Type Inference

November 29, 2001
CS 131: Programming Languages

The Type Checking Problem
• Given a program where the type of every variable is known, determine whether the program is well-typed.

```
fun f(x : bool) : real =
  if x then
    3.0
  else
    2.0 * f (not x)
```

• Straightforward if we have principal types.

The Type Inference Problem
• Given a program with some or all type annotations missing, can types be inserted to make the program typecheck?

```
fun f(x) =
  if x then
    3.0
  else
    2.0 * f (not x)
```

• Sometimes called type reconstruction.

An "Algorithm" for Type Inference (without Polymorphism)
1. Allocate a metavariable for each missing type annotation.

```
fun f(x : M_1) : M_2 =
  if x then
    3.0
  else
    2.0 * f (not x)
```
"Algorithm" continued

2. Compute the type of each subexpression in terms of these metavariables. Figure out all the constraints that a type checker would expect to hold

```haskell
fun f(x : M1) : M2 =
    if x then
        3.0
    else
        2.0 * f(not x)
```

"Algorithm" continued

3. Find values of the metavariables such that all these equational constraints are satisfied.

Solving Constraints

- What is a solution to a set of constraints?
  - A type for each metavariable, such that, when these are plugged in all the equations become true.

- Does this idea sound familiar?
  - Say, from Prolog in CS 60?
  - Or (more recently) from Resolution Theorem Proving in CS 80?
Unification

- General problem:
  - Given two "phrases" containing constants and variables, find values for the variables that makes the two phrases equal.

Unification Specification

- If asked to unify \( \text{int} \) with \( \text{int} \), we don't have to do anything.
  - Same for any other base types.
- If asked to unify \( t_1 \rightarrow t_2 \) with \( u_1 \rightarrow u_2 \) it is necessary and sufficient to find values for metavariables making \( t_1 = u_1 \) and \( t_2 = u_2 \) both true.
  - Similar for \( t_1 \ast t_2 \) and \( u_1 \ast u_2 \)
- If asked to unify a metavariable with itself, we don't have to do anything.
- If asked to unify a metavariable \( M \) with any other type \( t \),
  - We can simply define the value of \( M \) to be \( t \), so long as \( M \) doesn't already have a definition and as long as the type \( t \) does not involve \( M \).
    - Latter condition is called the "occurs check", and prevents circular definitions
    - By the way, Prolog skips this check for speed purposes.
  - If \( M \) already has a definition, then we just need to check that this definition unifies with \( t \).

Another Example

\[
((\text{fn } f \Rightarrow f) \ (\text{fn } x \Rightarrow x))(3)
\]

Another Example

\[
\text{fn } f \Rightarrow (f \ 0) + (f \ \text{true})
\]
Another Example

\((\text{fn } x \Rightarrow x \ x)(\text{fn } x \Rightarrow x \ x)\)

ML Polymorphism
(a.k.a. Hindley-Milner Polymorphism)
(a.k.a. Let-polymorphism)

- Clearly, some pieces of SML code do not yield unique solutions for the omitted types: \(\text{fn } x \Rightarrow x\)
- In SML, variables defined via \textit{val} or \textit{fun} are allowed to be considered generic/parametric in unconstrained metavariables (after all definitions are expanded out).

```
let val id = (fn x => x)
in
  (id 3, id true)
end
```

Modifications to Type Inference

- When typechecking a variable definition, need to figure out "how polymorphic" the definition is \textit{before} we can typecheck uses of that definition.
  - We need to completely finish type inference on the definition before going on.
- Forced to interleave constraint generation and solving
  - More efficient anyway.
  - Implementations don't actually "construct" constraints to be solved later, but just invoke \texttt{unify}

Handling Polymorphism

- When typechecking a definition like
  ```
  val f = fn x => fn y => x
  ```
  we first do type inference on the definition, yielding
  \(M_1 \rightarrow M_2 \rightarrow M_1\)
  with \(M_1\) and \(M_2\) unconstrained.
- So we plug in two type \texttt{variables} and universally quantify, yielding the polymorphic type
  \(\forall (\texttt{a}, \texttt{b}). \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{a}\)
  which, as you recall, SML prints out just as
  ```
  val f : \texttt{a} \rightarrow \texttt{b} \rightarrow \texttt{a}
  ```
Handing Polymorphism

• When we come across the variable $f$ actually being used, we know that it is being used as a function of type $t_1 \rightarrow t_2 \rightarrow t_1$, except that we might not know $t_1$ and $t_2$ yet.

• No problem...just plug in fresh metavariables for the polymorphic variables `'a` and `'b`!

Example

```
let val id = (fn x => x)
in
    (id 3, id true)
end
```

Pitfall...

• If we find that the type of a defined variable involves a metavariable without a definition, does it follow that this variable is polymorphic?

```
fun foo(x) =
    let val y = x
    in
        y+y
    end
```
Constrained Metavariables

- An unset metavariable with still *constrained* if it also occurs in the type of some variable already in scope.
  - And said to be *unconstrained* otherwise.
- We can only make definitions polymorphic in unconstrained, unset metavariables.
  - Value restriction: if the definition is not a value, we can't even make it polymorphic in unconstrained, unset metavariables!

Alternate Approach to Polymorphism

- Whenever you see
  ```ml
  let val x = e₁ in e₂
  ```
  (where e₁ is a value) use the monomorphic algorithm on the program
  ```ml
  e₂[ x → e₁ ]
  ```

let val id = (fn x => x)
in  
  (id 3, id true)
end

Complexity Results

- Given a monomorphic expression of length \( n \),
  - Determining whether the expression has a type (and if so what type) can be done in time \( O(n) \).
  - However, the type may have length \( O(2^n) \)
- Given a polymorphic expression of length \( n \),
  - Determining whether the expression has a type (and if so what type) can be done in time \( O(2^n) \).
  - However, the type may have length \( O(2^{2^n}) \)
- In practice, algorithm is much faster.