Lecture #13: Type Inference and Unification

Administrivia.

- Deadline on project #1 was pushed 24 hours to compensate for SVN glitch.
- Be sure to check your instructional mail accounts!
- Project #2 spec and files released.
- Review session Sunday at 1700 (place TBA).
- Reader containing notes and lecture slides will be at Vick Copy (corner Euclid and Hearst) this weekend.

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.

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.
- Initially, each variable is *unbound*.

Unification Algorithm

- For any type expression, define

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

- Now proceed recursively:

```
unify (T1,T2):
  T1 = binding(T1); T2 = binding(T2);
  if T1 = T2:  return true;
  if T1 is a type variable and does not appear in T2:
    bind T1 to T2; return true
  if T2 is a type variable and does not appear in T1:
    bind T2 to T1; return true
  if T1 and T2 are S1 list and S2 list:  return unify (S1,S2)
  if T1 and T2 are D1→  C1 and D2→  C2:
     return unify(D1,D2) and unify(C1,C2)
  else:  return false
```


Example of Unification

- 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$_1$, ..., xn: 'a$_n$ E:'a$_0$, f: 'a$_1$ → ... → 'a$_n$ → 'a$_0$.

Using the Type Rules

- Apply these rules to a 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`

- It's type will be `int` $\rightarrow$ `int` $\rightarrow$ `int` (Note: $\rightarrow$ 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

```
def f x L = if L = [] then [] else
              if x != hd(L) then f x (tl L)
                            else x ::  f x (tl L) fi
            fi
```

- Let's initially use `'f`, `'x`, `'L`, etc. as the fresh type variables.
- Using the rules then generates equations like this:

```
'f = 'a0 →  'a1 →  'a2      # def rule
'L = 'a3 list               # = rule,  [] rule
'L = 'a4 list               # hd rule,
'x = 'a4                    # != rule
'x = 'a0                    # call rule
'L = 'a5 list               # tl rule
'a1 = 'a5 list              # tl rule, call rule
...
```