Type Inference

Prof. Clarkson
Fall 2018

Today’s music: Cool, Calm, and Collected by The Rolling Stones
What is the type of
\texttt{fun x -> (fun y -> x)}?

A. \texttt{'a -> ('b -> 'a)}
B. \texttt{'a * ('b -> 'a)}
C. \texttt{('a -> 'b) -> 'a}
D. \texttt{('a * 'b) -> 'b}
Review

Previously in 3110: Interpreters

Today: Type inference
Hindley-Milner type inference algorithm
Robin Milner

Awarded 1991 Turing Award for “...ML, the first language to include polymorphic type inference and a type-safe exception handling mechanism...”

1934-2010
HM guarantees

- It never infers the wrong types
- It never fails to infer types
- It usually runs in linear time
Simplified HM

Let's omit:

- polymorphic types
- recursive definitions
- making only one pass over program

(more coverage in CS 4110/6110)
Discussion

let \( g \ x = 5 + x \)

What is the type of \( g \)?

...how did you figure it out?

...how could an algorithm compute it?
Algorithm

For each top-level definition, in order:
• decorate each AST node with preliminary type variable
• collect type constraints from each AST node
• use unification to solve constraints and produce a substitution
• use substitution to infer type of definition
AN INFORMAL EXAMPLE
Example

$$\text{let } g \ x = 5 + x$$

Desugared:

$$\text{let } g = \text{fun } x \rightarrow ((+ \ 5) \ x)$$

AST:
Example

\[ \text{let } g = \text{fun } x \rightarrow ((+) \ 5) \ x \]

Step 1: Assign preliminary types to all subexpressions

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>fun x \rightarrow ((+) 5) x</td>
<td></td>
</tr>
</tbody>
</table>
Example

```ml
let g = fun x -> ((+) 5) x
```

Step 1: Assign preliminary types to all subexpressions

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun x -&gt; ((+) 5) x</code></td>
<td></td>
</tr>
<tr>
<td><code>x</code></td>
<td></td>
</tr>
<tr>
<td><code>((+) 5) x</code></td>
<td></td>
</tr>
<tr>
<td><code>(+ 5)</code></td>
<td></td>
</tr>
<tr>
<td><code>(</code> <code>+</code> <code>)</code></td>
<td></td>
</tr>
<tr>
<td><code>5</code></td>
<td></td>
</tr>
<tr>
<td><code>x</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

\texttt{let } g = \texttt{fun x -> ((+ 5) x)

Step 1: Assign preliminary types to all subexpressions

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{fun x -&gt; ((+ 5) x)</td>
<td></td>
</tr>
<tr>
<td>\texttt{x}</td>
<td></td>
</tr>
<tr>
<td>\texttt{((+ 5) x}</td>
<td></td>
</tr>
<tr>
<td>\texttt{(+ 5)}</td>
<td>\texttt{int -&gt; int -&gt; int}</td>
</tr>
<tr>
<td>\texttt{(+)}</td>
<td>\texttt{int}</td>
</tr>
<tr>
<td>\texttt{5}</td>
<td>\texttt{int}</td>
</tr>
<tr>
<td>\texttt{x}</td>
<td></td>
</tr>
</tbody>
</table>
Let \( g = \text{fun } x \rightarrow ((+) 5) x \)  

### Step 1: Assign preliminary types to all subexpressions

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{fun } x \rightarrow ((+) 5) x )</td>
<td>( R )</td>
</tr>
<tr>
<td>( x )</td>
<td>( U )</td>
</tr>
<tr>
<td>((+) 5) ( x )</td>
<td>( S )</td>
</tr>
<tr>
<td>((+)) ( 5 )</td>
<td>( T )</td>
</tr>
<tr>
<td>((+))</td>
<td>( \text{int } \rightarrow \text{int } \rightarrow \text{int} )</td>
</tr>
<tr>
<td>( 5 )</td>
<td>( \text{int} )</td>
</tr>
<tr>
<td>( x )</td>
<td>( V )</td>
</tr>
</tbody>
</table>

\( R,S,T,U,V \) are preliminary type variables used during inference
### Example

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun x -&gt; ((+ 5) x</code></td>
<td>$R$</td>
</tr>
<tr>
<td><code>x</code></td>
<td>$U$</td>
</tr>
<tr>
<td><code>((+ 5) x</code></td>
<td>$S$</td>
</tr>
<tr>
<td><code>(+ 5</code></td>
<td>$T$</td>
</tr>
<tr>
<td><code>(+)</code></td>
<td>int -&gt; int -&gt; int</td>
</tr>
<tr>
<td><code>5</code></td>
<td>int</td>
</tr>
<tr>
<td><code>x</code></td>
<td>$V$</td>
</tr>
</tbody>
</table>

### Diagram

```
fun : R
  x : U
  apply : S
    apply : T
      x : V
        (+) 5 : int
          : int -> int -> int
```
Example

```
let g = fun x -> ((+) 5) x
```

Step 2: Collect constraints
Example

```plaintext
let g = fun x -> ((+) 5) x
```

Step 2: Collect constraints

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fun x -&gt; ((+) 5) x</code></td>
<td>R</td>
</tr>
<tr>
<td><code>x</code></td>
<td>U</td>
</tr>
<tr>
<td><code>((+) 5) x</code></td>
<td>S</td>
</tr>
</tbody>
</table>

Constraint from function:

\[ R = U \rightarrow S \]
Example

```ml
let g = fun x -> ((+) 5) x
```

Step 2: Collect constraints

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>U</td>
</tr>
<tr>
<td>x</td>
<td>V</td>
</tr>
</tbody>
</table>

Constraint from variable usage:

\[ U = V \]
Example

```ml
let g = fun x -> ((+) 5) x
```

Step 2: Collect constraints

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>((+) 5) x</td>
<td>S</td>
</tr>
<tr>
<td>x</td>
<td>V</td>
</tr>
<tr>
<td>(+) 5</td>
<td>T</td>
</tr>
</tbody>
</table>

Constraint from application:

\[ T = V \rightarrow S \]
Example

\texttt{let \ g \ = \ fun \ x \rightarrow \ ((+\ 5) \ x)\n
Step 2: Collect constraints

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+) 5</td>
<td>(T)</td>
</tr>
<tr>
<td>(+)</td>
<td>int (\rightarrow) int (\rightarrow) int</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
</tr>
</tbody>
</table>

Constraint from application:
\texttt{int \(\rightarrow\) int \(\rightarrow\) int = int \(\rightarrow\) T}
Example

```plaintext
let g = fun x -> ((+ 5)) x
```

Step 2: Collect constraints

\[
\begin{align*}
U &= V \\
R &= U \rightarrow S \\
T &= V \rightarrow S \\
\text{int} \rightarrow \text{int} \rightarrow \text{int} &= \text{int} \rightarrow T
\end{align*}
\]
Example

\[ \text{let } g = \text{fun } x -> (((+) 5) x) \]

Step 3: Solve constraints

\[
\begin{align*}
U &= V \\
R &= U->S \\
T &= V->S \\
\text{int} \rightarrow \text{int} \rightarrow \text{int} &= \text{int} \rightarrow T
\end{align*}
\]
Example

let g = fun x -> ((+) 5) x

Step 3: Solve constraints

\[
\begin{align*}
U & = V \\
R & = U \rightarrow S \\
T & = V \rightarrow S \\
\text{int} \rightarrow \text{int} \rightarrow \text{int} & = \text{int} \rightarrow T
\end{align*}
\]
Example

$$\text{let } g = \text{fun } x \rightarrow ((+) 5) x$$

Step 3: Solve constraints

$$R = U \rightarrow S$$
$$T = U \rightarrow S$$
$$\text{int } \rightarrow \text{int } \rightarrow \text{int } = \text{int } \rightarrow T$$
Example

\[
\text{let } g = \text{ fun } x \rightarrow ((+) 5) x
\]

Step 3: Solve constraints

\[
R \quad = \quad U \rightarrow S
\]
\[
T \quad = \quad U \rightarrow S
\]
\[
\text{int } \rightarrow \text{ int } \rightarrow \text{ int} \quad = \quad \text{int } \rightarrow \rightarrow T
\]
Example

```ocaml
let g = fun x -> ((+) 5) x
```

Step 3: Solve constraints

\[ R = U \rightarrow S \]

\[ \text{int} \rightarrow \text{int} \rightarrow \text{int} = \text{int} \rightarrow U \rightarrow S \]
Example

let g = fun x -> ((+) 5) x

Step 3: Solve constraints

\[ R = U \rightarrow S \]

\[ \text{int} \rightarrow \text{int} \rightarrow \text{int} = \text{int} \rightarrow U \rightarrow S \]
Example

```
let g = fun x -> ((+) 5) x
```

Step 3: Solve constraints

\[ R = \text{int} \rightarrow \text{int} \]
Example

```
let g = fun x -> ((+) 5) x
```

Step 3: Solve constraints

\[ R = \text{int} \rightarrow \text{int} \]

Done: type of \( g \) is \( \text{int} \rightarrow \text{int} \)
CONSTRAINT COLLECTION
Algorithm for constraint collection

- **Input:** an expression $e$
  Assume for convenience that every anonymous function in $e$ has a different variable name as its argument

- **Output:** a set of constraints

- **Key idea:** each node in AST generates some constraints based on typing rule for that node
Def and Use

Define two functions that return preliminary type variables assigned to AST node:

- **D**: *definition* of an argument
  - $D(x)$ returns the preliminary type variable assigned to variable $x$

- **U**: *use* of a subexpression
  - $U(e)$ returns the preliminary type variable assigned to subexpression $e$
Def and Use

Example:
• Input: `fun x -> (fun y -> x)`
• Def and Use functions:
  – $D(x) = R$
  – $D(y) = S$
  – $U(fun x -> (fun y -> x)) = T$
  – $U(fun y -> x) = X$
  – $U(x) = Y$
Constraint collection

Collect constraint at each AST node:

- At a variable \( x \):
  \[ U(x) = D(x) \]

- At a function application \( e_1 \ e_2 \):
  \[ U(e_1) = U(e_2) \rightarrow U(e_1 \ e_2) \]

- At an anonymous function \( \text{fun } x \rightarrow e \):
  \[ U(\text{fun } x \rightarrow e) = D(x) \rightarrow U(e) \]

Note how these are essentially the static semantics!
Constraint collection

Continued example:

• Input: fun x -> (fun y -> x)

What constraints would be collected?

Def and Use functions:

- D(x) = R
- D(y) = S
- U(fun x -> (fun y -> x)) = T
- U(fun y -> x) = X
- U(x) = Y

Constraints collected:

• At a variable x:
  - U(x) = D(x)
• At a function application e1 e2:
  - U(e1) = U(e2) -> U(e1 e2)
• At an anonymous function fun x -> e:
  - U(fun x -> e) = D(x) -> U(e)
Constraint collection

Example (continued):

• **Input:** \( \texttt{fun \ x \ -> \ (fun \ y \ -> \ x)} \)
  
• From \( x \), constraint is \( Y = R \)
  
• From \( \texttt{fun} \ y \ -> \ x \), constraint is \( X = S -> Y \)
  
• From \( \texttt{fun} \ x \ -> \ (fun \ y \ -> \ x) \),
  
  constraint is \( T = R -> X \)

Def and Use functions:

- \( D(x) = R \)
- \( D(y) = S \)
- \( U(\texttt{fun} \ x \ -> \ (\texttt{fun} \ y \ -> \ x)) = T \)
- \( U(\texttt{fun} \ y \ -> \ x) = X \)
- \( U(x) = Y \)

Constraints collected:

- At a variable \( x \):
  
  \( U(x) = D(x) \)

- At a function application \( e1 \ e2 \):
  
  \( U(e1) = U(e2) -> U(e1 \ e2) \)

- At an anonymous function \( \texttt{fun} \ x \ -> \ e \):
  
  \( U(\texttt{fun} \ x \ -> \ e) = D(x) -> U(e) \)
Algorithm for constraint solving

- **Input**: a set of constraints
- **Output**: a solution to that set of equations
- **Key idea**: analogous to Gaussian elimination
Unification algorithm

Repeat until constraint set is empty:
• Pick and remove a constraint $t_1 = t_2$ from set
• Reduce based on $t_1$ and $t_2$:
  – Update solution; add new constraint(s) back to set
  – Or, fail: inconsistent equations

Invented by John Alan Robinson (d. 2016), professor at Syracuse University
Reductions

• \( t = t \)
  – no change to solution, no new constraints

• \( t_1 \rightarrow t_2 = t_3 \rightarrow t_4 \)
  – no change to solution, add two new constraints:
    \( t_1 = t_3 \) and \( t_2 = t_4 \)

• \( X = t \) (where \( X \) does not appear in \( t \))
  – substitute \( t \) for \( X \) throughout constraint set
    thus eliminating \( X \) from system of equations
  – append substitution \( X = t \) to solution
FINISH TYPE INFERENCE
Final step

• Setup:
  – Trying to infer type of $e$
  – Preliminary type variable for $e$ was $U(e)$
  – Have a list of substitutions as output of unification

• How to finish:
  – Apply substitutions, in order, to $U(e)$
  – If no preliminary type variables remain, done!
  – Otherwise, expression is polymorphic; not covered here
Upcoming events

• N/A

This is cool, calm, and collected.

THIS IS 3110