An Overview of Structures in the Basis Library

The Basis Library contains a wide variety of useful structures. Here is an overview of some of the most important ones.

- **Option**
  Operations for the `option` type.
- **Bool**
  Operations for the `bool` type.
- **Char**
  Operations for the `char` type.
- **String**
  Operations for the `string` type.
- **Byte**
  Operations for the `byte` type.
- **Int**
  Operations for the `int` type.
- **IntInf**
  Operations for an unbounded precision integer type.
- **Real**
  Operations for the `real` type.
- **Math**
  Various mathematical values and operations.
- **List**
  Operations for the `list` type.
- **ListPair**
  Operations on pairs of lists.
- **Vector**
  A polymorphic type for immutable (unchangeable) sequences.
- **IntVector, RealVector, BoolVector, CharVector**
  Monomorphic types for immutable sequences.
- **IntArray, RealArray, BoolArray, CharArray**
  Monomorphic types for mutable sequences.
- **IntArray2, RealArray2, BoolArray2, CharArray2**
  Monomorphic 2 dimensional mutable types.
- **TextIO**
  Character-oriented text IO.
- **BinIO**
  Binary IO operations.
- **OS, Unix, Date, Time, Timer**
  Operating systems types and operations.

ML Type Inference

One of the most novel aspects of ML is the fact that it infers types for all user declarations.

How does this type inference mechanism work?

Essentially, the ML compiler creates an unknown type for each declaration the user makes. It then solves for these unknowns using known types and a set of type inference rules. That is, for a user-defined identifier `i`, ML wants to determine `T(i)`, the type of `i`. 
The type inference rules are:

1. The types of all predefined literals, constants and functions are known in advance. They may be looked-up and used. For example,
   
   2 : int
   true : bool
   [] : 'a list
   :: : 'a * 'a list -> 'a list

2. All occurrences of the same symbol (using scoping rules) have the same type.

3. In the expression
   
   I = J
   we know \( T(I) = T(J) \).

4. In a conditional
   
   (if \( E_1 \) then \( E_2 \) else \( E_3 \))
   we know that
   
   \( T(E_1) = \text{bool} \),
   \( T(E_2) = T(E_3) = T(\text{conditional}) \)

5. In a function call
   
   \( (f \ x) \)
   we know that if \( T(f) = 'a \rightarrow 'b \) then \( T(x) = 'a \) and \( T(f \ x) = 'b \)

6. In a function definition
   
   \( \text{fun} \ f \ x = \text{expr}; \)
   if \( T(x) = 'a \) and \( T(\text{expr}) = 'b \) then \( T(f) = 'a \rightarrow 'b \)

7. In a tuple \((e_1, e_2, \ldots, e_n)\)
   
   if we know that \( T(e_i) = 'a_i \leq i \leq n \) then \( T(e_1, e_2, \ldots, e_n) = 'a_1 * 'a_2 * \ldots * 'a_n \)

8. In a record
   
   \( \{ a=e_1, b=e_2, \ldots \} \)
   if \( T(e_i) = 'a_i \leq i \leq n \) then the type of the record =
   
   \( \{ a:'a_1, b:'a_2, \ldots \} \)

9. In a list \([v_1, v_2, \ldots v_n]\)
   
   if we know that \( T(v_i) = 'a_i \leq i \leq n \) then we know that
   
   \( 'a_1 = 'a_2 = \ldots = 'a_n \) and
   \( T([v_1, v_2, \ldots v_n]) = 'a_1 \text{ list} \)

To Solve for Types:

1. Assign each untyped symbol its own distinct type variable.
2. Use rules (1) to (9) to solve for and simplify unknown types.
3. Verify that each solution “works” (causes no type errors) throughout the program.

Examples

Consider

\( \text{fun} \ \text{fact}(n) = \)
   
   \( \text{if} \ n = 1 \ \text{then} \ 1 \ \text{else} \ n * \text{fact}(n-1); \)

To begin, we’ll assign type variables:

\( T(\text{fact}) = 'a \rightarrow 'b \) (\( \text{fact} \) is a function)
\( T(n) = 'c \)
Now we begin to solve for the types ’a’, ’b’ and ’c’ must represent.
We know (rule 5) that ’c = ’a since n is the argument of fact.
We know (rule 3) that ’c = T(1) = int since n=1 is part of the definition.
We know (rule 4) that T(1) = T(if expression)=’b since the if expression is the body of fact.
Thus, we have
’a = ’b =’c = int, so
T(fact) = int -> int
T(n) = int
These types are correct for all occurrences of fact and n in the definition.

A Polymorphic Function:

fun leng(L) =
  if L = []
  then 0
  else 1+len(tl L);
To begin, we know that
T([]) = ’a list and
T(tl) = ’b list -> ’b list
We assign types to leng and L:
T(leng) = ’c -> ’d
T(L) = ’e
Since L is the argument of leng,
’e = ’c
From the expression L=[] we know
’e = ’a list

Type Inference for Patterns

Type inference works for patterns too.
Consider

fun leng [] = 0
  | leng (a::b) = 1 + leng b;
We first create type variables:
T(leng) = ’a -> ’b
T(a) = ’c
T(b) = ’d
From leng [] we conclude that
’a = ’e list
From leng [] = 0 we conclude that
’b = int
From leng (a::b) we conclude that
’c = ’e and ’d = ’e list
Thus we have
T(leng) = ’e list -> int
T(a) = 'e
T(b) = 'e list
This solution is type correct throughout the definition.

Not Everything can be Automatically Typed in ML

Let’s try to type
fun f x = (x x);
We assume
T(f) = 'a -> 'b
t(x) = 'c
Now (as usual) 'a = 'c since x is the argument of f.
From the call (x x) we conclude that 'c must be of the form 'd -> 'e (since x is being used as a function). Moreover, 'c = 'd since x is an argument in (x x).
Thus 'c = 'd -> 'e = 'c -> 'e.
But 'c = 'c -> 'e has no solution, so in ML this definition is invalid. We can’t pass a function to itself as an argument—the type system doesn’t allow it.

In Scheme this is allowed:
(define (f x) (x x))
but a call like
(f f)
certainly doesn’t do anything good!

Type Unions

Let’s try to type
fun f g = ((g 3), (g true));
Now the type of g is 'a -> 'b since g is used as a function.
The call (g 3) says 'a = int and the call (g true) says 'a = boolean.
Does this mean g is polymorphic?
That is, is the type of f
f : ('a -> 'b) -> 'b* 'b?
NO!
All functions have the type 'a -> 'b but not all functions can be passed to f.
Consider not : bool->bool.
The call (not 3) is certainly illegal.
What we’d like in this case is a union type. That is, we’d like to be able to type \( g \) as \((\text{int}|\text{bool})\rightarrow\text{b}\) which ML doesn’t allow.

Fortunately, ML does allow type constructors, which are just what we need.

Given

\[
\text{datatype } T = \\
\text{I of int} | \text{B of bool};
\]

we can redefine \( f \) as

\[
\text{fun } f \ g = \\
(g \ (\text{I}(3)), \ g \ (\text{B}(\text{true})));
\]

Finally, note that in a definition like

\[
\text{let } \\
\text{val } f = \\
\text{fn } x \Rightarrow x \ (* \ id \ function*) \\
in (f ~3, f \ \text{true}) \\
\text{end;}
\]

type inference works fine:

\[
\text{val it = (3, true) : int * bool}
\]

Here we define \( f \) in advance, so its type is known when calls to it are seen.