Type Reconstruction and Inference

Lecture 20
CS 565
4/19/06
Type Variables

- Two separate issues:
  - Describing a mapping from type variables to types
  - Applying the mapping to yield instances
  - Must guarantee that types are preserved under substitution: if $\sigma$ is a type substitution and $\Gamma \vdash t : \tau$, then $\sigma\Gamma \vdash \sigma t : \sigma\tau$
View of Type Variables

- Keep type variables abstract during type checking
  - A well-typed term behaves properly regardless of the concrete types that are substituted for the variables: (System F)
    - \( \lambda f : \tau \rightarrow \tau \cdot \lambda x : \tau. f (f \ x) \)
- The term may not be well-typed, but there exists an instantiation of type variables that make it well-typed: (Type Inference)
  - \( \lambda f : \tau \cdot \lambda x : \tau'. f (f \ x) \)
Inference

- If we omit type parameters, we must discover whether the intended use of an expression matches its actual use.

- Implications for compilation:
  - How do we generate code for a polymorphic procedure that may be applied to objects with very different representations?

- First need to understand how inference works.
Constraint-Based Typing

- Constraints define equation between type expressions that may contain type variables
- Typing rules calculates types (and their constraints)
- Validate the correctness of a given set of constraints under a substitution
**Constraints (Example)**

\[
\begin{align*}
\Gamma & \vdash t_1 : \tau | C \\
C' & = C \cup \{ \tau = \text{int} \} \\
\Gamma & \vdash \text{pred } t_1 : \text{int} | C'
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash t_1 : \tau_1 | C_1 \\
\Gamma & \vdash t_2 : \tau_2 | C_2 \\
C' & = C_1 \cup C_2 \cup \{ \tau_1 = \tau_2 \rightarrow X \} \\
X & \text{ fresh} \\
\Gamma & \vdash t_1 \ t_2 : X | C'
\end{align*}
\]

Constraints for \( t = \lambda x : X \rightarrow Y. (x \ 0) \) is

\[
\{ \text{int } \rightarrow Z = X \rightarrow Y \}
\]
Unification

- Allows us to calculate a solution to a constraint set:

unify (C) =

  if c is empty then []
  else let \{ S = T \} U C' = C in

  if S = T
    then unify(C')
    else if S = X and X not \in\ FV(T)
      then unify([X \rightarrow T]C') o [X \rightarrow T]
    else if T = X and X not \in\ FV(S)
      then unify([X \rightarrow S]C') o [X \rightarrow S]
    else if S = S1 \rightarrow S2 and T = T1 \rightarrow T2
      then unify(C' U \{ S1 = T1, S2 = T2\})
  else fail
Type-checking

- Match type operators and instantiate type variables.
- Need to define where type variables can appear.
- Must also enforce contextual dependencies:
  - 'a → 'a : substituting “int” for 'a must be done uniformly for all occurrences of the type variable in the type.
Type-checking

- Perform context-sensitive type instantiation using unification.
  - Unification fails when
    - trying to match two distinct type operators (int and bool)
    - instantiating a type variable to a term containing that variable ('a and 'a → int)
  - Example: try to type-check the following expression:
    \[ \text{fn } x \rightarrow x(x) \]
Example

The type of length in the following program:

```ml
let fun length l = if (null l)
    then 0
    else succ(length(tl(l)))
in ... 
```

is 'a list -> int. How does the ML typechecker deduce this type?

Perform a bottom-up inspection of the program, matching and synthesizing types while proceeding to the root.

- The type of an expression is computed from the type of its subexpressions and the type constraints imposed by the context.
- Important property: order in which we examine programs and perform unification does not affect final result.


Consider the type of length. Perform type-checking using a bottom-up derivation:

<table>
<thead>
<tr>
<th>l:</th>
<th>'a</th>
<th>type of l initially unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>null:</td>
<td>'b list -&gt; bool</td>
<td>definition of null</td>
</tr>
<tr>
<td>null(l):</td>
<td>bool</td>
<td>by definition of null</td>
</tr>
<tr>
<td>0:</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>tl:</td>
<td>'c list -&gt; 'c list</td>
<td>by definition of tl</td>
</tr>
<tr>
<td>tl(l):</td>
<td>'c list</td>
<td>unification</td>
</tr>
<tr>
<td>l:</td>
<td>'c list</td>
<td></td>
</tr>
</tbody>
</table>
### Example (cont)

```
let fun length l = if (null l) then 0 else succ(length(tl(l)))
```

<table>
<thead>
<tr>
<th>length:</th>
<th>'a → 'd</th>
<th>by definition of fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>length(tl(l)):</td>
<td>'d</td>
<td>unification</td>
</tr>
<tr>
<td></td>
<td>'a = 'c list</td>
<td></td>
</tr>
<tr>
<td>succ:</td>
<td>int → int</td>
<td>by definition of succ</td>
</tr>
<tr>
<td>succ(length(..)):</td>
<td>int</td>
<td>unification</td>
</tr>
<tr>
<td></td>
<td>'d = int</td>
<td></td>
</tr>
<tr>
<td>if (null...):</td>
<td>int</td>
<td>by definition of conditional</td>
</tr>
<tr>
<td>fn l =&gt; ...</td>
<td>'c list → int</td>
<td></td>
</tr>
</tbody>
</table>
Basic algorithm

1. A variable \( x \) introduced as a function argument assigned a new type variable. Store \(<x,'a>\) in a type environment, where 'a is fresh.

2. In a conditional, predicate type unified with bool, the true and false branch unified with one another. This type (call it 'b) is the type of the conditional.

3. The type of \( e \) in a function \( fn \ x \Rightarrow e \) is inferred in a context where \( x \) is associated with a new type variable.

4. In an application, \( (f \ x) \), \( f \) is unified against \( A \Rightarrow 'b \) where \( A \) is the type of \( x \) and 'b is a new type variable.

   The type of \( f \) is therefore a function type whose domain is unifiable to 'b. 'b (or its instantiation) is returned as the type of the function.
To type-check let expressions introduce notion of genericity.

What is the type of the expression:

- fn f => (f(3), f(true))

Cannot type this in ML because f’s type is considered non-generic.

- The first occurrence of f determines a type int → 'a, and the second determines a type bool → 'a.
- Can’t unify these two terms

Non-generic type variables cannot be instantiated multiple times within their defined context.

To implement generic types, make a copy of the type for every distinct context in which it occurs.
Algorithm (cont)

- What about:
  
  ```ml
  let val f = fn x => x
  in (f(3), f(true))
  end
  ```

- Here, we will assign `f` type `'a -> 'a` and view `'a` as generic:
  - `'a` can assume different values for different instantiations of `f` in the let-body.
Algorithm (cont)

- Need to be careful to not copy non-generic variables:
  
  ```
  let val f = fn g => let val h = g
               in pair(h(3),h(true))
               end
  
  in ...
  
  end
  ```

Def. A type variable occurring in the type of an expression `e` is generic (with respect to `e`) iff it does not occur in the type of the binder of any function definition enclosing `e`. 
Algo{rithm} (cont)

To typecheck a let expression, typecheck its declaration, obtaining an environment of identifiers and types used to typecheck the let-body.

Recursive definitions:

\[
\text{let fun } f(...) = \ldots \ f \ldots \\
\text{in } \ldots f \ldots
\]

Instances of the type variable in the recursive definition must be non-generic, while instances in the body are generic.