Type inference for functions

Type declaration of function result can be omitted

- infer function result type from body expression result type

```
- fun max(x:int, y:int) =
=     if x >= y then x else y;
val max = fn : int * int -> int
```

Can even omit type declarations on arguments to functions

- infer all types based on how arguments are used in body
- fancy, constraint-based algorithm to do type inference

```
- fun max(x, y) =
=     if x >= y then x else y;
val max = fn : int * int -> int
```

Type Inference: A Big Idea

Functions with many possible types

Some functions could be used on arguments of different types

Some examples:

`null`: can test an `int list`, or a `string list`, or ....

- in general, work on a list of any type *T*:
  `null:` *T* `list -> bool`

`hd`: similarly works on a list of any type *T*, and returns an element of that type:
  `hd:` *T* `list ->` *T*

`swap`: takes a pair of an *A* and a *B*, returns a pair of a *B* and an *A*:
  `swap:` *A* `*` *B* `->` *B* `*` *A*

How to define such functions in a statically-typed language?

- in C: can't (or have to use casts)
- in C++: can use templates
- in ML: allow functions to have **polymorphic types**

Polymorphic types

A polymorphic type contains one or more **type variables**

- an identifier starting with a quote

E.g.

```
'a list
'a * 'b * 'a * 'c
{x:'a, y:'b} list * 'a -> 'b
```

A polymorphic type describes a set of possible (regular) types, where each type variable is replaced with some type

- each occurrence of a type variable must be replaced with the same type

Polymorphic Types: A Huge Idea

Polymorphic functions

Functions can have polymorphic types:

```
null   : 'a list -> bool
hd     : 'a list -> 'a
tl     : 'a list -> 'a list
(op ::): 'a * 'a list -> 'a list
swap   : 'a * 'b -> 'b * 'a
```

When calling a polymorphic function, need to find the **instantiation** of the polymorphic type into a regular type

- caller knows types of arguments
- can compute how to replace type variables so that the replaced function type matches the argument types
- derive type of result of call

E.g.

`hd([3,4,5])`

- actual argument type: `int list`
- polymorphic type of hd: `'a list -> 'a`
- replace `'a` with `int` to make a match
- instantiated type of hd for this call: `int list -> int`
- type of result of call: `int`

Polymorphic values

Regular values can polymorphic, too

```
nil: 'a list
```

Each reference to `nil` finds the right instantiation for that use, separately from other references

E.g.

```
(3 :: 4 :: nil) :: (5 :: nil) :: nil
```


Polymorphic type inference

ML infers types of expressions automatically, as follows:

- assign each declared variable a fresh type variable
  - result of function is implicit variable
  - each reference to a polymorphic function or value gets fresh type variables to describe that instantiation
- each subexpression in construct places constraints on types of its operands
- solve constraints

Overconstrained (unsatisfiable constraints) ⇒ type error

Underconstrained (still some type variables) ⇒ a polymorphic result

Some details:

- resolving overloaded operators like +, <
- resolving the special =, <> operators ("equality types")
- some restrictions on use of polymorphic results at top-level ("type vars not generalized because of value restriction")

Polymorphic Type Inference: A Big Idea

Recursive types

Lists are a **recursively defined** data type:

"A ***list*** is either
nil, or
a pair of a head value and a tail ***list***"

This definition has
a **base case** (which **is not** recursively defined) and
an **inductive case** (which **is** recursively defined)

All well-founded recursive definitions have
at least one base case (to be able to stop the recursion) and
at least one inductive case (so there's some recursion),
where the inductive cases refer to a smaller subcases

A value of a recursive type is made up of
one of the base cases
possibly extended with one or more recursive cases

"The list [1,2] is the pair of 1 and (the pair of 2 and (nil))"

Recursive Types: A Big Idea

Recursive functions

Recursive types are naturally manipulated with recursive functions

- operations on lists
- operations on trees
- some operations on numbers
- ...

Pattern:

- check if have base case #1
  if so, then compute appropriate result
- repeat for other base cases, if any
- then check for inductive case #1
  if so, then
  - compute results for subproblems
  - combine into result for overall problem
- repeat for other inductive cases, if any

Recursive functions apply "divide and conquer"

- divide big problem into some smaller subproblems
- solve them separately
- solve big problem using the subproblem solutions

Recursive functions on lists

Given pattern of list data type:

"A ***list*** is either
nil, or
a pair of a head value and a tail ***list***"

Have a standard pattern of recursive function over list data type:

```
fun f(..., lst, ...) =
  if null(lst) then
    (* base case *)
    ...
  else
    (* inductive case *)
    ... hd(lst) ... f(..., tl(lst), ...) ...
```


Recursive functions on integers

Given pattern of "natural number" data type:

"A ***natural number*** is either
0, or
1 + a ***natural number***"

Have a standard pattern of recursive function over natural nums:

```
fun f(..., n, ...) =
  if n=0 then
    (* base case *)
    ...
  else
    (* inductive case *)
    ... n ... f(..., n-1, ...) ...
```

(Could have several base cases)

Recursion vs. iteration

Recursion more general than iteration

- anything a loop can do, a recursive function can do
- some recursive functions require a loop + a stack

Recursion often considered less efficient (both time and space) than iteration

- procedure calls and stack allocation/deallocation on each "iteration"
- some "natural" inductive definitions less efficient than iterative definitions

Tail recursion

Tail recursion: recursive call is last operation before returning

- can be implemented just as efficiently as iteration, in both time and space, since tail-caller isn't needed after callee returns

Some tail-recursive functions:

```
fun last(lst) =
  let val tail = tl(lst) in
    if null(tail) then
      hd(lst)
    else
      last(tail)
  end

fun includes(lst, x) =
  if null(lst) then
      false
  else if hd(lst) = x then
      true
  else
      includes(tl(lst), x)
```

Some non-tail-recursive functions:
`length`, `square_all`, `append`, `fact`, `fib`, ...

Converting to tail-recursive form

Can often rewrite a recursive function into a tail-recursive one

- introduce a helper function
- the helper function has an extra accumulator argument
- the accumulator holds the partial result computed so far
- accumulator returned as full result when base case reached

This isn't tail-recursive:

```
fun fact(n) =
  if n <= 1 then
    1
  else
    n * fact(n-1)
```

This is:

```
fun fact_helper(n,res) =
  if n <= 1 then
    res
  else
    fact_helper(n-1, res*n)
fun fact(n) =
  fact_helper(n, 1)
```