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?

```
fn x => (x + x)
((fn f => f)(fn x => x))(3)
fn f => (f 0) + (f tt)
fn f => f(f)
fn x => x
```

Source Language

$$t ::= \mathtt{Int} \mid \mathtt{Bool} \mid t_1 \text{ -> } t_2 \mid t_1 * t_2$$

$$v ::= \underline{n} \mid \underline{tt} \mid \underline{ff}$$

$$\begin{aligned} e ::= {} & v \mid e_1 + e_2 \mid e_1 < e_2 \\ \mid {} & \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \\ \mid {} & x \\ \mid {} & \mathtt{fn}\ x \Rightarrow e \mid e_1\ e_2 \\ \mid {} & \langle e_1, e_2 \rangle \mid e.1 \mid e.2 \end{aligned}$$

Annotated Language

$$
\begin{array}{lll}
\mathtt{t} & \mathtt{::=} & \mathtt{Int} \mid \mathtt{Bool} \mid \mathtt{t}_1 \;\mathtt{->}\; \mathtt{t}_2 \mid \mathtt{t}_1 * \mathtt{t}_2 \\
\\
\mathtt{v} & \mathtt{::=} & \underline{\mathtt{n}} \mid \underline{\mathtt{tt}} \mid \underline{\mathtt{ff}} \\
\mathtt{e} & \mathtt{::=} & \mathtt{v} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{e}_1 < \mathtt{e}_2 \\
 & \mid & \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \\
 & \mid & \mathtt{x} \\
 & \mid & \mathtt{fn}\ (\mathtt{x} : \mathtt{t})\ \mathtt{=>}\ \mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2 \\
 & \mid & <\mathtt{e}_1,\mathtt{e}_2> \mid \mathtt{e.1} \mid \mathtt{e.2}
\end{array}
$$

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

```
fn x => x + x
```

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

fn (x:t_1**) => x + x**

Example

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

$$\overbrace{\texttt{fn (x:}t_1\texttt{) => } \overbrace{\overbrace{\texttt{x}}^{t_2} \texttt{ + } \overbrace{\texttt{x}}^{t_3}}^{t_4}}^{t_5}$$

Example

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

Example

- Step 4:
  - Find a solution to these constraints

Other Examples

```
((fn f => f)(fn x => x))(3)
```

```
fn x => x
```

Other Examples

```
fn f => (f 0) + (f tt)
```

```
fn f => 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 **`Unify`**$(\mathbf{t_1},\mathbf{t_2})$
    - Returns a substitution if one exists
    - Otherwise, fails
  - See your CS 80 notes

Unification Algorithm for Types

Unify(Int,Int) = id

Unify(t,t) = id

Unify(t,t) = if (t occurs in t) then Fail

else id,[t$\rightarrow$t]

Unify(t,t) = ...*similar*...

Unify(t_1->t_2, t_1'->t_2') =

let S_1 = Unify(t_1,t_1')

S_2 = Unify(S_1(t_2), S_1(t_2'))

in

S_2 o S_1

end

Unify(t_1*t_2, t_1'*t_2') = ...*similar*...

Otherwise, Fail.

Part 2

Hindley-Milner Polymorphism

(aka ML-style Polymorphism)

(aka Let-polymorphism)

An Unannotated Language

$$\begin{array}{lll}
\mathtt{t} & \mathtt{::=} & \mathtt{Int} \mid \mathtt{Bool} \mid \mathtt{t}_1 \ \mathtt{\text{->}} \ \mathtt{t}_2 \mid \mathtt{t}_1 \ \mathtt{*} \ \mathtt{t}_2 \\
 & & \\
\mathtt{v} & \mathtt{::=} & \underline{\mathtt{n}} \mid \underline{\mathtt{tt}} \mid \underline{\mathtt{ff}} \\
\mathtt{e} & \mathtt{::=} & \mathtt{v} \mid \mathtt{e}_1 + \mathtt{e}_2 \mid \mathtt{e}_1 < \mathtt{e}_2 \\
 & \mid & \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3 \\
 & \mid & \mathtt{x} \\
 & \mid & \color{red}{\mathtt{let}\ \mathtt{x}\ \mathtt{be}\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2} \\
 & \mid & \mathtt{fn}\ \mathtt{x}\ \mathtt{=>}\ \mathtt{e} \mid \mathtt{e}_1\ \mathtt{e}_2 \\
 & \mid & \mathtt{<}\mathtt{e}_1\mathtt{,}\mathtt{e}_2\mathtt{>} \mid \mathtt{e.1} \mid \mathtt{e.2}
\end{array}$$

An Annotated Language

u ::= Int | Bool | u_1 -> u_2 | u_1 * u_2 | a

t ::= u | "a.t

v ::= $\underline{n}$ | $\underline{tt}$ | $\underline{ff}$

e ::= v | e_1 + e_2 | e_1 < e_2

| if e_1 then e_2 else e_3

| x

| let x : t be e_1 in e_2

| fn (x : u) => e | e_1 e_2

| <e_1,e_2> | e.1 | e.2

| Fn a => e | e[u]

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 a => fn x:a => <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

  $$\mathbf{let\ x = e_1\ in\ e_2}$$

  first typecheck $\mathbf{e_1}$, make it polymorphic in *unconstrained* metavariables, then do $\mathbf{e_2}$.
- 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

  `let x = e`$_1$ **`in e`**$_2$

  and that the type of **`e`**$_1$ is t **`->`** t.
- Does it follow that **`x`** should be polymorphic?

Generalizable Type Variables

- Consider the code

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

Constrained Metavariables

- 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)

$\mathtt{W(}G\mathtt{,\underline{n}) = (id, int)}$

$\mathtt{W(}G\mathtt{,x) = (id, t[}a_{\mathtt{i}}®t_{\mathtt{i}}\mathtt{])}$

$\qquad\qquad\qquad\qquad\mathtt{where}\ G\mathtt{(x) = }\text{"}a_{\mathtt{1}}\ldots\text{"}a_{\mathtt{n}}\mathtt{.}t$

$\qquad\qquad\mathtt{and}\ t_{\mathtt{1}}\mathtt{,\ldots,}t_{\mathtt{n}}\ \mathtt{are\ fresh\ metavariables}$

$\mathtt{W(}G\mathtt{,fn\ x=>e) = let\ (S,t) = W((}G\mathtt{,x:}t\mathtt{),e)}$

$\qquad\qquad\mathtt{in\ (S,\ S(}t\mathtt{)->t)}$

$\qquad\qquad\qquad\mathtt{where}\ t\ \mathtt{is\ a\ fresh\ metavariable}$

$\mathtt{W(}G\mathtt{,e_1\ e_2) = let\ (S_1,t_1) = W(}G\mathtt{,e_1)}$

$\qquad\qquad\qquad\mathtt{(S_2,t_2) = W(S_1(}G\mathtt{),e_2)}$

$\qquad\qquad\qquad\mathtt{S_3 = U(S_2t_1,\ t_2->}t\mathtt{)}$

$\qquad\qquad\mathtt{in\ \ (S_3\ o\ S_2\ o\ S_1,\ S_3(}t\mathtt{))}$

$\qquad\qquad\qquad\qquad\qquad\mathtt{where}\ t\ \mathtt{is\ fresh}$

Milner's Algorithm W (Excerpts)

$$\mathbf{W(}\mathrm{G}\mathbf{,let\ x\ be\ e_1\ in\ e_2)\ =}$$

$$\mathbf{let\ (S_1,t_1)\ =\ W(}\mathrm{G}\mathbf{,e_1)}$$

$$\mathbf{(S_2,t_2)\ =\ W((S_1(}\mathrm{G}\mathbf{),x:Clos(S_1(}\mathrm{G}\mathbf{),t_1),\ e_2)}$$

$$\mathbf{in\ (S_2\ o\ S_1,\ t_2)}$$

Where **Clos(**G**,t)** is obtained by taking **t** and universally quantifying over all unconstrained metavariables in **t** that do not appear in the context G.

For example,

$$\mathbf{Clos((x:}\mathrm{t}_{\mathbf{1}}\mathbf{\text{->}}\mathrm{t}_{\mathbf{1}}\mathbf{),}\mathrm{t}_{\mathbf{1}}\mathbf{*}\mathrm{t}_{\mathbf{2}}\mathbf{*}\mathrm{t}_{\mathbf{3}}\mathbf{)}\ =\ "\,\mathrm{a}_{\mathbf{2}}"\,\mathrm{a}_{\mathbf{3}}\mathbf{.}\mathrm{t}_{\mathbf{1}}\mathbf{*}\mathrm{a}_{\mathbf{2}}\mathbf{*}\mathrm{a}_{\mathbf{3}}$$

Practical Type Inference

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

Let-Polymorphism

- Almost equivalent formulation
  - Whenever you see

    let x be e_1 **in** e_2

    typecheck as if the user wrote

    $e_2[x \circledR e_1]$

  - Consider

```
let id be (fn x => <x,x>) 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 $O(2^{2^n})$

- In practice, much better

Complexity Results

- If we add in ordinary recursive functions

  `fun f(x:u`$_1$`):u`$_2$` is e`

  then the problem isn't any harder.

- If we add polymorphic recursive functions

  `Fun f(`a`):u`$_2$` 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.