Polymorphic type inference

ML infers types of functions (etc.) automatically, as follows:

1. Assign each bound variable & subexpression
   a fresh type variable
   • plus fresh type variables for function’s argument & result types

2. For each subexpression, generate constraints on types
   of its operands and/or result
   • constraints of the form typeExpr₁ = typeExpr₂, e.g.
     'a == int or (string * 'b) == ('c * 'd list)
   • constrain each function case’s argument pattern to be equal to
     function’s argument type variable
   • constrain each function case’s body expression to be equal to
     function’s result type variable
   • before using a polymorphic identifier, replace quantified type
     variables with fresh ones for that occurrence

3. Solve constraints
   • if overloaded operator is unresolved after constraint solving,
     default to int version
   • overconstrained (unsatisfiable constraints) ⇒ type error
   • underconstrained (still some unconstrained type variables) ⇒
     a polymorphic result

Example

fun sum lst =
  if null lst then 0
  else hd lst +
      sum (tl lst)

Another example

fun map f nil = nil |
map f (x::xs) =
  f x ::
map f xs

Unification

Key operation during type inference: constraint solving
• all constraints are equalities between type expression trees
• yield further (simpler) constraints
  on any embedded type variables in either tree

Unification is key subroutine that
• checks whether structures of two trees are compatible
• yields equality constraints on embedded type variables

After a type variable is constrained to be equal to some other
type expression, then (conceptually) replace that variable
with the type expression in all later constraint solving
• special case: one type variable same as another
• sophisticated implementations use union-find data
  structures for fast merging of equivalent type variables

But what about 'a == (int * 'a list) ?
• occurs check: reject programs that try to constrain
  a type variable to be equal to
  a different type expression that contains that variable
Let-bound polymorphism

ML type inference supports only let-bound polymorphism
• only val- or fun-declared names can be polymorphic,
  not names of formals
• implies that all implicit quantifiers of polymorphic variables
  are at outer level (“prefix form”)
  - fun id(x) = x;
  val id = fn : 'a -> 'a
  (* with explicit quantifier: val id = fn : ∀'a. 'a->'a *)
  - fun g(ε) = (ε 3, ε "hi");
  (* type error in ML; ε cannot be given a polymorphic type *)
  (* this (legal) ML type wouldn’t allow the two different ε calls: *)
  val g = fn : ∀'a. ('a->'a) -> int*string
  - fun g (f : ∀'a. 'a->'a) = (g 3, f "hi")
  val g = fn : (∀'a. 'a->'a) -> int*string
  = g(id);
  val lit = (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(lst) =
  ... f(hd(lst)) ... f(tl(lst)) ...

If support polymorphic recursion,
then ε is considered polymorphic in its body,
and each recursive call uses a fresh instantiation
(like any call to a polymorphic function)

If support only monomorphic recursion,
then treat ε as having a non-polymorphic type in its body,
which forces recursive call to pass same argument types as formals

Type inference under polymorphic recursion is undecidable
(but only in obscure cases)
• and hard to implement since don’t know what type variables
  ε will have when recursive reference encountered
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 not equally flexible for callers...

'a inside ε is a non-generalizable type variable
• don’t replace with a fresh type variable when g called

Monomorphic recursion restriction implied as a special case

Properties of ML type inference

A.k.a. Hindley-Milner type inference
• allows let-bound polymorphism only
• universal unconstrained parametric polymorphism
• SML: hacks for overloading, 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
References

Allow side-effects through explicit reference values:

```plaintext
type 'a ref
val ref : 'a -> 'a ref
val ! : 'a ref -> 'a
val (op :=) : 'a ref * 'a ref -> unit

val v = ref 0;
val v = ref 0 : int ref
!

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

(ML also has arrays: efficiently indexable, mutable locations)

Language design principles:
* must say which things are mutable
* mutation is compartmentalized

References to polymorphic values?

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

hmmm...
- !fp 5
CRASH!!!
```

Cannot allow refs containing polymorphic values

In general,

val can bind to polymorphic values (e.g. fn..., []),
but not polymorphic expressions (e.g. ref...)
* type vars not generalized because of value restriction”
error otherwise
* SML'90 had “weakly polymorphic types” instead

Functors

Can parameterize structures by other structures

```plaintext
- signature MAP = 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;

- structure Assoc_List :> MAP = ...;
- structure Hash_Table :> MAP = ...;

- functor MapUser(M:MAP) = struct
  = ... M.T ... M.store ... M.fetch ...
  = end;
```

Instantiate functors to build regular structures:

```plaintext
- structure MU1 = MapUser(Assoc_List);
- structure MU2 = MapUser(Hash_Table);
```

Can typecheck MapUser separately from its instantiations
* unlike C++ templates,
  parameterized modules of most other languages

Functors for “bounded parametric polymorphism”

Want to write polymorphic code that’s still able to perform
operations like =, <, print, etc. on its data
* can use first-class functions for this (as we saw)
* can use functions for this (as we’ll now see)

Define a signature representing the operations needed

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

Define polymorphic algorithms as elements of functors
parameterized by required signature

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

Aside: use \texttt{IntOrder:ORDERED, not IntOrder:ORDERED}  
- Using : instead of :\texttt{>} allows type binding (T=int) to bleed through to users of \texttt{IntOrder}
- \texttt{IntOrder} is a view/extension of an existing type, \texttt{int}; it isn’t creating a new ADT w/ only 2 operations
- \texttt{transparent} (vs. \texttt{opaque}) signature ascription

Another instantiation of Sort

Can create nested, multiply parameterized functors:
\begin{verbatim}
functor PairOrder {  
    structure First:ORDERED;  
    structure Second:ORDERED:ORDERED = struct  
    type T = First.T * Second.T;  
    (* lexicographic comparison *)  
    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));
\end{verbatim}

- \texttt{IntStringSort.sort{  
  = [(3,"hi"),(3,"there"),(2,"bob")]);  
  val it = [(2,"bob"),(3,"hi"),(3,"there"）: ...}

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

Doesn’t have to be an exact match: 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 functor

Functors are not first-class values  
- must be named  
- must be declared at top-level

No type inference for functor arguments

Cannot use structures as data
Cannot instantiate functors at run-time to create “objects”  
- \texttt{\Rightarrow} cannot simulate classes and object-oriented programming just using structures and functors

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:
- a module defining 0 types is useful if adding operations to existing type(s)
- e.g. a library of integer or array functions
- cleaner than dummy class containing static fields & methods
- a module defining multiple types is useful if need to share private data & operations across types
- cleaner than friend declarations in C++

“Module + type” is more orthogonal, flexible than “class=type”
- perhaps less convenient for common case

Funcors 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

**Scheme**

Shares many features with ML:
- functional
  - functions are first-class values
  - largely side-effect free
- strongly typed
- expression-oriented, recursion-oriented
- garbage-collected heap
- highly regular and expressive

Unlike ML:
- dynamically typed, not statically typed
- lacks
  - pattern matching (but some Scheme extensions have this)
  - exceptions (but has continuations)
  - modules (but some Scheme extensions have this)
- syntax blends data and program
- good macro system

Lisp designed by McCarthy in late 50’s
Scheme dialect introduced by Steele and Sussman in mid 70’s as “executable lambda calculus”

**Syntax**

```
Program ::= { Definition | Expr }

Definition ::= (define id Expr)
          | (define (id_formal1 ... id_formalN) Expr)

Expr ::= id
        | Constant
        | SpecialForm
        | (Expr1 Exprarg1 ... ExprargN)

Constant ::= int | float | string | symbol
            | (lambda (id_formal1 ... id_formalN) Expr)
            | ...

SpecialForm ::= (if Exprtest Exprthen Exprelse)
              | ...
```

**Uniform prefix “calls”**

Examples:
- (+ 3 4) → 7
- (+ (* 3 8) (/ 8 2)) → 28
- (define seven (+ 3 4))
  seven → 7
- (+ seven 8) → 15
- (define (square n) (* n n))
  (square seven) → 49
- (define (fact n)
    (if (<= n 0)
      1
      (* n (fact (- n 1))))))
  (fact 20) → 2432902008176640000

Treating all operators & function calls in prefix syntax uniformly is simple, regular, and unambiguous, but not “traditional”
- don’t have to define precedence and associativity!
- can have 0, 1, 2, or many arguments to a “binary” operator
### Special forms

Regular call expressions evaluate all argument exprs (including function expr) then invoke function value passing argument values
- all user-defined procedures work this way

**Special forms** are special “functions” where arguments aren’t all treated as expressions to be evaluated first
- can define new special forms using macros

**Example:**
```
(define x 0)
(define y 5)
(if (= x 0) 0 (/ y x))  \rightarrow  0
(define (my-if test then else)
  (if test then else))
(my-if (= x 0) 0 (/ y x))  \rightarrow  \text{error!}
(define-syntax my-if
  (syntax-rules ()
    ((my-if test then else)
      (if test then else)))))
(define (my-if test then else)
  (if test then else))
(my-if (= x 0) 0 (/ y x))  \rightarrow  0
```

### Other special forms

**cond:** like if-elseif...else chain:
```
(cond ((> x 0) 1)
      ((= x 0) 0)
      (else -1))
```

**Short-circuiting and and or** (like ML’s andalso and orelse)
```
(or (= x 0) (> (/ y x) 5) ...)
```

**let:** “simultaneous” local variable bindings:
```
(define x 1)  (define y 2)  (define z 3)
(let ((x 5)
      (y (+ 3 4))
      (z (+ x y z)))
  (+ x y z))  \rightarrow  5+7+(1+2+3)=18
```

**let:** “sequential” local variable bindings (like ML’s let):
```
(let* ((x 5)
       (y (+ 3 4))
       (z (+ x y z)))
  (+ x y z))  \rightarrow  5+7+(5+7+3)=27
```

---

### Lists

**Translation between ML and Scheme**

<table>
<thead>
<tr>
<th>ML</th>
<th>Scheme</th>
</tr>
</thead>
<tbody>
<tr>
<td>nil</td>
<td>()</td>
</tr>
<tr>
<td>x :: xs</td>
<td>(cons x xs)</td>
</tr>
<tr>
<td>[x, y, z]</td>
<td>(list x y z)</td>
</tr>
<tr>
<td>hd(lst)</td>
<td>(car lst)</td>
</tr>
<tr>
<td>tl(lst)</td>
<td>(cdr lst)</td>
</tr>
<tr>
<td>null(lst)</td>
<td>(null? lst)</td>
</tr>
</tbody>
</table>

**Examples:**
```
(define lst (list 5 6 7 8))  \rightarrow  (5 6 7 8)
(define lst2 (cons 4 lst))  \rightarrow  (4 5 6 7 8)
(+ (car lst) (car lst2))  \rightarrow  9
(define lst3 (cdr lst))  \rightarrow  (6 7 8)
- lst, lst2, and lst3 have shared subpieces
```

---

### Dynamic typing

There are no static types, neither explicit nor inferred
Any variable, and any data structure, can hold any type of value
Values have (run-time) types, variables are typeless

Typechecking is performed only when absolutely necessary
**E.g.**
- car & cdr check that argument is a cons cell, and
- + checks that arguments are numbers, but
- cons and list check nothing!

Lists can be heterogeneous:
```
(list 3 4.5 () "hi" (list 3 5))
\rightarrow  (3 4.5 () "hi" (3 5))
- lists in Scheme subsume both tuples and lists in ML
```

**E.g.** an association list of key-value pairs:
```
(define Zips (list (list "Seattle" 98195)
                   (list "Boston" 02115)
                   (list "Reston" 22091)))
\rightarrow  (("Seattle" 98195)
              ("Boston" 02115)
              ("Reston" 22091))
```
Type testing

Programs can test the type of values at run-time

Some type-testing predicates:
- null?
- pair?
- symbol?
- boolean?
- number? integer? ...
- string?
- ...

Quoting

List literals via quote or ' special form:
- `(list 3 (list 4 5) 6) → (3 (4 5) 6)
- `(quote (3 (4 5) 6)) → (3 (4 5) 6)
- '(3 (4 5) 6) → (3 (4 5) 6)

Quoted identifiers are symbol constants:
- 'positive → positive
- (car '(if (> a b) 3 4)) → if

Programs and data share same regular syntax
Makes it very easy to write programs that build, take apart, and transform programs