Lecture #18: Type Inference and Unification (A Side Trip)

The language ML

- ML (for "Metalanguage") is a language dating back to the 1970's, originally developed to support another project (a prover for LCF: Language for Computable Functions).
- Descendants include Haskell and OCaml.
- Has some interesting features. Example:

```
fun map f [] = []
  |  map f (a :: y) = (f a) :: (map f y)
```

- This defines the function `map` by giving two patterns that show what it is supposed to produce on the empty list (`[]`) and on a list formed from a head (formal parameter `a`) and a tail (`tail`). The `::` operator is like Scheme's `cons` function: constructs a list by prepending a single element to a list.
- A function call `map(f, y)` is written '`map f y`'.

Typing In ML

```
fun map f [] = []
 |  map f (a :: y) = (f a) :: (map f y)
fun reduce f init [] = init
 |  reduce f init (a :: y) = reduce f (f init a) y
fun count [] = 0
 |  count (_ :: y) = 1 + count y
fun addt [] = 0
    addt ((a,_,c) :: y) = (a+c) :: addt y
```

- Despite lack of explicit types here, this language is statically typed!
- Compiler will reject the calls `map 3 [1, 2]` and `reduce (op +) [] [3, 4, 5]`.
- Does this by *deducing* types from their uses.

Type Inference

- In simple case:

```
fun add [] = 0
  |  add (a :: L) = a + add L
```

compiler deduces that `add` has type `int list` $\rightarrow$`int`.

- Uses facts that (a) 0 is an `int`, (b) `[]` and `a::L` are lists (`::` is cons), (c) + yields `int`.
- More interesting case:

```
fun count [] = 0
  |  count (_ :: y) = 1 + count y
```

(`_` means "don't care" or "wildcard"). In this case, compiler deduces that `count` has type α `list` $\rightarrow$`int`.

- Here, α is a *type parameter* (we say that `count` is it polymorphic).

Aside: Runtime Implementation of Polymorphism

- The last example works for any value of α:

```
fun count [] = 0
 |  count (_ :: y) = 1 + count y
```

- As is also the case here, where the type of `x` is known to be `bool`, but the types of `z` and `y` are unknown.

```
fun iffy x y z = if x then z else y;
```

- No special run-time testing is required to bring this about.
- In typical implementations, all types have the same representation at the machine-code level—they are words containing pointers (or possibly integers), for which assignment and parameter passing involve the same instructions regardless of contents.
- Hence, a single translation works for all types.

Doing Type Inference

- Given a definition such as

```
fun add [] = 0
 |  add (a :: L) = a + add L
```

- First give each named entity here an unbound type parameter as its type: `add:`α`, a:`β`, L:`γ.
- Now use the type rules of the language to give types to everything and to *relate* the types:
  - 0: `int`, `[]`: δ `list`.
  - Since `add` is function and applies to `int`, must be that $\alpha = \iota \rightarrow \kappa$, and $\iota = \delta$ `list`
  - etc.
- Gives us a large set of *type equations*, which can be solved to give types.
- Solving involves *pattern matching*, known formally as *unification*.

Type Expressions

- For this lecture, a type expression can be
  - A *primitive type* (`int`, `bool`);
  - A *type variable* (ML's notation: 'a, 'b, 'c$_1$, etc.);
  - The *type constructor* T `list`, where T is a type expression (like `List<T>` in Java);
  - A *function type* $D \rightarrow C$, where D and C are type expressions.
- Will formulate our problems as systems of *type equations* between pairs of type expressions.
- Need to find the substitution (the *unifier*) for the type variables that solves the system (simultaneously makes all the equations true).

Solving Simple Type Equations

- Simple example: solve

  $\texttt{'a list} = \texttt{int list}$

- Easy: $\texttt{'a} = \texttt{int}$.
- How about this:

  $\texttt{'a list} = \texttt{'b list list};\ \texttt{'b list} = \texttt{int list}$

- Also easy: $\texttt{'a} = \texttt{int list};\ \texttt{'b} = \texttt{int}$.
- On the other hand:

  $\texttt{'a list} = \texttt{'b} \rightarrow \texttt{'b}$

  is unsolvable: lists are not functions.
- Also, if we require *finite* solutions, then

  $\texttt{'a} = \texttt{'b list};\texttt{'b} = \texttt{'a list}$

  is unsolvable.

Most General Solutions

- Rather trickier:

  `'a list = 'b list list`

- Clearly, there are lots of solutions to this: e.g,

  `'a` $=$ `int list`; `'b` $=$ `int`

  `'a` $=$ `(int` $\to$ `int) list`; `'b` $=$ `int` $\to$ `int`

  etc.

- But prefer a *most general* solution that will be compatible with *any* possible solution.
- Any substitution for `'a` must be some kind of list, and `'b` must be the type of element in `'a`, but otherwise, no constraints
- Leads to solution

  `'a` $=$ `'b list`

  where `'b` remains a free type variable.

- In general, our solutions look like a bunch of equations `'a`$_i = T_i$, where the T_i are type expressions and none of the `'a`$_i$ appear in any of the T's.

Finding Most-General Solution by Unification

- To *unify* two type expressions is to find substitutions for all type variables that make the expressions identical.
- The set of substitutions is called a *unifier*.
- Represent substitutions by giving each type variable, `'`τ, a *binding* to some type expression.
- The algorithm that follows treats type expressions as objects (so two type expressions may have identical content and still be different objects). All type variables with the same name are represented by the same object.
- Initially, each type expression object is *unbound*.

Unification Algorithm

- For any type expression, T, and unifier u, define

$$u[T] = \begin{cases} u[T'], & \text{if } T \text{ is bound to type expression } T' \\ T, & \text{otherwise} \end{cases}$$

- Now proceed recursively:

```
unify (TA,TB,u):
   """Extends and returns unifier u so that it unifies TA and TB; else None."""
   TA = u[TA]; TB = u[TB]
   if TA.isFreeTypevar(u):    #  If TA is type variable not bound in u
      return u.bind(TA, TB)   #  Modify u so u(TA) = TB.
   if TB.isFreeTypeVar(u):
      return u.bind(TB, TA)
   if TA is $C$(TA$_1$,TA$_2$,...,TA$_n$) and TB is $C$(TB$_1$,...,TB$_n$):
                             #  C is a  type constructor, like C<T> in Java.
      for i in range($n$):
         u = unify(TA$_i$, TB$_i$, u)
         if u is None: return None
      return u
   return None
```

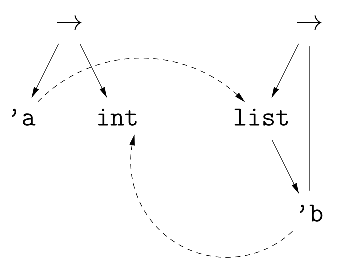
Example of Unification I

- Try to solve $A = B$, where

  A = `'a` $\to$`int`; B = `'b list`$\to$`'b`

  by computing `unify`(A, B).

Dashed arrows are bindings
Red items are current TA and TB



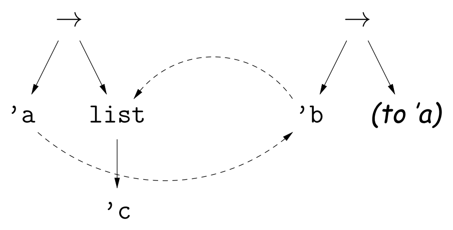
So `'a` $=$ `int list` and `'b` $=$ `int`.

Example of Unification II

- Try to solve $A = B$, where

  A = `'a` $\to$ `'c list`; B = `'b` $\to$ `'a`

  by computing `unify`(A, B).



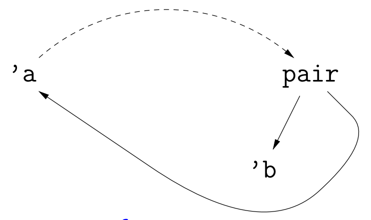
So `'a` $=$ `'b` $=$ `'c list` and `'c` is free.

Example of Unification III: Simple Recursive Type

- Introduce a new type constructor: `('h,'t) pair`, which is intended to model typed Lisp cons-cells (or nil). The `car` of such a pair has type `'h`, and the `cdr` has type `'t`.
- Try to solve $A = B$, where

  A = `'a`; B = (`'b`, `'a`) `pair`

  by computing `unify`(A, B).
- This one is very easy:



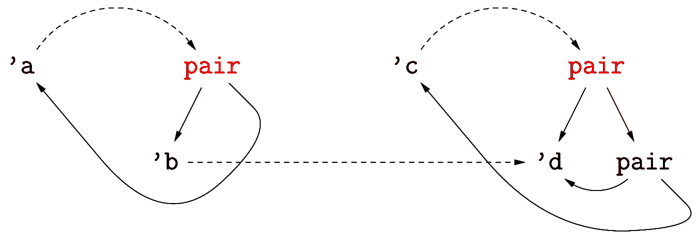
So `'a` = (`'b`, `'a`) `pair`; `'b` is free.

Example of Unification IV: Another Recursive Type

- This time, consider solving $A = B,\ C = D,\ A = C$, where

  A = `'a`; B = (`'b`, `'a`) `pair`; C = `'c`; D = (`'d`, (`'d`, `'c`) `pair`) `pair`.

  We just did $A = B$, and $C = D$ is almost the same, so we'll just skip those steps, and work on $A = C$.



Now we're in trouble: infinite recursion.

Circular Types

- It is possible to extend the unification algorithm to handle Example IV and others like it.
- However, most languages opt for a different approach.
- For example, in ML, we can define a linked-list type as):

```
datatype 'a llist = Nil | Cons of ('a * 'a llist)
```

  Here, `Nil` and `Cons(head, tail)` are *constructors* of values of type `llist`. `'a` is a formal generic type parameter (like <T> in Java).
- Now all the recursing happens explicitly in the definition of `llist`, and we don't need to produce types such those in the last two examples.
- Instead, we institute the *occurs check*: We do not allow binding a type variable to a type expression that contains that type variable.

Example of Unification V

- Try to solve

  ’b list = ’a list; ’a→’b = ’c;
  ’c →bool = (bool →bool) →bool

- We unify both sides of each equation (in any order), updating the unifier as we go.

```
       Unifier              Unifications
                      Unify ’b  list, ’a  list:
 ’a:  bool              Unify ’b, ’a
                      Unify ’a→’b, ’c
 ’b:  ’a              Unify ’c → bool, (bool → bool) → bool
     bool               Unify ’c, bool → bool:
                          Unify ’a → ’b, bool → bool:
 ’c:  ’a → ’b               Unify ’a, bool
     bool → bool            Unify ’b, bool:
                              Unify bool, bool
                        Unify bool, bool
```

Some ML Type Rules

Construct	Type	Conditions
Integer literal	`int`	
[]	`'a list`	
hd (L)	’a	L: ’a list
tl (L)	’a list	L: ’a list
E_1+E_2	int	E_1: int, E_2: int
E_1::E_2	’a list	E_1: ’a, E_2: ’a list
$E_1 = E_2$	bool	E_1: ’a, E_2: ’a
E_1!=E_2	bool	E_1: ’a, E_2: ’a
if E_1 then E_2 else E_3 fi	’a	E_1: bool, E_2: ’a, E_3: ’a
E_1 E_2	’b	E_1: ’a →’b, E_2: ’a
`def f x1 ...xn = E`		x1: ’a$_1$, . . . , xn: ’a$_n$ E:’a$_0$, f: ’a$_1$ →. . .→’a$_n$ →’a$_0$.

Using the Type Rules

- Interpret the notation $E : T$, where E is an expression and T is a type, as

  type(E) = T

- Seed the process by introducing a set of fresh type variables to describe the types of all the variables used in the program you are attempting to process. For example, given

  ```
  def f x = x
  ```

  we might start by saying that

  ```
  type(f) = ’a0, type(x) = ’a1
  ```

- Apply the type rules to your program to get a bunch of Conditions.
- Whenever two Conditions ascribe a type to the same expression, equate those types.
- Solve the resulting equations.

Aside: Currying

- Writing

  ```
  def sqr x = x*x;
  ```

  means essentially that `sqr` is defined to have the value λ `x. x*x`.

- To get more than one argument, write

  ```
  def f x y = x + y;
  ```

  and `f` will have the value λ `x.` λ `y. x+y`

- Its type will be `int →int →int` (Note: →is right associative).
- So, `f 2 3 = (f 2) 3 = (`λ `y. 2 + y) (3) = 5`
- This trick of turning multi-argument functions into one-argument functions is called *currying* (after Haskell Curry), although it was first used by Moses Schönfinkel. (And, yes, you saw it in CS61A!)

Example

```
if p L then init else f init (hd L) fi + 3
```

- Let's initially use 'p, 'L, etc. as the fresh type variables giving the types of identifiers.
- Using the rules then generates equations like this:

```
'p = 'a0→'a1, 'L = 'a0, type(p L) = 'a1 # call rule
'L = 'a2 list, type(hd L) = 'a2          # hd rule
'f = 'a3→'a4, 'init = 'a3, type(f init) = 'a4
                                         # call rule
'a4 = 'a5→'a6, 'a2 = 'a5, type(f init (hd L)) = 'a6
                                         # call rule
'a1 = bool, 'init = 'a7, 'a6 = 'a7, type(if...  fi) = 'a7
                                         # if rule
'a7 = int, int = int, type(if...  fi+3) = int   # + rule
```

etc.

Example, contd.

Solve all these equations by sequentially unifying the two sides of each equation, in any order, keeping the bindings as you go.

```
'p = 'a0→'a1, 'L = 'a0
'L = 'a2 list
    'a0 = 'a2 list
'f = 'a3→'a4, 'init = 'a3
'a4 = 'a5→'a6, 'a2 = 'a5
'a1 = bool, 'init = 'a7, 'a6 = 'a7
    'a3 = 'a7
'a7 = int, int = int
```

So (eventually),

```
'p = 'a5 list→bool, 'L = 'a5 list, 'init = int,
'f = int →'a5→int
```

Introducing Fresh Variables

- The type rules for the simple language we've been using generally call for introducing fresh type variables for each application of the rule.
- Example: in the expression

  ```
  if x = [] then [] else x::y fi
  ```

  the two [] are treated as having two different types, say `'a0 list` and `'a1 list`, which is a good thing, because otherwise, this expression cannot be made to type-check [why?].
- You'd probably want to do the same with count:

  ```
  fun count [] = 0
   |  count (_ :: y) = 1 + count y
  ```

  Analyzing this gives a type of `'a list→int`. Suppose we have two calls later in the program: `count (0::x)` and `count ([1]::y)`.
- Obviously, we also want to replace `'a` in each case with a fresh type variable, since otherwise, `count` would be specialized to work only on lists of integers or only on lists of lists.

. . . Or not?

- But we *don't* want to introduce a fresh type variable for each call when inferring the type of a function from its definition:

  ```
  fun switcher x y z = if x=0 then y else switcher(x-1,z, y) fi
  ```

- Here, we want the type of `switcher` to come out to be `int→'y→'y→'y`, but that can't happen if the recursive call to `switcher` can take argument types that are independent of those of `y` and `z`.
- Same problem with a set of mutually recursive definitions.
- So must always specify which groups of definitions get resolved together, and when calling a function is supposed to create a fresh set of type variables instead.