Lecture #23: Type Inference and Unification

Typing In the Language ML

- Examples from the language ML:

```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:

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

compiler deduces that `add` has type `int list → int`.

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

- More interesting case:

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

(underscore means "don’t care" or "wildcard"). In this case, compiler deduces that `count` has type `α list → int`.

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

Doing Type Inference

- Given a definition such as

```ml
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 `α = τ → ρ`, and `τ = δ 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, 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

  \[ 'a \text{ list} = \text{int list} \]

- Easy: \( 'a = \text{int} \)

- How about this:

  \[ 'a \text{ list} = 'b \text{ list list}; 'b \text{ list} = \text{int list} \]

- Also easy: \( 'a = \text{int list}; 'b = \text{int} \)

- On the other hand:

  \[ 'a \text{ list} = 'b \rightarrow 'b \]

  is unsolvable: lists are not functions.

- Also, if we require **finite** solutions, then

  \[ 'a = 'b \text{ list}; 'b = 'a \text{ list} \]

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

**Most General Solutions**

- Rather trickier:

  \[ 'a \text{ list} = 'b \text{ list list} \]

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

  \[ 'a = \text{int list}; 'b = \text{int} \]

  \[ 'a = (\text{int} \rightarrow \text{int}) \text{ list}; 'b = \text{int} \rightarrow \text{int} \]

  etc.

- But prefer a **most general** solution that will be compatible with any possible solution.

- As such solution 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

  \[ 'a = 'b \text{ list} \]

  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 \( T_i \)'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.

- 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:

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

Example of Unification I

• Try to solve \( A = B \), where

\[ A = 'a \rightarrow \text{int}; B = 'b \text{ list} \rightarrow 'b \]

by computing unify \( (A, B) \).

Example of Unification II

• Try to solve \( A = B \), where

\[ A = 'a \rightarrow 'c \text{ list}; B = 'b \rightarrow 'a \]

by computing unify \( (A, B) \).

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

\[ A = 'a; B = ('b, 'a) \text{ pair} \]

by computing unify \( (A, B) \).

• This one is very easy:

\[ 'a \text{ pair} \]

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
\[
A : \text{'a}; \quad B : (\text{'b, 'a})\text{pair}; \quad C : \text{'c;} \quad D : (\text{'d, ('d, 'c)}\text{pair})\text{pair}.
\]
We just did the first one, and the second is almost the same, so we'll just skip those steps.

\( \text{So 'a = 'c = ('d, 'a)}\text{pair; 'b = 'd; 'd is free.} \)

---

Example of Unification V

• Try to solve
\[
\text{'b list = 'a list; 'a → 'b = 'c; 'c → bool = (bool → bool) → bool}
\]
• We unify both sides of each equation (in any order), keeping the bindings from one unification to the next.

\[
\text{'a: bool Unify 'b list, 'a list:}
\]
\[
\quad \text{Unify 'b, 'a}
\]
\[
\text{'b: 'a Unify 'a → 'b, 'c}
\]
\[
\quad \text{bool Unify 'c → bool, (bool → bool) → bool}
\]
\[
\quad \text{Unify 'c, bool → bool:}
\]
\[
\text{'c: 'a → 'b Unify 'a → 'b, bool → bool:}
\]
\[
\quad \text{bool → bool Unify 'a, bool}
\]
\[
\quad \text{Unify 'b, bool:}
\]
\[
\quad \text{Unify bool, bool}
\]

---

Some Type Rules (reprise)

<table>
<thead>
<tr>
<th>Construct</th>
<th>Type</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer literal</td>
<td>int</td>
<td>'a list</td>
</tr>
<tr>
<td>[]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hd (L)</td>
<td>'a</td>
<td>L: 'a list</td>
</tr>
<tr>
<td>tl (L)</td>
<td>'a list</td>
<td>L: 'a list</td>
</tr>
<tr>
<td>E₁ + E₂</td>
<td>int</td>
<td>E₁: int, E₂: int</td>
</tr>
<tr>
<td>E₁ :: E₂</td>
<td>'a list</td>
<td>E₁: 'a, E₂: 'a list</td>
</tr>
<tr>
<td>E₁ = E₂</td>
<td>bool</td>
<td>E₁: 'a, E₂: 'a</td>
</tr>
<tr>
<td>E₁ != E₂</td>
<td>bool</td>
<td>E₁: 'a, E₂: 'a</td>
</tr>
<tr>
<td>if E₁ then E₂ else E₃ fi</td>
<td>'a</td>
<td>E₁: bool, E₂: 'a, E₃: 'a</td>
</tr>
<tr>
<td>E₁ E₂</td>
<td>'b</td>
<td>E₁: 'a → 'b, E₂: 'a</td>
</tr>
<tr>
<td>def f x₁ ... xₙ = E</td>
<td>x₁: 'a₁, ..., xₙ: 'aₙ, E: 'a₀</td>
<td>f: 'a₁ → ... → 'aₙ → 'a₀</td>
</tr>
</tbody>
</table>

---

Using the Type Rules

• Interpret the notation \( E : T, \) where \( E \) is an expression and \( T \) is a type, as
\[ \text{type}(E) = T \]
• 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
\[
\text{def f x = x}
\]
we might start by saying that
\[ \text{type(f) = 'a₀, type(x) = 'a₁} \]
• 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 \( \lambda x. x^2 \).

- To get more than one argument, write
  
  ```
  def f x y = x + y;
  ```

  and \( f \) will have the value \( \lambda x. \lambda y. x + y \).

- Its type will be \( \text{int} \to \text{int} \to \text{int} \).

- So, \( f \ 2 \ 3 = (f \ 2) \ 3 = (\lambda y. 2 + y) \ (3) = 5 \).

- 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 \( \text{init} \) else \( f \ \text{init} \ (\text{hd} \ L) \) 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 \to 'a1, 'L = 'a0, \text{type}(p \ L) = 'a1 \# \text{call rule}
  'L = 'a2 \text{list}, \text{type(hd} \ L) = 'a2 \# \text{hd rule}
  'f = 'a3 \to 'a4, 'init = 'a3, \text{type(f \ init) = 'a4}
  \hspace{1cm} \# \text{call rule}
  'a4 = 'a5 \to 'a6, 'a2 = 'a5, \text{type(f \ init (hd \ L)) = 'a6}
  \hspace{1cm} \# \text{call rule}
  'a1 = \text{bool}, \text{init = 'a7, 'a6 = 'a7, type(if... fi) = 'a7}
  \hspace{1cm} \# \text{if rule}
  'a7 = \text{int, int = int, type(if... fi+3) = int \ # + rule}
  ```

Example, contd.

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

- \( 'p = 'a0 \to 'a1, 'L = 'a0 \)
- \( 'L = 'a2 \text{list} \)
  - \( 'a0 = 'a2 \text{list} \)
- \( 'f = 'a3 \to 'a4, \text{init = 'a3} \)
- \( 'a4 = 'a5 \to 'a6, 'a2 = 'a5 \)
- \( 'a1 = \text{bool, init = 'a7, 'a6 = 'a7} \)
  - \( 'a3 = 'a7 \)
- \( 'a7 = \text{int, int = int} \)

So (eventually),

- \( 'p = 'a5 \text{list} \to \text{bool, 'L = 'a5 \text{list, 'init = int,} \) 
- \( 'f = \text{int} \to 'a5 \to \text{int} \)

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 \( 'a0 \text{ list} \) and \( 'a1 \text{ list} \), which is a good thing, because otherwise, this expres-
sion cannot be made to type-check [why?].

- You’d probably want to do the same with \( \text{count} \):

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

  Analyzing this gives a type of \( 'a \text{ list} \to \text{int} \). Suppose we have two
calls later in the program: \( \text{count} (0::x) \) and \( \text{count} ([1]::y) \).

- Obviously, we also want to replace \( 'a \) in each case with a fresh type
  variable, since otherwise, \( \text{count} \) would be specialized to work only on
  lists of integers or only on lists of lists.
• But we don’t want to introduce a fresh type variable for each call when inferring the type of a function from its definition:

  \[
  \text{fun \ switcher\ x\ y\ z = if x=0 then y else switcher(x-1,z, y) fi}
  \]

• Here, we want the type of \texttt{switcher} to come out to be \texttt{int} → \texttt{'y} → \texttt{'y} → \texttt{'y}, but that can’t happen if the recursive call to \texttt{switcher} can take argument types that are independent of those of \texttt{y} and \texttt{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.