Lecture #23: Type Inference and Unification

Typing In the Language ML

- Examples from the language 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 *polymorphic*).

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: $add : \alpha$, $a : \beta$, $L : \gamma$.
- 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 *type unification*.

Type Expressions

- For this lecture, a type expression can be
  - A *primitive type* (`int`, `bool`);
  - A *type variable* (today we'll use ML notation: 'a, 'b, 'c$_1$, etc.);
  - The *type constructor* T `list`, where T is a type expression;
  - 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

Solving Simple Type Equations

- Simple example: solve

  'a list = int list
- Easy: 'a = int.
- How about this:

  'a list = 'b list list; 'b list = int list
- Also easy: 'a = int list; 'b = int.
- On the other hand:

  'a list = 'b → 'b

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

  'a = 'b list; 'b = 'a list

  is unsolvable. However, our algorithm will allow infinite solutions.

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 → int) list; 'b = int → 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.
- It generalizes binding by allowing *all* type expressions (not just type variables) to be bound to other type expressions
- Initially, each type expression object is *unbound*.

Unification Algorithm

- For any type expression, define

$$\text{binding}(T) = \begin{cases} \text{binding}(T'), & \text{if } T \text{ is bound to type expression } T' \\ T, & \text{otherwise} \end{cases}$$

- Now proceed recursively:

```
unify (TA,TB):
  TA = binding(TA); TB = binding(TB);
  if TA is TB: return True;  #  True if TA and TB are the same object
  if TA is a type variable:
    bind TA to TB; return True
  bind TB to TA;  #  Prevents infinite recursion
  if TB is a type variable:
    return True
  #  Now check that binding TB to TA was really OK.
  if TA is C(TA_1,TA_2,...,TA_n) and TB is C(TB_1,...,TB_n):
     return unify(TA_1,TB_1) and unify(TA_2,TB_2) and ...
     # where C is some type constructor
  else:  return False
```

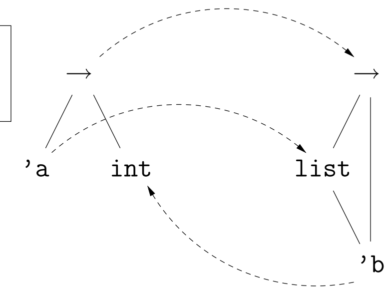

Example of Unification I

- Try to solve $A = B$, where

  A = 'a $\rightarrow$ int; B = 'b list$\rightarrow$ 'b

  by computing unify(A, B).

Dashed arrows are bindings
Red items are current TA and TB



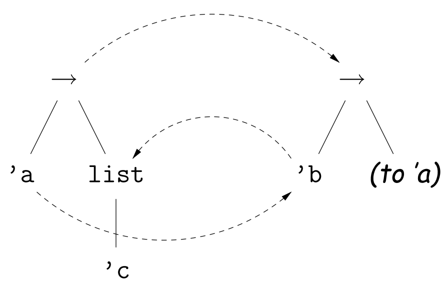
So 'a = int list and 'b = int.

Example of Unification II

- Try to solve $A = B$, where

  A = 'a $\rightarrow$ 'c list; B = 'b $\rightarrow$ 'a

  by computing unify(A, B).



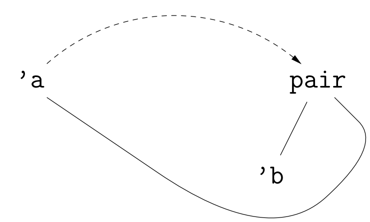
So 'a = 'b = 'c list and 'c is free.

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:



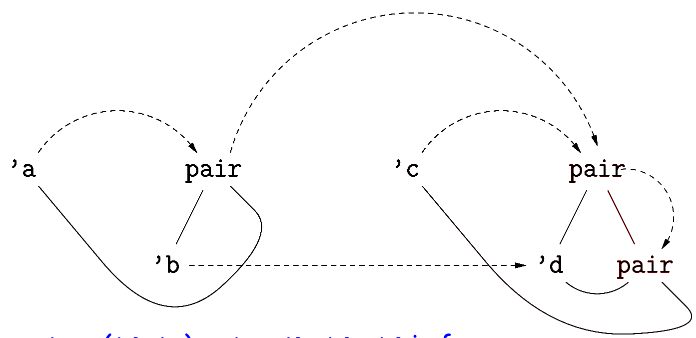
So 'a = ('b, 'a) pair; 'b is free.

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 the first one, and the second is almost the same, so we'll just skip those steps.



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

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), keeping the bindings from one unification to the next.

```
'a:  bool             Unify 'b  list, 'a  list:
                        Unify 'b, 'a
'b:  'a               Unify 'a→ 'b, 'c
     bool             Unify 'c →  bool, (bool →  bool) →  bool
                        Unify 'c, bool →  bool:
'c:  'a →  'b             Unify 'a →  'b, bool →  bool:
     bool  →  bool          Unify 'a, bool
                            Unify 'b, bool:
                              Unify bool, bool
                        Unify bool, bool
```


Some Type Rules (reprise)

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: $'a_1, \ldots,$ xn: $'a_n$ E:$'a_0$, f: $'a_1 \to \ldots \to 'a_n \to 'a_0$.

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
- Zounds! It's the CS61A substitution model!
- This trick of turning multi-argument functions into one-argument functions is called *currying* (after Haskell Curry).

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 our language must always state which groups of definitions get resolved together, and when calling a function is supposed to create a fresh set of type variables instead.