Lecture #23: Conversion and Type Inference

Administrivia.
- Due date for Project #2 moved to midnight tonight.
- Midterm mean 20, median 21 (my expectation: 17.5).

Conversion vs. Subtyping
- In Java, this is legal:
  
  ```java
  Object x = "Hello";
  ```

  - Can explain by saying that static type of string literal is a subtype of Object.
  - That is, any String is an Object.
  - However, Java calls the assignment to x a widening reference conversion.

Integer Conversions
- One can also write:
  
  ```java
  int x = 'c';
  float y = x;
  ```

  The relationship between char and int, or int and float not generally called subtyping.
  - Instead, these are conversions (or coercions), implying there might be some change in value or representation.
  - In fact, in case of int to float, can lose information (example?)

Conversions: Implicit vs. Explicit
- With exception of int to float and long to double, Java uses general rule:
  - Widening conversions do not require explicit casts. Narrowing conversions do.
  - A widening conversion converts a "smaller" type to a "larger" (i.e., one whose values are a superset).
  - A narrowing conversion goes in the opposite direction.
Conversion Examples

• Thus,

Object x = ...
String y = ...
int a = 42;
short b = 17;
x = y; a = b;  // { OK}
y = x; b = a;  // { ERRORS}
x = (Object) y;  // { OK}
a = (int) b;   // { OK}
y = (String) x;  // { OK, but may cause exception}
b = (short) a; // { OK, but may lose information}

• Possibility of implicit coercion can complicate type-matching rules (see C++).

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 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 → 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

(underscore means "don't care" or "wildcard"). In this case, compiler deduces
that count has type \( \alpha \) list → int.
• Here, \( \alpha \) 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:
  - O: int, []: \( \delta \) list.
  - Since add is function and applies to int, must be that \( \alpha = \lambda \to \kappa \),
    and \( \lambda = \delta \) 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, etc.);
  - The **type constructor** $T$ list, where $T$ is a type expression;
  - A **function type** $D \to C$, 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 \to '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 \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 $T_i$'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.
- 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:
  
  \[ \text{unify}(T_1, T_2): \]

  1. If \( T_1 = T_2 \): return true
  2. If \( T_1 \) is a type variable and does not appear in \( T_2 \):
     bind \( T_1 \) to \( T_2 \); return true
  3. If \( T_2 \) is a type variable and does not appear in \( T_1 \):
     bind \( T_2 \) to \( T_1 \); return true
  4. If \( T_1 \) and \( T_2 \) are \( S_1 \) list and \( S_2 \) list:
     return unify \((S_1, S_2)\)
  5. If \( T_1 \) and \( T_2 \) are \( D_1 \rightarrow C_1 \) and \( D_2 \rightarrow C_2 \):
     return unify \((D_1, D_2)\) and unify \((C_1, C_2)\)
  6. else: return false

Example of Unification

• Try to solve
  
  \[- 'b \text{ list} = 'a \text{ list}; 'a \rightarrow 'b = 'c; \]
  \[- 'c \rightarrow \text{bool} = (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} \]

• We unify both sides of each equation (in any order), keeping the bindings from one unification to the next.
  
  \[ 'a: \text{bool} \quad \text{Unify 'b list, 'a list:} \]
  \[ 'b: 'a \quad \text{Unify 'a \rightarrow 'b, 'c} \]
  \[ 'c: 'a \rightarrow 'b \quad \text{Unify 'c \rightarrow \text{bool}, (bool \rightarrow \text{bool}) \rightarrow \text{bool}} \]
  \[ \text{bool} \rightarrow \text{bool} \]

Type Rules for a Small Language

• Each of the \( 'a \), \( 'a_i \) mentioned is a "fresh" type variable, introduced for each application of the rule.

  \[(i\ \text{an integer literal}) \quad i: \text{int} \]
  \[ \quad L: 'a \text{ list} \]
  \[ \quad \text{hd}(L): 'a \]
  \[ \quad \text{tl}(L): 'a \text{ list} \]
  \[ E_1: \text{int} \quad E_2: \text{int} \]
  \[ E_1 + E_2: \text{int} \]
  \[ E_1 : 'a, E_2 : 'a \text{ list} \]
  \[ E_1 :: E_2 : 'a \text{ list} \]
  \[ E_1 = E_2 : \text{bool} \]
  \[ E_1 != E_2 : \text{bool} \]
  \[ E_1: 'a \rightarrow 'b, E_2: 'a \]
  \[ \text{if } E_1 \text{ then } E_2 \text{ else } E_3: 'a \]
  \[ E_1 E_2: 'b \]
  \[ \text{def } f \ x_1 \ldots x_n = E \]
  \[ x_1: 'a_1, \ldots, x_n: 'a_n, f: 'a_1 \rightarrow \ldots \rightarrow 'a_n \rightarrow 'a_0 \rightarrow E: 'a_0 \]
  \[ \text{def } f \ x_1 \ldots x_n = \text{void} \]
  \[ f: 'a_1 \rightarrow \ldots \rightarrow 'a_n \rightarrow 'a_0 \]

Alternative Definition

<table>
<thead>
<tr>
<th>Construct</th>
<th>Type</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{integer literal} ]</td>
<td>\text{int}</td>
<td>\text{L: 'a list}</td>
</tr>
<tr>
<td>[ [] ]</td>
<td>'a \text{ list}</td>
<td>\text{hd(L): 'a}</td>
</tr>
<tr>
<td>[ \text{hd}(L) ]</td>
<td>'a \text{ list}</td>
<td>L: 'a list</td>
</tr>
<tr>
<td>[ \text{tl}(L) ]</td>
<td>'a \text{ list}</td>
<td>L: 'a list</td>
</tr>
<tr>
<td>[ E_1 + E_2 ]</td>
<td>int</td>
<td>E_1: int, E_2: int</td>
</tr>
<tr>
<td>[ E_1 :: E_2 ]</td>
<td>'a \text{ list}</td>
<td>E_1: 'a, E_2: 'a list</td>
</tr>
<tr>
<td>[ E_1 = E_2 ]</td>
<td>bool</td>
<td>E_1: 'a, E_2: 'a</td>
</tr>
<tr>
<td>[ E_1 != E_2 ]</td>
<td>bool</td>
<td>E_1: 'a, E_2: 'a</td>
</tr>
<tr>
<td>[ \text{if } E_1 \text{ then } E_2 \text{ else } E_3 ]</td>
<td>'a</td>
<td>E_1: bool, E_2: 'a, E_3: 'a</td>
</tr>
<tr>
<td>[ E_1 E_2 ]</td>
<td>'b</td>
<td>E_1: 'a \rightarrow 'b, E_2: 'a</td>
</tr>
<tr>
<td>[ \text{def } f \ x_1 \ldots x_n = E ]</td>
<td>x_1: 'a_1, \ldots, x_n: 'a_n, E: 'a_0, f: 'a_1 \rightarrow \ldots \rightarrow 'a_n \rightarrow 'a_0</td>
<td></td>
</tr>
</tbody>
</table>
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 \( \lambda x. x \cdot x \).

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

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

- It's type will be `int \rightarrow int \rightarrow int` (Note: \( \rightarrow \) is right associative).

- 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

```
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 \rightarrow 'a1 \rightarrow 'a2 # def rule
  'L = 'a3 list # = rule, [] rule
  'L = 'a4 list # hd rule,
  'x = 'a4 # != rule
  'x = 'a0 # cal rule
  'L = 'a5 list # tl rule
  'a1 = 'a5 list # tl rule, cal rule
  ```