4);
  7 : int

- let z = (3,4) in add z;
  7 : int

- add 3;
Type Clash  in:  (add 3)
Looking  for:    int # int
I have found:    int
```

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

```
- let sumdiff(x,y) = (x+y,x-y);
> sumdiff = \ : (int # int) -> (int # int)
```

```
- sumdiff(3,4);
  (7,~1) : int # int
```

Incidentally, note that the unary negation operation on numbers is '~' instead of '-'; hence one should write '~3' for negative numbers and '~(n-1)' for the complement of n-1.

Recursion

The following is an attempt to define the factorial function:

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

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

```
- let f n = n+1;
> f = \ : int -> int

- let f n = if n=0 then 1 else n*f(n-1);
> f = \ : int -> int

- f 3;
  9 : int
```

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

```
- let rec fact n = if n=0 then 1 else n*fact(n-1);
> fact = \ : int -> int

- fact 3;
  6 : int
```

'rec' is another environment operator like 'and', 'enc' and 'ins'; it can be nested inside complex declarations, and it binds more weakly than 'and' but more strongly that 'enc' and 'ins'.

Assignment and sequencing

Assignment operations act on reference objects. A reference is an updateble pointer to an object. References are 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 the trivial value '()' (called triv), which is the only object of the trivial type ':' (also called triv).

```
- let a = ref 3;
> a = (ref 3) : int ref

- a:=5;
  () : .

- !a;
  5 : int
```

When several side-effecting operations have to be executed in sequence, it is useful to use sequencing '(e_1; ... ;e_n)' (the parenthesis are needed), which evaluates e_1 ... e_n in turn and returns the value of e_n.

```
- (a:=!a+1; a:=!a/2; !a);
  3 : int
```

Iteration

The construct 'if e_1 then e_2 elseloop e_3' is the same as 'if e_1 then e_2 else e_3' in the true case; when e_1 evaluates to false, e_3 is evaluated and control loops back to the front of the construct again. As an illustration here is an iterative definition of 'fact' which uses two local assignable variables: 'count' and 'result' (note that the prompt '-' changes to '=' when an expression spans several lines).

```
- let fact n =
=     let count = ref n and result = ref 1
=     in     if !count=0
```

```
=           then !result
=           elseloop (result := !count * !result;
=                     count := !count-1);
> fact = \ : int -> int

- fact 4;
  24 : int
```

The 'then' in 'if e1 then e2 else e3' may be replaced by 'thenloop' to cause iteration when e1 evaluates to true. Thus 'if e1 thenloop e2 else e3' is equivalent to 'if not(e1) then e3 elseloop e2'. The conditional/loop construct can have a number of conditions, each preceded by 'if'; the expression guarded by each condition may be preceded by 'then', or by 'thenloop' when the whole construct is to be reevaluated after evaluating the guarded expression:

```
- let gcd(x,y) =
=     let x,y = ref x, ref y
=     in      if !x>!y thenloop x:=!x-!y
=             if !x<!y thenloop y:=!y-!x
=             else !x;
> gcd = \ : (int # int) -> int

- gcd(12,20);
  4 : int
```

The 'else' branch must always be present in normal conditionals and in the iterative forms.

Lists

If e1,...,en all have type ty then the ML expression '[e1;...;en]' has type 'ty list'. The standard functions on lists are 'hd' (head), 'tl' (tail), 'null' (which tests whether a list is empty - i.e. is equal to '[]' (nil)), and the infixed operators '::' (cons) and '@' (append, or concatenation).

```
- let m = [1;2;(2+1);4];
> m = [1;2;3;4] : int list

- hd m, tl m;
  (1,[2;3;4]) : int # (int list)

- null m, null [];
  (false,true) : bool # bool

- 0::m;
  [0;1;2;3;4] : int list

- [1;2] @ [3;4;5;6];
  [1;2;3;4;5;6] : int list

- [1;true;2];
Type Clash  in:  [1;true;2]
Looking  for:    int
I have found:    bool
```

All the members of a list must have the same type (although this type could be a sum, or disjoint union, type - see 2.4).

Tokens

A sequence of characters in token quotes (`) is a token.

```
- `this is a token`;
  `this is a token` : tok

- "this is a token list";
  "this is a token list" : tok list
```

```
- it = ("this is a" @ ['token';'list']);
  true : bool
```

The expression "tok1 tok2 ... tokn" is an alternative syntax for ['tok1'; 'tok2'; ... ;'tokn'].

Polymorphism

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

```
- hd [1;2;3];
  1 : int

- hd [true;false;true];
  true : bool

- hd "this is a token list";
  'this' : tok
```

Thus 'hd' has more than one type, for example above it is used with types '(int list) -> int', '(bool list) -> bool' and '(tok list) -> tok'. In fact if ty is any type then 'hd' has the type '(ty list) -> ty'. Functions, like 'hd', with many types are called polymorphic , and ML uses type variables '*a', '*b', '*1', '*2', '*', '**', '***' etc to represent their types.

```
- hd;
  \ : (*a list) -> *a

- let rec map f l =
=    if null l then []
=              else f(hd l)::map f (tl l);
> map = \ : (*a -> *b) -> ((*a list) -> (*b list))

- map fact [1;2;3;4];
  [1; 2; 6; 24] : int list
```

map takes a function f (with argument type *a and result type *b), and a list l (of elements of type *a), and returns the list obtained by applying f to each element of l (which is a list of elements of type *b). map can be used at any instance of its type: above, both *a and *b were instantiated to int; below, *a is

instantiated to (int list) and *b to bool. Notice that the instance need not be specified; it is determined by the typechecker.

```
- map null [[1;2]; []; [3]; []];
  [false; true; false; true] : bool list
```

Lambda-expressions

The expression '\x.e' evaluates to a function with formal parameter x and body e. Thus 'let f x = e' is equivalent to 'let f = \x.e'. Similarly 'let f(x,y)z = e' is equivalent to 'let f = \(x,y).\z.e'. Repeated '\'s, as in '\(x,y).\z.e', may be abbreviated by '\(x,y)z.e'. The character '\' is our representation of lambda, and expressions like '\x.e' and '\(x,y)z.e' are called lambda-expressions.

```
- \x.x+1;
  \ : int -> int

- it 3;
  4 : int

- map (\x.x*x) [1;2;3;4];
  [1;4;9;16] : int list

- let doubleup = map (\x.x@x);
> doubleup = \ : ((*a list) list) -> ((*a list) list)

- doubleup ["a b";"c"];
  ["a b a b";"c c"] : (tok list) list

- doubleup [[1;2];[3;4;5]];
  [[1;2;1;2];[3;4;5;3;4;5]] : (int list) list
```

Failure

Some standard functions fail at run-time on certain arguments, yielding a token (which is usually the function name) to identify the sort of failure. A failure with token 't' may also be generated explicitly by evaluating the expression 'failwith 't" (or more generally 'failwith e' where e has type tok).

```
- hd(tl[2]);
Failure: hd

- 1/0;
Failure: /

- (1/0)+1000;
Failure: /

- failwith (hd "this is a token list");
Failure: this
```

A failure can be trapped by '?'. The value of the expression 'e1?e2' is that of e1, unless e1 causes a failure, in which case it is the value of e2.

```
- hd(tl[2]) ? 0;
  0 : int

- (1/0)?1000;
  1000 : int

- let half n =
=     if n=0 then failwith 'zero'
=           else let m=n/2
=                in  if n=2*m then m else failwith'odd';
> half = \ : int -> int
```

The function half only succeeds on non-zero even numbers; on 0 it fails with 'zero', and on odd numbers it fails with 'odd'.

```
- half 4;
  2 : int

- half 0;
Failure: zero

- half 3;
Failure: odd
```

```
- half 3 ? 1000;
  1000 : int
```

Failures may be trapped selectively (on token) by '??'; if e1 fails with token "t", then the value of 'e1 ??"t1 ... tn" e2' is the value of e2 if t is one of t1,...,tn, otherwise the expression still fails with "t".

```
- half 0 ?? "zero plonk" 1000;
  1000 : int

- half 1 ?? "zero plonk" 1000;
Failure: odd
```

One may add several '??' traps to an expression, and one may add a '?' trap at the end as a catchall.

```
- half 1
=    ?? "zero" 1000
=    ?? "odd"  2000;
  2000 : int

- hd(tl[half(4)])
=    ?? "zero" 1000
=    ?? "odd"  2000
=    ? 3000;
  3000 : int
```

Defined Types

Types can be given names:

```
- let type intpair = int # int;
> type intpair = int # int

- let p = 12,20;
> p = (12,20) : int # int

- p : intpair;
  (12,20) : intpair
```

The new name is simply an abbreviation; for example, 'intpair' and 'int # int' are completely equivalent. The system may use any of these equivalent type names when printing types, but tries to maintain the names which have been explicitely forced by ':'.

Abstract Types

New types (rather than mere abbreviations) can also be defined. For example, to define a type 'time' we could do:

```
- let type time <=> int # int
= with maketime(hrs,mins) =
=         if  hrs<0 Or 23<hrs Or mins<0 Or 59<mins
=         then fail
=         else abstime(hrs,mins)
= and  hours t = fst(reptime t)
= and  minutes t = snd(reptime t);
> type time = -
| maketime = \ : (int # int) -> time
| hours = \ : time -> int
| minutes = \ : time -> int
```

This declaration defines an abstract (i.e. new) type 'time' together with three primitive functions: 'maketime', 'hours' and 'minutes'. In general an abstract type declaration has the form 'type ty <=> ty1 with d' where d is a declaration, i.e. the kind of phrase that can follow 'let'. Such a declaration introduces a new type ty which is represented by ty1. Only within d can one use the (automatically

declared) functions absty (of type ty'->ty) and repty (of type ty->ty'), which map between a type and its representation. In the example above 'abstime' and 'reptime' are only available in the definitions of 'maketime', 'hours' and 'minutes'; these latter three functions, on the other hand, are defined throughout the scope of the declaration. Thus an abstract type declaration simultaneously declares a new type together with primitive functions for the type. The representations of the type (i.e. ty'), and of the primitives (i.e. the right hand sides of the definitions in b), are not accessible outside the with-part of the declaration.

```
- let t = maketime(8,30);
> t = - : time

- hours t, minutes t;
  (8,30) : int # int
```

Notice that values of an abstract type are printed as '-'. When interacting with the system, it is useful to be able to define abstract types incrementally in order to make experiments with them. This can be achieved by an open ended '<=>' declaration, where the 'with' part is omitted.

```
- let type time <=> int # int;
> type time = -
| abstime = \ : (int # int) -> time
| reptime = \ : time -> (int # int)
```

Now 'abstime' and 'reptime' are available at the top level and can be used to define 'maketime', 'hours' and 'minutes' as before. The 'with' construct is not a special syntax for abstract types: it is an environment operator which behaves like 'ins' on values (making 'abstime' and 'reptime' private) and like 'enc' on types (making 'time' available). Hence abstract types are obtained from the interaction of two orthogonal features: the isomorphism type constructor '<=>' and the environment operators.

Type Operators

Both 'list' and '#' are examples of type operators; 'list' has one argument (hence '*a list') whereas '#' has two (hence '*a # *b'). Each type operator has various primitive operations associated with it, for example 'list' has 'null', 'hd', 'tl',... etc, and '#' has 'fst', 'snd' and the infix ','.

```
- let z = it;
> z = 8,30 : int # int

- fst z;
  8 : int

- snd z;
  30 : int
```

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

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

These are illustrated by:

```
- let x = inl 1
= and y = inr 2;
> x = inl 1 : int + *a
| y = inr 2 : *b + int

- isl x;
  true : bool

- isl y;
  false : bool

- outl x;
  1 : int

- outl y;
Failure: outl
```

```
- outr x;
Failure: outr

- outr y;
  2 : int
```

The abstract type 'time' defined above can be thought of as a type operator with no arguments (i.e. a nullary operator); its primitives are 'maketime', 'hours' and 'minutes'. The 'type...<=>...with...' construct may also be used to define non-nullary type operators (with 'rec type' in place of 'type' if these are recursive). For example trees analogous to LISP S-expressions could be defined by:

```
- let rec type * sexp <=> * + (* sexp) # (* sexp)
=  with cons(s1,s2) = abssexp(inr(s1,s2))
=  and  car s = fst(outr(repsexp s))
=  and  cdr s = snd(outr(repsexp s))
=  and  atom s = isl(repsexp s)
=  and  makeatom a = abssexp(inl a);
> type *a sexp = -
| cons = \ : (*a sexp) # (*a sexp) -> *a sexp
| car = \ : *b sexp -> *b sexp
| cdr = \ : *c sexp -> *c sexp
| atom = \ : *d sexp -> bool
| makeatom = \ : *e -> *e sexp
```

Exercise: introduce references in the above definition of sexp, so that the LISP operations 'replaca' and 'replacd' can be defined. Exercise: modify the definition of sexp to obtain lazy S-expressions. (Hints: use functional objects to implement suspensions which prevent early evaluation, and use references to ensure that suspensions are evaluated at most once.)

Records

A record is very similar to a tuple '$(x_1,...,x_n)$' where each x_i is given a different label; then the order of the x_i is not important because every component can be identified by its label.

```
- let r = (|a=3; c="1"; b=true|);
> r = (|a=3; b=true; c="1"|) : (|a:int; b:bool; c:tok list|)
```

```
- r = (|c="1"; b=true; a=3|);
  true : bool
```

The special brackets '(|' and '|)' are used to delimit records and record types. Note how the labels 'a', 'b' and 'c' are rearranged in alphabetical order by the system. The only operation defined on records is field selection 'r.a' where a is a label and r has a record type.

```
- r.a;
  3 : int

- (|a=r.a; b=r.b|);
  (|a=3; b=true|) : (|a:int; b:bool|)
```

Labels are not identifiers and cannot be computed. In the expression '\a. r.a' the first 'a' is a variable while the second 'a' is a label; it is not possible to parameterize with respect to a label. Also, labels are not tokens.

```
- \a. r.a;
  \ : *a -> int

- it ();
  3 : int
```

A record field of the form 'a=a' (where the first 'a' is a label and the second one is a variable) can be abbreviated to 'a'; this is useful because often one uses variables having the same name as record fields for mnemonic reasons. The same applies to record types.

```
- let {type R = (|bool; int|)}
= and f(x,y) = (|x; y|);
> type R = (|bool:bool; int:int|)
| f = \ : (*a # *b)  -> (|x:*a; y:*b|)
```

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

information about 'r'.

```
- \r. r.a;
Unresolvable Record Selection of type:  (|a:*a|)
Field selector is:                      a

- (\r. r.a) r;
  3 : int
```

Variants

Variants and records can be considered as complementary concepts; a record type is a labelled product of types, while a variant type is a labelled 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 a finite set of possibilities.

```
- let type OneOfTwo = [|int: int; bool: bool|];
> type OneOfTwo = [|bool:bool; int:int|]
```

The special brackets '[|' and '|]' are used to delimit variants and variant types. Again, the labels 'bool' and 'int' are rearranged in alphabetical order (they should not be confused with the types 'bool' and 'int' which follow the ':'). Two basic operations are defined on variants: 'is' tests the label of a variant object, and 'as' returns the object contained in the variant.

```
- let v = [|int=3|] : OneOfTwo;
> v = [|int=3|] : OneOfTwo

- v is int, v is bool;
  (true,false) : bool # bool

- v as int;
  3 : int

- v as bool;
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:

```
- [|int=3|];
Unresolvable Variant Type:          [|int:int|]
```

A very useful abbreviation concerning varian types is the following: whenever we have a variant type with a field 'a: .' we can abbreviate that field specification to 'a', and whenever we have a variant object '[|a=()|]' we can abbreviate it to '[|a|]'.

```
- let type color = [|red; orange; yellow|];
=      and fruit = [|apple; orange; banana|];
> type color = [|orange; red; yellow|]
| type fruit = [|apple; banana; orange|]

- let fruitcolor (fruit: fruit): color =
=    case fruit of
=      [|apple.  [|red|];
=        banana. [|yellow|];
=        orange. [|orange|]
=      |];
> 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' is never executed; a compile-time error message is given if some case is missing). 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.

Varstructs

Structured variables (varstructs) can appear wherever a formal parameter is expected; we have already seen the simple case of tuples:

```
- let a,b,c = 1,2,3;
> a = 1 : int
| b = 2 : int
```

```
| c = 3 : int
```

All the variables in a varstruct must be distinct. so that no ambiguites can arise. Lists can also be used in varstructs, and they can be nested with tuples and with all the other kinds of varstructs:

```
- let [a;b] = [1;2];
> a = 1 : int
| b = 2 : int

- let a::b = [1;2;3];
> a = 1 : int
| b = [2;3] : int list

- let a,[();b,c] = 1,[2,3;4,5];
| a = 1 : int
| b = 4 : int
| c = 5 : int

- let [a;b] = [3];
Failure: varstruct
```

List varstructs may fail when the number of elements in a list is not the expected one. The varstrtuct '()' can be used to ignore parts of a structure. Record and variant varstructs are also allowed.

```
- let (|a=x; b=[|int=y|]|) = (|a=3; b=[|int=4|]:OneOfTwo|);
> x = 3 : int
| y = 4 : int
```

Varstructs can also appear as formal parameters of functions and lambda-expressions, and in case statements. Note that record varstructs in functions allow to pass arguments in random order by associating them to labels; this feature is called 'call by keyword' in some languages.

```
- let f [a;b,c] = a,b+c;
> f = \ : ((int # int) list) -> ((int # int) # int)

- (\(|a; b|). a,b) (|b=3; a=4|);
```

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

- let {type IntOrPair = [|int:int; pair:(|x:int; y:int|)|]}
= enc total (a: IntOrPair): int =
=        case a of
=          [|int=z. z;
=            pair=(|x; y|). x+y
=          |]
= in total [|pair= (|x=3; y=4|)|];
  7 : int
```

The varstruct '(|x; y|)' above is an abbreviation for '(|x=x; y=y|)', just like in record expressions.