References

Allow side-effects through explicit reference values:

- `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;
val it = () : unit
- !v;
val it = 1 : int
```

Arrays: indexable mutable locations

Must say which things are mutable

Mutation is compartmentalized

References to polymorphic values?

```
- fun id(x) = x;
val ID = fn : 'a -> 'a
- val fp = ref id;
(* error in real SML; pretend it's not *)
val fp = ref fn : ('a -> 'a) ref
- (!fp true, !fp 5);
(true, 5) : bool * int
- fp := not;
hmmmm...
- !fp 5
CRASH!!!
```

Cannot allow references to polymorphic values

- exception arguments similarly cannot be polymorphic

In general, only polymorphic *literals* can be bound in `val`/`fun` bindings, not polymorphic *expressions*

- "non-generalizable type variable" error otherwise
- SML'90 had "weakly polymorphic types" instead

Polymorphic type inference

ML infers types of expressions automatically, as follows:

- assign each declared variable & subexpression a fresh type variable
  - result of function is another type variable
  - share arg. and result type variables across function cases
- for each subexpression, generate constraints on types of its operands
  - before applying a polymorphic function, replace quantified type variables with fresh ones for that application
- solve constraints

Constraint solving can partially refine types, e.g.:

- `'a` ⇒ `'b list`
- `'b` ⇒ `''c`

If overloaded operator is unresolved after constraint solving, default to `int` version

Overconstrained (unsatisfiable constraints) ⇒ type error

Underconstrained (still some type variables) ⇒ a polymorphic result

Let-bound polymorphism

ML type inference supports only let-bound polymorphism

- only `val`- or `fun`-declared names can be polymorphic, not names of formals
- implicit quantifiers of polymorphic variables are at outer level ("prenex form")

```
- fun id(x) = x;
val id = fn : 'a -> 'a
(* with explicit quantifier: val id = fn : ∀'a.'a->'a *)
- fun g(f) = (f 3, f "hi");
(* type error in ML; f cannot be given a polymorphic type *)
(* this (legal) ML type wouldn't allow the two different f calls:
   val g = fn : ∀'a.(('a->'a) -> int*string) *)
```

What if ML allowed explicitly quantified polymorphic types for formals?

```
- fun g(f:∀'a.'a->'a) = (f 3, f "hi");
val g = fn : (∀'a.'a->'a) -> int*string
- g(id);
val it = (3, "hi") : int * string
```

Type inference precludes first-class polymorphic values

Polymorphic vs. monomorphic recursion

When analyzing the body of a polymorphic function,
what do we do when we encounter a recursive call?

```
fun f(x) =
  ... f(hd(x)) ... f(tl(x)) ...
```

If allow polymorphic recursion,
then `f` is considered polymorphic in body,
and each recursive call uses a fresh instantiation
(like any call to a polymorphic function)

If only monomorphic recursion,
then force recursive call to pass same argument types as formals (don't make a fresh instantiation)

Type inference under polymorphic recursion is undecidable
(but only in obscure cases)

ML uses monomorphic recursion

Nested polymorphic functions

After doing type inference for a function, if any type variables remain in its type, then make the function polymorphic over them

But what about a nested function?

```
fun f(x) =
  let
    fun g(u, v) = ([x,u], [v,v])
  in
    ... g(x, 5) ... (* does this work? *)
    ... g([x], true) ... (* does this? *)
  end
```

Type of `f`: `'a -> '`...

Type of `g`: `'a * 'b -> 'a list * 'b list`

- but `'a` and `'b` are different...

`'a` is a non-generalizable type variable

- don't replace with a fresh type variable when `g` called

Handles monomorphic recursion restriction, too

Properties of ML type inference

Hindley-Milner type inference

- allows let-bound polymorphism only
- universal parametric polymorphism,
  no constrained polymorphism (other than equality types)

Type inference yields **principal type** for expression

- single most general type that can be inferred

Worst-case complexity of type inference: exponential time

Average case complexity: linear time

Modules for name-space management

A file full of types and functions can be cumbersome to manage

Would like some hierarchical organization to names

Modules allow grouping declarations to achieve
a hierarchical name-space

`structure` declarations in ML create modules

```
- structure Assoc_List = struct
=   type (''k,'v) assoc_list = (''k*'v) list
=   val empty = nil
=   fun store(alist, key, value) = ...
=   fun fetch(alist, key) = ...
= end;
structure Assoc_List : sig
  type ('a,'b) assoc_list = ('a*'b) list
  val empty : 'a list
  val store : (''a*'b) list * ''a * 'b ->
              (''a*'b) list
  val fetch : (''a*'b) list * ''a -> 'b
end
```

Using structures

To access declarations in a structure, use dot notation

```
- val league = Assoc_List.empty;
val l = [] : 'a list

- val league =
=   Assoc_List.store(league, "Mariners", {..});
val league = [("Mariners", {..})]
           : (string*{..}) list

- ...

- Assoc_List.fetch("Mariners");
val it = {wins=78,losses=4} : {..}
```

Other definitions of `empty`, `store`, `fetch`, etc. don't clash

Common names can be reused by different structures

The `open` declaration

To avoid typing a lot of structure names, can use the **open** *struct_name* declaration to introduce local synonyms for all the declarations in a structure (usually in a `let`, `local`, or within some other `struct`)

```
fun create_league(names) =
  let
    open Assoc_List
    val init = {wins=0,losses=0}
  in
    reduce(fn(name,league)=>
               store(league,name,init),
           empty, names)
  end
```


Modules for encapsulation

Want to hide details of data structure implementations from clients, i.e., **data abstraction**

- simplify interface to clients
- allow implementation to change without affecting clients

In C++ and Java, use `public`/`private` annotations

In ML:

- define a `signature` that specifies the desired interface
- specify the signature with the `structure` declaration

E.g. a signature that hides the implementation of `assoc_list`:

```
- signature ASSOC_LIST = sig
=   type (''a,'b) T
=   val empty : (''a,'b) T
=   val store : (''a,'b) T * ''a * 'b ->
=               (''a,'b) T
=   val fetch : (''a,'b) T * ''a -> 'b
= end;
signature ASSOC_LIST = sig ... end
```


Specifying the signatures of structures

Specify desired signature of structure when declaring it:

```
- structure Assoc_List :> ASSOC_LIST = struct
=   type (''k,'v) T = (''k*'v) list
=   val empty = nil
=   fun store(alist, key, value) = ...
=   fun fetch(alist, key) = ...
=   fun helper(...) = ...
= end;
structure Assoc_List : ASSOC_LIST
```

The structure's interface is the given one, not the default interface that exposes everything

Hidden implementation

Now clients can't see implementation, nor guess it

```
- val teams = Assoc_List.empty;
val teams = - : (''a,'b) Assoc_List.T

- val teams' = "Mariners"::"Yankees"::teams;
Error: operator and operand don't agree
  operator: string * string list
  operand:  string * (''Z,'Y) Assoc_List.T

- Assoc_List.helper(...);
Error: unbound variable helper in path
  Assoc_List.helper

- type Records = (string,...) Assoc_List.T;
type Records = (string,...) Assoc_List.T
- fun sortStandings(nil:Records):Records = nil
=   | sortStandings(pivot::rest) = ...;
Error: pattern and constraint don't agree
  pattern:    'Z list
  constraint: Records
in pattern: nil : Records
```

How to write `sortStandings`, if implementation is hidden?

Including fold etc. in external interfaces

To provide a complete interface if representation is hidden, often need to include ways of traversing the data structure

`fold` or its equivalent is often needed, as the most general pattern of iteration or recursion

E.g.:

```
- signature ASSOC_LIST = sig
=   ...
=   val fold: ((''a * 'b) * 'c -> 'c)
=             -> 'c -> (''a,'b) T -> 'c
= end
= structure Assoc_List :> ASSOC_LIST = struct
=   ...
=   fun fold f base alist = ...
= end;
...
- fun sortStandings(records) =
=   ... Assoc_List.fold ... records ...
...
```


Partially exposed representations

Can reveal parts of implementation of types, if desired, by specifying in signature

```
- signature ASSOC_LIST = sig
=   type (''k,'v) T = (''k*'v) list
=   ...
= end;
```

An alternative to providing `fold` etc.

But hinders future evolution of the implementation

Functors

Can parameterize structures by other structures

```
functor AListUser(AL:ASSOC_LIST) = struct
  ... AL.store ... AL.fetch ...
end
```

- only know aspects of `AL` that are defined by `ASSOC_LIST`

Instantiate functors to build regular structures:

```
- structure ALU1 = AListUser(Assoc_List);
- structure ALU2 = AListUser(Hash_Assoc_List);
```

Functors for bounded quantification

Define a signature representing the operations needed

```
signature ORDERED = sig
  type T
  val eq: T * T -> bool
  val lt: T * T -> bool
end
```

Define quantified algorithms as elements of functors parameterized by required signature

```
functor Sort(O:ORDERED) = struct
  fun min(x,y) =
    if O.lt(x,y) then x else y
  fun sort(lst) =
    ... O.lt(x, y) ...
end
```


An instantiation of Sort

Create specialized sorter by instantiating functor with appropriate operations

```
- structure IntOrder:ORDERED = struct
=   type T = int;
=   val lt = (op <);
=   val eq = (op =);
= end;
...
- structure IntSort = Sort(IntOrder);
...
- IntSort.sort([3,5,~2,...]);
...
```

Use `IntOrder:ORDERED`, not `IntOrder:>ORDERED`

- Using `:` instead of `:>` allows type binding (`T=int`) to bleed through to users of `IntOrder`
- `IntOrder` is a view/extension of an existing type, `int`; it isn't creating a new ADT w/ only 2 operations

Another instantiation of Sort

Can create nested, multiply parameterized functors:

```
functor PairOrder(
    structure First:ORDERED;
    structure Second:ORDERED):ORDERED =
struct
  type T = First.T * Second.T;
  fun lt((x1,x2),(y1,y2)) =
    First.lt(x1,y1) andalso Second.lt(x2,y2);
  fun eq((x1,x2),(y1,y2)) = ...;
end

structure IntStringSort = Sort(
    PairOrder(structure First = IntOrder;
              structure Second = StringOrder))
```


Signature "subtyping"

Signature specifies a particular interface

Any structure that satisfies that interface can be used where that interface is expected

- e.g. in functor application

Structure can have

- more operations
- more polymorphic operations
- more details of implementation of types

than required by signature

Some limitations of ML modules

Structures are not first-class values

- must be named or be argument to functor application
- must be declared at top-level or
  nested inside another structure or signature

Cannot instantiate functors at run-time to create “objects”
⇒ cannot simulate classes and object-oriented programming

No type inference for functor arguments

These constraints are (in part) to enable type inference of core

Modules vs. classes

Classes (abstract data types) implicitly define a **single** type,
with associated constructors, observers, and mutators

Modules can define 0, 1, or many types in same module,
with associated operations over several types

- no new types if adding operations to existing type(s)
  - e.g. a library of integer or array functions
  - hard to do in C++
- multiple types can share private data & operations
  - requires `friend` declarations in C++
- one new type requires a name for the type (e.g. `T`)
  - class name is also type name in C++, conveniently

Functors similar to parameterized classes

C++’s public/private is simpler than ML’s separate signatures,
but C++ doesn’t have a simple way of describing just an
interface