Type Inference

Prof. Clarkson
Fall 2016

Today’s music: Cool, Calm, and Collected by The Rolling Stones
Review

Previously in 3110:
• Interpreters: ASTs, evaluation, parsing
• Formal syntax
• Formal semantics
  – Small-step
  – Big-step

Today:
• Type inference
Kinds of typing

• **Static**: type checking done by analysis of program
  – Compiler/interpreter verifies that type errors cannot occur
  – e.g., C, C++, F#, Haskell, Java, OCaml

• **Dynamic**: type checking done by run-time
  – Run-time detects type errors and report them. Usually requires keeping extra tag information for each value in memory.
  – e.g., JavaScript, LISP, Matlab, PHP, Python, Ruby

• Can be a spectrum, e.g., `instanceof` in Java: some checking done at compile time, rest of checking done at run time
Kinds of typing

- **Strong**: type of a value is independent of how it’s used
  - Can’t pass a `string` where an `int` expected, etc.
  - e.g., OCaml, Haskell, Python, Java, Ruby

- **Weak**: type of value is dependent on how it’s used
  - If a `string` is used where an `int` expected, it gets converted automatically or by type cast to an `int`
  - e.g., C, C++, Perl

- Can be a spectrum
  - e.g., Java `+` operator converts objects to strings

- Troll alert: strong vs. weak is debated a lot; probably not helpful to degenerate into such debates
## Typing quadrant

<table>
<thead>
<tr>
<th></th>
<th>Weak</th>
<th>Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>C, C++</td>
<td>OCaml, Java, Haskell</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Perl, Assembly</td>
<td>Ruby, Python, Scheme</td>
</tr>
</tbody>
</table>
Kinds of typing

- **Manifest**: type information supplied in source code
  - e.g., C, C++, Java

- **Implicit**: type information not supplied in source code
  - Implementation 1: Dynamic typing
    - e.g., LISP, Python, Ruby, PHP
  - Implementation 2: **Type inference**
    - e.g., Haskell, OCaml
  - Tradeoff: ease of implementation vs. run-time performance

- Can be a spectrum
  - e.g., no reasonable language requires you to write to provide the type of 5 in `x:int = 5`
Type inference

• Goal is to reconstruct types of expressions based on known types of some symbols that occur in expressions
  – Type checkers have to do some of this anyway
  – Difference between inference and checking is really a matter of degree

• Best known in functional languages
  – Especially useful in managing the types of higher-order functions
  – But starting to appear in mainstream languages, e.g., C++11:
    • `auto x = e;` declares variable `x`, initialized with expression `e`, and type of `x` is automatically inferred
    • `decltype(e)` is a type that means “whatever type `e` has”

• Invented by Robin Milner for SML (though other people also deserve credit; see the notes)
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
Is type inference hard?

• The algorithm used in ML is quite clever yet relatively easy to implement
• Difficulty of doing type inference for any particular language is often hard to determine
• Designing type inference for a particular language can be quite hard; must balance
  – expressivity of type system with
  – possibility of inferring all types without requiring annotations
HM type inference

- Algorithm used in OCaml is called HM
  - Hindley & Milner invented it independently

- Guarantees of HM:
  - **It never makes mistakes.** HM will never infer types that cause a program to fail to type check.
  - **It never fails.** HM will never reject a program that could have been type-checked if programmer had written down all the types.
    - (true of nearly all the language; over time some features have been added for which it's not true; see RWO for examples)
HM type inference

• Determine types of definitions in order
  – Use types of earlier definitions to infer later
  – (which is one reason why you can’t use later definitions in file)

• For each definition:
  – collect constraints on types
  – solve constraints to determine type
Example

\[
\text{let } g \ x = 5 + x
\]

Desugar:

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

```plaintext
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>
</tbody>
</table>
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>\text{fun} \ x \rightarrow (+(5)) \ x</td>
<td></td>
</tr>
<tr>
<td>\text{x}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[(+(5)) \ x | |
| 

\[+(5) | |
| 

\[+(5) | |
| 

\[5 | |
| 

\[x | |}
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</code></td>
<td><code>int</code> -&gt; <code>int</code> -&gt; <code>int</code></td>
</tr>
<tr>
<td><code>(+)</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>5</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td></td>
</tr>
</tbody>
</table>
Example

```plaintext
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>( R )</td>
</tr>
<tr>
<td><code>x</code></td>
<td>( U )</td>
</tr>
<tr>
<td><code>(+) 5</code></td>
<td>( S )</td>
</tr>
<tr>
<td><code>((+) 5) x</code></td>
<td>( T )</td>
</tr>
<tr>
<td><code>int -&gt; int -&gt; int</code></td>
<td></td>
</tr>
<tr>
<td><code>(+)</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>5</code></td>
<td></td>
</tr>
<tr>
<td><code>x</code></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><code>R</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>U</code></td>
</tr>
<tr>
<td><code>((+) 5) x</code></td>
<td><code>S</code></td>
</tr>
<tr>
<td><code>(+) 5</code></td>
<td><code>T</code></td>
</tr>
<tr>
<td><code>(+)</code></td>
<td><code>int -&gt; int -&gt; int</code></td>
</tr>
<tr>
<td><code>5</code></td>
<td><code>int</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>V</code></td>
</tr>
</tbody>
</table>

```
fun x -> ((+) 5) x
```

```
apply : S

x : U

apply : T

(+)

5: int

: int->int->int
```
Question

Did we really need to give \( x \) two different preliminary type variables?

A. Yes
B. No
Question

Did we really need to give \( x \) two different preliminary type variables?

A. Yes
B. No
Example

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

Step 2: Collect constraints
Example

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

Step 2: Collect constraints

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

Constraint from function:
\[ R = U \rightarrow S \]
Example

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

Step 2: Collect constraints

<table>
<thead>
<tr>
<th>Subexpression</th>
<th>Preliminary type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x}</td>
<td>\texttt{U}</td>
</tr>
<tr>
<td>\texttt{x}</td>
<td>\texttt{V}</td>
</tr>
</tbody>
</table>

Constraint from variable usage:

\[ U = V \]
Example

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

Step 2: Collect constraints

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

Constraint from application:
\[ T = V \rightarrow S \]
Example

```haskell
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</td>
<td>T</td>
</tr>
<tr>
<td>(+)</td>
<td>int -&gt; int -&gt; int</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
</tr>
</tbody>
</table>

Constraint from application:

```
int -> int -> int = int -> T
```
Example

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

Step 2: Collect constraints

- \( U = V \)
- \( R = U \rightarrow S \)
- \( T = V \rightarrow S \)
- \( \text{int} \rightarrow \text{int} \rightarrow \text{int} = \text{int} \rightarrow T \)
Example

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

Step 3: Solve constraints

```
U = V

R = U -> S

T = V -> S

int -> int -> int = int -> T
```
Example

```ml
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

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

\texttt{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

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

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

Step 3: Solve constraints

$R = \text{int} \rightarrow \text{int}$
Example

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

Step 3: Solve constraints

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

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

• **Input:** an expression $e$
  – Assume that every anonymous function in $e$ has a different variable name as its argument
  – Easy to ensure that holds, thanks to lexical scope: rename arguments as necessary

• **Output:** a set of constraints
Constraint collection

• Intuition: assign a unique type variable (e.g., R, S, T, ...),
  – one to each argument \( x \) of a function in \( e \)
  – one to every subexpression \( e' \) in \( e \)
  – like how we decorated (aka annotated) AST in example

• Formally: define two functions that return type variables:
  – \( D \): definition of an argument
  – \( U \): use of a subexpression
  – \( D(x) \) returns the type variable assigned to argument \( x \)
  – \( U(e') \) returns the type variable assigned to subexpression \( e' \)
Constraint collection

Example:
• Input: \texttt{fun x -> (fun y -> x)}
• Define two functions for type variables:
  \begin{itemize}
  \item \(D(x) = R\)
  \item \(D(y) = S\)
  \item \(U(\texttt{fun x -> (fun y -> x)}) = T\)
  \item \(U(\texttt{fun y -> x}) = X\)
  \item \(U(x) = Y\)
  \end{itemize}
Constraint collection

Constraints that are collected (intuition):
• For each kind of expression (application, anonymous function, let, etc.), collect a set of equations that must hold for that kind of expression
  – e.g., the type of entire anonymous function must equal type of its argument arrow type of its body
  – which is what we did in example earlier
Constraint collection

Constraints that are collected (formally):

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

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

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

- At a let expression $\text{let} \ x = e_1 \ \text{in} \ e_2$:
  \[ U(\text{let} \ x = e_1 \ \text{in} \ e_2) = U(e_2) \text{ and } D(x) = U(e_1) \]

- etc.

- Unioned with constraints collected at each subexpression

- Note how these are essentially the static semantics!

Return those constraints as output of algorithm
Constraint collection

Example (continued):

• **Input:** fun x -> (fun y -> x)
  - x occurs as subexpression, so generate constraint U(x) = D(x)
    - Already have U(x) = Y and D(x) = R
    - So constraint is Y = R

• **fun y -> ux** occurs as subexpression, so generate constraint U(fun y -> x) = D(y) -> U(x)
  - Already have U(x) = Y, and U(fun y -> x) = X, and D(y) = S
  - So constraint is X = S -> Y

• **fun x -> (fun y -> x)** occurs as subexpression, so generate constraint U(fun x -> (fun y -> x)) = D(x) -> U(fun y -> x)
  - Resulting constraint is T = R -> X
Solving constraints

• After collection, have a set of constraints
  – Really a set of equations

• Need to solve those equations for type of main expression of interest

• *Unification* algorithm [Robinson 1965]
  – roughly like Gaussian elimination to solve system of matrix equations in linear algebra
  – see notes for the algorithm
Upcoming events

• nothing this week

This is cool, calm, and collected.

THIS IS 3110