Computer Science 131
Programming Languages

October 26, 2000
Type Inference
Part 1

Simply-Typed Languages
(No Polymorphism)
The Type Inference Problem

• Given code with missing type annotations
  - Is there a way to reconstruct type annotations that makes the code typecheck?
  - If so, what are these annotations?

\[
\begin{align*}
\text{fn } x &\Rightarrow (x + x) \\
((\text{fn } f \Rightarrow f)(\text{fn } x \Rightarrow x))(3) \\
\text{fn } f &\Rightarrow (f \, 0) + (f \, \text{tt}) \\
\text{fn } f &\Rightarrow f(f) \\
\text{fn } x &\Rightarrow x
\end{align*}
\]
Source Language

t ::= Int | Bool | t₁ -> t₂ | t₁ * t₂

v ::= n | tt | ff

e ::= v | e₁ + e₂ | e₁ < e₂
  | if e₁ then e₂ else e₃
  | x
  | fn x => e | e₁ e₂
  | <e₁,e₂> | e.1 | e.2
Annotated Language

\[ t ::= \text{Int} \mid \text{Bool} \mid t_1 \rightarrow t_2 \mid t_1 \ast t_2 \]

\[ v ::= n \mid \text{tt} \mid \text{ff} \]

\[ e ::= v \mid e_1 + e_2 \mid e_1 < e_2 \]
\[ \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]
\[ \mid x \]
\[ \mid \text{fn } (x : t) \Rightarrow e \mid e_1 e_2 \]
\[ \mid <e_1,e_2> \mid e.1 \mid e.2 \]
Simple Type Inference

- Given: Expression in source language.
  1. Translate to annotated language by inserting metavariables (placeholders) representing unknown types
  2. Create metavariables representing the types of each sub-expression
  3. Determine the constraints that the metavariables must obey
  4. Solve this system of constraints.
Example

• Infer types for the expression

\[ \text{fn } x \Rightarrow x + x \]

• Step 1:
  - Allocate a metavariable for each unknown type annotation.

\[ \text{fn } (x:\tau_1) \Rightarrow x + x \]
Example

• Step 2:
  - Give names to the types of all subexpressions.

\[
\begin{array}{c}
\tau_5 \\
\text{\hspace{1cm} \tau_4} \\
\quad \tau_2 \quad \tau_3 \\
\text{\hspace{1cm} \quad x + x} \\
\text{\textit{fn} (x:\tau_1) => x + x}
\end{array}
\]
Example

• Step 3:
  - Determine the constraints these variables must satisfy
Example

• Step 4:
  - Find a solution to these constraints
Other Examples

$$((\text{fn } f \Rightarrow f)(\text{fn } x \Rightarrow x))(3)$$

$$\text{fn } x \Rightarrow x$$
Other Examples

\[ \text{fn } f \Rightarrow (f \_0) + (f \_tt) \]

\[ \text{fn } f \Rightarrow f(f) \]
Constraint Solving

• What is a solution to a set of constraints?
  - A type for each metavariable
  - Can be viewed as a substitution.
  - When these types are plugged in, all the equations become identities

• Does this sound familiar?
Unification

• General statement:
  - Given two "phrases" containing constants and variables, find a substitution that makes the two phrases equal.

  - Defines a function \( \text{Unify}(t_1, t_2) \)
    • Returns a substitution if one exists
    • Otherwise, fails

  - See your CS 80 notes
Unification Algorithm for Types

\[
\begin{align*}
    \text{Unify}(\text{Int}, \text{Int}) &= \text{id} \\
    \text{Unify}(\tau, \tau) &= \text{id} \\
    \text{Unify}(\tau, t) &= \text{if} \ (\tau \text{ occurs in } t) \ \text{then} \ Fail \\
    &\quad \text{else } \text{id}, [\tau \rightarrow t] \\
    \text{Unify}(t, \tau) &= \ldots \text{similar} \ldots \\
    \text{Unify}(t_1 \rightarrow t_2, t_1' \rightarrow t_2') &= \\
    &\quad \text{let } S_1 = \text{Unify}(t_1, t_1') \\
    &\quad \quad S_2 = \text{Unify}(S_1(t_2), S_1(t_2')) \\
    &\quad \quad \text{in} \\
    &\quad \quad \quad S_2 \circ S_1 \\
    &\quad \quad \text{end} \\
    \text{Unify}(t_1 \ast t_2, t_1' \ast t_2') &= \ldots \text{similar} \ldots \\
\text{Otherwise, Fail.}
\end{align*}
\]
Part 2

Hindley-Milner Polymorphism
(aka ML-style Polymorphism)
(aka Let-polymorphism)
An Unannotated Language

t ::= Int | Bool | t₁ → t₂ | t₁ * t₂

v ::= n | tt | ff

e ::= v | e₁ + e₂ | e₁ < e₂
| if e₁ then e₂ else e₃
| x
| let x be e₁ in e₂
| fn x => e | e₁ e₂
| <e₁,e₂> | e.1 | e.2
An Annotated Language

\[
\begin{align*}
u & ::= \text{Int} \mid \text{Bool} \mid u_1 \to u_2 \mid u_1 \times u_2 \mid \alpha \\
t & ::= u \mid \forall \alpha . t \\
v & ::= n \mid \text{tt} \mid \text{ff} \\
e & ::= v \mid e_1 + e_2 \mid e_1 < e_2 \\
& \quad | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
& \quad | x \\
& \quad | \text{let } x : t \text{ be } e_1 \text{ in } e_2 \\
& \quad | \text{fn } (x : u) \Rightarrow e \mid e_1 e_2 \\
& \quad | <e_1, e_2> \mid e.1 \mid e.2 \\
& \quad | \text{Fn } \alpha \Rightarrow e \mid e[u]
\end{align*}
\]
Let-Polymorphism

• Idea:
  - For some definitions, constraints do not yield unique solution
    ```
    let id be (fn x => <x,x>) in
    <id 3, id tt>
    ```
  - Allow definitions like `id` to be parametric in unconstrained type variables.
    ```
    let id be (Fn α => fn x:α => <x,x>) in
    <id[Int] 3, id[Bool] tt>
    ```
Algorithm Modifications

• Must solve constraints to see "how polymorphic" a definition is.
• This must be done before typechecking any uses of this definition.

• Must intermix constraint generation and solving
  - More efficient anyway
  - Don't actually create constraints, just call Unify
Adding Polymorphic Inference

• When processing

\[
\text{let } x = e_1 \text{ in } e_2
\]

first typecheck \(e_1\), make it polymorphic in unconstrained metavariables, then do \(e_2\).

• When you come across a variable, insert as many type applications (to metavariables) as necessary in order to make it monomorphic.

• Keep everything else the same.
Example

let id be (fn x => <x,x>)
in
  <id 3, id tt>
end
Careful...

• When is a metavariable unconstrained?
• Suppose that while typechecking the body of a function, we come across
  
  \[ \text{let } x = e_1 \text{ in } e_2 \]

  and that the type of \( e_1 \) is \( \tau \rightarrow \tau \).

• Does it follow that \( x \) should be polymorphic?
Generalizable Type Variables

• Consider the code

```plaintext
fun foo(x) =
  let y be x
  in
  y + y
```
Constrained Metavariabes

• A metavariable is constrained when
  - It is the type of some variable in scope
    • More generally, when there is a known relationship between the metavariable and the type of some variable in scope.
  - Or, it is known to be equal to a type that is not a metavariable.
Milner's Algorithm $W$ (Excerpts)

\[
W(\Gamma, \underline{n}) = (\text{id}, \text{int})
\]
\[
W(\Gamma, x) = (\text{id}, t[\alpha_i \rightarrow \tau_i])
\]
\[
\text{where } \Gamma(x) = \forall \alpha_1 \cdots \forall \alpha_n. \tau
\]
and $\tau_1,\ldots,\tau_n$ are fresh metavariables

\[
W(\Gamma, \text{fn } x \Rightarrow e) = \text{let } (S, t) = W((\Gamma, x : \tau), e)
\]
\[
in (S, S(t) \Rightarrow t)
\]
\[
\text{where } \tau \text{ is a fresh metavariable}
\]

\[
W(\Gamma, e_1 \; e_2) = \text{let } (S_1, t_1) = W(\Gamma, e_1)
\]
\[
(S_2, t_2) = W(S_1(\Gamma), e_2)
\]
\[
S_3 = U(S_2 \; t_1, \; t_2 \Rightarrow \tau)
\]
\[
in (S_3 \circ S_2 \circ S_1, \; S_3(\tau))
\]
\[
\text{where } \tau \text{ is fresh}
\]
Milner's Algorithm $W$ (Excerpts)

$W(\Gamma, \text{let } x \text{ be } e_1 \text{ in } e_2) =$

let $(S_1, t_1) = W(\Gamma, e_1)$

$(S_2, t_2) = W((S_1(\Gamma), x : \text{Clos}(S_1(\Gamma), t_1)), e_2)$

in $(S_2 \circ S_1, t_2)$

Where $\text{Clos}(\Gamma, t)$ is obtained by taking $t$ and universally quantifying over all unconstrained metavariables in $t$ that do not appear in the context $\Gamma$.

For example,

$\text{Clos}((x : \tau_1 \rightarrow \tau_1), \tau_1 * \tau_2 * \tau_3) = \forall \alpha_2 \forall \alpha_3. \tau_1 * \alpha_2 * \alpha_3$
Practical Type Inference

- Actual implementations
  - Don't explicitly construct constraints
    - While walking over program, instead of recording a constraint, call \texttt{Unify} immediately, as in Algorithm W
  - Use imperative implementations of unification
    - Instead of remembering the substitution \( id, [\alpha \rightarrow t] \) simply set \( \alpha := t \)
    - Avoids having to compose or apply substitutions.
Let-Polyomorphism

• Almost equivalent formulation
  - Whenever you see
    \[
    \text{let } x \text{ be } e_1 \text{ in } e_2
    \]
    typecheck as if the user wrote
    \[
    e_2[x \mapsto e_1]
    \]
  - Consider
    \[
    \text{let } id \text{ be } (\text{fn } x \Rightarrow <x,x>) \text{ in } <id \_3, \_id \tt>
    \]
Complexity Results

- Given simply-typed 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 ML-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 $O(2^{2n})$
- In practice, much better
Complexity Results

- If we add in ordinary recursive functions
  \[
  \text{fun } f(x:u_1):u_2 \text{ is } e
  \]
  then the problem isn't any harder.

- If we add polymorphic recursive functions
  \[
  \text{Fun } f(\alpha):u_2 \text{ is } e
  \]
  then type inference is undecidable.
Problems

• Full type inference via unification is very successful for SML.

• But,
  - Restriction to prenex polymorphism
  - Hard to explain what went wrong when inference fails.
  - Does not extend when combined with subtyping

• Open research question how to do better.
  - Also, other applications.