Polymorphic type inference

Type inference for a construct proceeds as follows:

- assign each declared variable a fresh type variable
  - result of function is implicit variable
  - share argument and result type variables across function cases
  - references to let-bound polymorphic identifiers get distinct type variables
- each subexpression in construct places constraints on types of its operands
- solve constraints

Constraint solving can refine types:

- e.g. `'a` ⇒ `'b list`

For overloading, constraint solving must resolve overloading to particular instance

For = operation, refine `'a` to `''a` type

Overconstrained (unsatisfiable constraints) ⇒ type error

Underconstrained ⇒ OK; polymorphic bindings allowed

Properties of ML's type system

Hindley-Milner type system/inference

- allows code to depend on exact type of value, or have no dependencies on value (universal polymorphism)
- no constrained polymorphism, other than equality types...
- only let-bound polymorphism

Type inference yields **principal type** for expression

- i.e. unicity of typing

Complexity of type inference: DEXPTIME !

Concrete user-defined datatypes

Users can define their own (polymorphic) data structures

Simple example: ML's version of enumerated types

```
datatype sign = Positive | Zero | Negative;
```

Introduces constants

- can be used in patterns

```
fun signum(x) =
  if x > 0 then Positive
  else if x = 0 then Zero
  else Negative;

fun signum_value(Positive) = 1
|   signum_value(Zero)     = 0
|   signum_value(Negative) = ~1;
```


Variant records/tagged unions in ML

Each component of a datatype declaration can have info

- constructors act as functions to create values with that tag
- can be used in patterns to take apart values of a tag

Example:

```
datatype Sexpr =
  Nil |
  Integer of int |
  Symbol of string |
  Cons of (Sexpr * Sexpr);

(* '(3 hi) *)
val lst =
  Cons(Integer(3), Cons(Symbol("hi"), Nil);

fun first(Nil) = Nil
|   first(Cons(hd,_)) = hd;
fun rest (Nil) = Nil
|   rest (Cons(_,tl)) = tl;
```

Supports bounded, planned heterogeneity in ML

No abstraction or encapsulation, though

Recursive user-defined data types

Example: binary search trees of integers

```
datatype int_tree =
  Empty |
  Node of int * int_tree * int_tree;

(* insert: int_tree * int -> int_tree *)
fun insert(Empty, x) = Node(x,Empty,Empty)
|   insert(n as Node(y,t1,t2), x) =
    if x = y then n
    else if x < y then
      Node(y, insert(t1,x), t2)
    else
      Node(y, t1, insert(t2,x));

(* lookup: int_tree * int -> bool *)
fun lookup(Empty,x) = false
|   lookup(Node(y,t1,t2), x)
      if x = y then true
      else if x < y then
        lookup(t1, x)
      else
        lookup(t2, x);
```

Cannot be polymorphic, though

Polymorphic user-defined data types

Polymorphic tables:

```
datatype (''Key,'Value) table =
  Table of (''Key * 'Value) list;

exception NotFound;
fun lookup(Table(assocs),key) =
  let fun lookup_help(nil,key) =
            raise NotFound;
      |   lookup_help((k,v)::assocs,key)=
            if key = k then v
            else lookup_help(assocs,key)
  in lookup_help(assocs,key) end;

fun store(Table(assocs),key,value) =
  Table((key,value)::assocs);
```


Streams in ML

Represent a stream cons cell as a pair of a head value and a function which will return the next in line

```
datatype 'a stream =
  stream of 'a * (unit -> 'a);

fun cons_stream(x,f) = stream(x,f);
fun hd_stream(stream(x,f)) = x;
fun tl_stream(stream(x,f)) = f();

val ints_from(x) =
  cons_stream(x, fn() => ints_from(x+1));
val nats = ints_from(0);

fun map_stream(g,s) =
  cons_stream(
      g(hd_stream(x)),
      fn() => map_stream(g, tl_stream(s)));

val squares = map_stream(fn(x)=>x*x, nats);
```


References

Allow side-effects through explicit reference values (first-class locations):

- `ref: 'a → 'a ref`
- `!  : 'a ref → 'a`
- `(op :=) : 'a ref * 'a → unit`

```
- val v = ref 0;
val v = ref 0 : int ref
- v := !v + 1;
- !v;
1 : int
```

Arrays are indexed sequences of refs to things:

- `array: int * 'a → 'a array`
- `(op sub): 'a array * int → 'a`
- `update: 'a array * int * 'a → unit`
- `length: 'a array → int`
- `tabulate: int * (int → 'a) → 'a array`

Problems with polymorphic references

```
- fun I x = x;
val I = fn : 'a → 'a
- val fp = ref I
val fp = ref fn : ('a → 'a) ref
- (!fp true, !fp 5);
(true, 5) : bool * int
- fp := not;
```

hmmmm...

```
- !fp 5
```

CRASH!!!

Weakly polymorphic types

Restrict ref to disallow polymorphic instantiations:

- `ref: '1a → '1a ref`
- `array: int * '1a → '1a array`

`'1a` is a weak type variable:
it cannot be instantiated with any type
that contains a strong type variable (e.g. `'a` or `''a`)

```
ref I   (* illegal! *)
ref 3
ref (ref 3)
ref (I:bool→bool)
fun foo(r:'1a ref) = ref r; foo (ref 3)
```

Exceptions must have monomorphic types, for similar reasons