Lecture #22: Type Inference, Continued

- The issue from last time: given two type expressions defining sets of possible types, do they intersect?
- A type expression is a *pattern*. 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

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

- 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*,

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

  $\texttt{'a} = (\texttt{int} \rightarrow \texttt{int})\ \texttt{list}; \quad \texttt{'b} = \texttt{int} \rightarrow \texttt{int}$

  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

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

  where `'b` remains a free type variable.

- In general, our solutions look like a bunch of equations $\texttt{'a}_i = T_i$, where the T_i are type expressions and none of the $\texttt{'a}_i$ 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, $'\tau$, 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:`

`'b:`

`'c:`

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:                    Unify 'b  list, 'a  list:

'b:

'c:
```

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:                    Unify 'b  list, 'a  list:
                         Unify 'b, 'a
'b:  'a


'c:
```

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:                    Unify 'b  list, 'a  list:
                         Unify 'b, 'a
'b:  'a                Unify 'a→ 'b, 'c


'c:  'a →  'b
```

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

'c:  'a →  'b
```

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

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

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

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

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

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
```

Expressing Type Rules of a Small Language

- In rules, read $E : \tau$ as "Expression E must have type τ."
- Each of the `'a`, `'a`$_i$ mentioned is a "fresh" type variable, introduced for each application of the rule.

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	'a	E_1: bool, E_2: 'a, E_3: 'a
E_1 E_2	'b	E_1: 'a $\rightarrow$ 'b, E_2: 'a
`def f x1 ...xn = E`		x1: 'a$_1$, ..., xn: 'a$_n$, E:'a$_0$, f: 'a$_1$ $\rightarrow$ ... $\rightarrow$ 'a$_n$ $\rightarrow$ '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
...
```