Better Abstraction Mechanisms

Ordinary (first-order, monomorphic) functions allow us to abstract operations away from specific values.

```
- fun sum nil = 0
    | sum (h::t) = h + (sum t)
val sum = fn : int list -> int
- sum [1,2,3];
val it = 6 : int
- sum [4,5];
val it = 9 : int

- fun concat nil = ""
    | concat (h::t) = h ^ (concat t);
val concat = fn : string list -> string
- concat ["a","bc","def"];
val it = "abcdef" : string
```

These two functions are fundamentally similar. Can we take advantage of this by making them instances of a single, more abstract, function?

Functional languages permit us to build more powerful abstractions in several ways.

• **Polymorphism** allows us to write functions that work uniformly on **container** data structures without regard for the contained type.

• **Higher-order functions** allow us to define functions that share a uniform control structure as instances of a single function.

Type Inference

ML automatically **infers** the most general (a.k.a. **principal** type) of every variable and expression in the program. Of course, a given expression might have **lots** of legal types, but these are all **instances** of the principal type.

Example: These are all instances of `'a * 'b -> 'a`

```
'a * 'a -> 'a
'a * int -> 'a
real * 'c -> real
bool * int -> bool
(real -> int * bool) * bool -> (real -> int * bool)
```

Initially, every type is free, i.e., an independent type variable. The inference process consists of applying **constraints** dictated by the program, which pins down the types just as much as required for the program to make sense.

Polymorphism

Higher-order abstractions must be written explicitly, but polymorphism comes "for free."

Consider these functions:

```
- fun length nil = 0
    | length (_::t) = 1 + length t;
val length = fn : 'a list -> int

- fun reverse nil = nil
    | reverse (h::t) = (reverse t) @ [h]
val reverse = fn : 'a list -> 'a list

- fun swap nil = nil
    | swap ((a,b)::t) = (b,a)::(swap t);
val swap = fn : ('a * 'b) list -> ('b * 'a) list
```

The first two make perfectly good sense on lists containing **any** type. The last makes sense on lists containing **pairs** of **any** type. Moreover, the function "does the same thing" to its argument list regardless of of the underlying types.

Note that SML automatically gives them types in which the "don't care" components are denoted by **type variables** `'a,'b,`..., pronounced "alpha, beta, ...".

Unless we explicitly add type constraints, SML will always give every function the most general possible type, allowing the broadest possible re-use.

Constraints

• **Constants** have a precise monotype

`3.0:real,4:int,true:bool,"abc":string`

• Certain built-in **operators** (but not all!) have a precise monotype.

`(op div):int * int->int, chr:  int -> string`

• **Tuple** constructor applications `()` always produce a tuple type, but don't dictate the types in the tuple.

• **List** constructor applications `nil, ::, []` always result in a list type, and force the elements of the list to be the **same** type, but don't dictate what that type is.

• Each arm of a `case` expression must evaluate to the same type.

• For `let`-bound functions, types of formal parameters are constrained (only) by the way in which parameters are used in the function body. Types of actual arguments to applications of these functions must be instances of the formal argument types.

• For functions passed as arguments, things are a little more restrictive; more details later.

Examples for Type inference

```
fun f(a,b,0) = a
  | f(a,b,n) = if b then a else a / f(a,b,n-1)

fun g(a,h,0) = a
  | g(a,h,n) = h(a,g(a,h,n-1))

fun foo(a,b,c,d) =
    if a = b then
      c+1
    else
      if a > b then
        c
      else
        b+d
```

Equality

The equality operators (`=,<>`) are even more special. They are **almost** polymorphic: they define **structural** (i.e., component-wise) equality on almost any type **except** on function types. So we can write a function like:

```
- fun tripeq (a,b,c) = a = b andalso b = c;
val tripeq = fn : ''a * ''a * ''a -> bool
- tripeq(1,1,2);
val it = false : bool
- tripeq (3.0,3.0,3.0);
val it = true : bool
```

But we mustn't apply it like this:

```
- let fun f x = x + 1 in tripeq(f,f,f) end;
std_in:0.0-0.0 Error: operator and operand
                don't agree (equality type required)
  operator domain: ''Z * ''Z * ''Z
  operand:  (int -> int) * (int -> int) * (int -> int)
  in expression:
    tripeq (f,f,f)
```

The special **equality type** variables `''a,''b,...` can only be instantiated to types that don't contain an arrow.

Overloading

The arithmetic operators `+,-,*,` apply to both `real` and `int` types, and the relational operators `<,<=,>=,>` apply to `real`, `int`, and `string` types. But these operators are **not** polymorphic, because they do something **different** to their arguments depending on type. Instead, we say that they are **overloaded**.

When these operators are used in a program, SML insists that it be possible to figure out the precise base type to which they are being applied. Often this can be determined automatically (e.g., `sum`). But sometimes it is necessary to give an explicit type constraint to clarify which version of the operator is needed.e.g.:

```
- fun pairadd(x,y,z) = x + y;
std_in:0.0 Error: overloaded variable cannot
                                be resolved: +
```

There is no provision for adding user-defined overloaded operators in SML.