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:

        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:

    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
```

(`_` 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

\begin{verbatim}
fun add [] = 0
  | add (a :: L) = a + add L
\end{verbatim}

• 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:
  - \(0 : \text{int}, [] : \delta \text{ list}\).
  - Since \(add\) is function and applies to \(\text{int}\), must be that \(\alpha = \iota \rightarrow \kappa\),
    and \(\iota = \delta \text{ list}\)
  - 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** \((\text{int}, \text{bool})\);
  - A **type variable** (today we’ll use ML notation: ‘a, ‘b, ‘c₁, etc.);
  - The **type constructor** \(T\) \text{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, $\tau$, 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:

\[
\text{unify}(T1,T2):
\]
\[
T1 = \text{binding}(T1); \quad T2 = \text{binding}(T2);
\]
\[
\text{if } T1 = T2: \quad \text{return true};
\]
\[
\text{if } T1 \text{ is a type variable and does not appear in } T2: \\
\quad \text{bind } T1 \text{ to } T2; \quad \text{return true}
\]
\[
\text{if } T2 \text{ is a type variable and does not appear in } T1: \\
\quad \text{bind } T2 \text{ to } T1; \quad \text{return true}
\]
\[
\text{if } T1 \text{ and } T2 \text{ are } S1 \text{ list and } S2 \text{ list: } \quad \text{return unify}(S1,S2)
\]
\[
\text{if } T1 \text{ and } T2 \text{ are } D1 \rightarrow C1 \text{ and } D2 \rightarrow C2: \\
\quad \text{return unify}(D1,D2) \text{ and unify}(C1,C2)
\]
\[
\text{else: } \quad \text{return false}
\]
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.

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

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

\[
\begin{align*}
'a: & \\
'b: & 'a \\
'c: & 'a \rightarrow 'b
\end{align*}
\]

- Unify 'b list, 'a list:
  \[
  \begin{align*}
  &\text{Unify 'b, 'a} \\
  &\text{Unify 'a \rightarrow 'b, 'c} \\
  &\text{Unify 'c \rightarrow \text{bool, (bool \rightarrow \text{bool}) \rightarrow \text{bool}}
  \end{align*}
\]

- Unify 'c, bool \rightarrow bool:

- Unify 'a \rightarrow 'b, bool \rightarrow bool:
Example of Unification

• Try to solve

   \[ \text{\textbf{'b list= 'a list; 'a→ 'b = 'c;}} \]
   \[ \text{'c → bool= (bool→ bool) → bool} \]

• We unify both sides of each equation (in any order), keeping the bindings from one unification to the next.

   \[
   \begin{align*}
   \text{'a: bool} & \quad \text{Unify 'b list, 'a list:} \\
   \text{'b: 'a} & \quad \text{Unify 'b, 'a} \\
   \text{'c: 'a → 'b} & \quad \text{Unify 'a→ 'b, 'c} \\
   \text{'c → bool} & \quad \text{Unify 'c → bool, (bool→ bool) → bool} \\
   \text{Unify 'c, bool → bool:} \\
   \text{Unify 'c, bool → bool:} \\
   \text{Unify 'a → 'b, bool → bool:} \\
   \text{Unify 'a, bool}
   \end{align*}
   \]
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  
   'b: 'a  
   'c: 'a -> 'b  

Unify 'b list, 'a list:
   Unify 'b, 'a
Unify 'a -> 'b, 'c
   Unify 'c -> bool, (bool -> bool) -> bool
   Unify 'c, bool -> bool:
   Unify 'a, bool
   Unify 'b, bool:
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.

\[
\begin{align*}
\text{'a: bool} & \quad \text{Unify 'b list, 'a list:} \\
\text{'b: 'a} & \quad \text{Unify 'a \rightarrow 'b, 'c} \\
\text{'c: 'a \rightarrow 'b} & \quad \text{Unify 'c \rightarrow \text{bool}, (\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool:}} \\
\end{align*}
\]

\[
\begin{align*}
\text{Unify 'b, 'a} & \\
\text{Unify 'c \rightarrow \text{bool, (bool \rightarrow \text{bool}) \rightarrow \text{bool:}} \\
\text{Unify 'c, bool \rightarrow \text{bool:}} \\
\text{Unify 'a \rightarrow 'b, bool \rightarrow \text{bool:}} \\
\text{Unify 'a, bool} \\
\text{Unify 'b, bool:} \\
\text{Unify bool, bool}
\end{align*}
\]
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
    bool
    Unify ’a → ’b, ’c
    Unify ’c → bool, (bool → bool) → bool
      Unify ’c, bool → bool:
’c: ’a → ’b
  bool → bool
    Unify ’a → ’b, bool → bool:
      Unify ’a, bool
      Unify ’b, bool:
        Unify bool, bool
        Unify bool, 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 an integer literal)  \[ i : \text{int} \]

\[ \text{[]} : \text{'a list} \]

\[ \text{hd}(L) : \text{'a} \]

\[ \text{tl}(L) : \text{'a list} \]

\[ E_1 : \text{int} \quad E_2 : \text{int} \]

\[ E_1 + E_2 : \text{int} \]

\[ E_1 : \text{'a} \quad E_2 : \text{'a list} \]

\[ E_1 :: E_2 : \text{'a list} \]

\[ E_1 : \text{'a} \quad E_2 : \text{'a} \]

\[ E_1 = E_2 : \text{bool} \]

\[ E_1 \neq E_2 : \text{bool} \]

\[ E_1 : \text{bool} \quad E_2 : \text{'a} \quad E_3 : \text{'a} \]

\[ \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \text{'a} \]

\[ x_1 : \text{'a}_1, \ldots, x_n : \text{'a}_n, f : \text{'a}_1 \rightarrow \ldots \rightarrow \text{'a}_n \rightarrow \text{'a}_0 \vdash E : \text{'a}_0 \]

\[ \text{def } f \ x_1 \ldots \ x_n = E : \text{void} \]

\[ f : \text{'a}_1 \rightarrow \ldots \rightarrow \text{'a}_n \rightarrow \text{'a}_0 \]
### Alternative Definition

<table>
<thead>
<tr>
<th>Construct</th>
<th>Type</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer literal</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>'a list</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</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></td>
<td>x₁: 'a₁, ..., xₙ: 'aₙ E: 'a₀, f: 'a₁ → ... → 'aₙ → 'a₀</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

\[
\text{def } \text{sqr } x = x \times x;
\]

means essentially that \text{sqr} is defined to have the value \( \lambda \; x. \; x \times x \).

- To get more than one argument, write

\[
\text{def } f \; x \; y = x + y;
\]

and \text{f} will have the value \( \lambda \; x. \; \lambda \; y. \; x + y \)

- It’s type will be \text{int} \rightarrow \text{int} \rightarrow \text{int} (\text{Note: } \rightarrow \text{ is right associative}).

- So, \text{f} 2 3 = (\text{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 → '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
  ...

Last modified: Fri Oct 20 10:46:40 2006