CS 3110

Type Inference

Prof. Clarkson
Fall 2018

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

Attendance question

What is the type of

`fun x -> (fun y -> x)` ?

A. 'a -> ('b -> 'a)
B. 'a * ('b -> 'a)
C. ('a -> 'b) -> 'a
D. ('a * 'b) -> 'b

Review

Previously in 3110: Interpreters

Today: Type inference

HM

Hindley-Milner type inference algorithm

Robin Milner

1934-2010

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

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

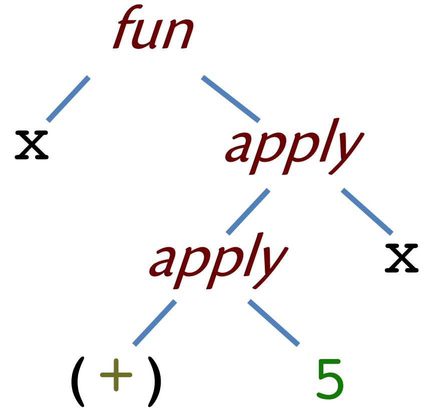
Example

```
let g x = 5 + x
```

Desugared:

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

AST:



Example

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

Step 1: Assign preliminary types to all subexpressions

Subexpression	Preliminary type
`fun x -> ((+) 5) x`	

Example

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

Step 1: Assign preliminary types to all subexpressions

Subexpression	Preliminary type
`fun x -> ((+) 5) x`	
`x`	
`((+) 5) x`	
`(+) 5`	
`(+)`	
`5`	
`x`	

Example

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

Step 1: Assign preliminary types to all subexpressions

Subexpression	Preliminary type
`fun x -> ((+) 5) x`	
`x`	
`((+) 5) x`	
`(+) 5`	
`(+)`	`int -> int -> int`
`5`	`int`
`x`	

Example

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

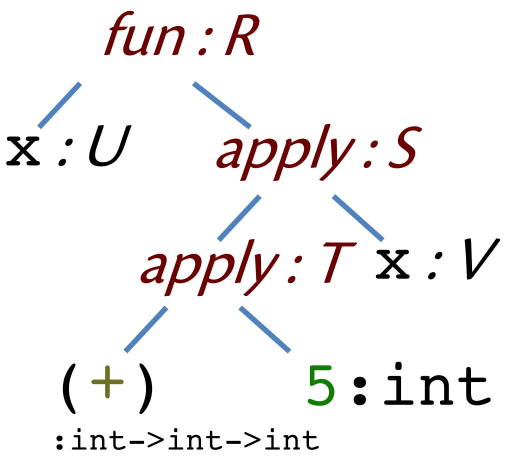
Step 1: Assign preliminary types to all subexpressions

Subexpression	Preliminary type
`fun x -> ((+) 5) x`	*R*
`x`	*U*
`((+) 5) x`	*S*
`(+) 5`	*T*
`(+)`	`int -> int -> int`
`5`	`int`
`x`	*V*

R,S,T,U,V are preliminary type variables used during inference

Example

Subexpression	Preliminary type
`fun x -> ((+) 5) x`	*R*
`x`	*U*
`((+) 5) x`	*S*
`(+) 5`	*T*
`(+)`	`int -> int -> int`
`5`	`int`
`x`	*V*



Example

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

Step 2: Collect constraints

Example

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

Step 2: Collect constraints

Subexpression	Preliminary type
`fun x -> ((+) 5) x`	R
`x`	U
`((+) 5) x`	S

Constraint from function:

$$R = U \texttt{->} S$$

Example

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

Step 2: Collect constraints

Subexpression	Preliminary type
`x`	U
`x`	V

Constraint from variable usage:

$$U = V$$

Example

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

Step 2: Collect constraints

Subexpression	Preliminary type
`((+) 5) x`	S
`x`	V
`(+) 5`	T

Constraint from application:

$T = V$ **->** S

Example

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

Step 2: Collect constraints

Subexpression	Preliminary type
`(+) 5`	T
`(+)`	`int -> int -> int`
`5`	`int`

Constraint from application:

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

Example

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

Step 2: Collect constraints

$$\begin{array}{rcl} U & = & V \\ R & = & U \texttt{->} S \\ T & = & V \texttt{->} S \\ \texttt{int -> int -> int} & = & \texttt{int ->}\ T \end{array}$$

Example

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

Step 3: Solve constraints

$$
\begin{aligned}
U &= V \\
R &= U \texttt{->} S \\
T &= V \texttt{->} S \\
\texttt{int -> int -> int} &= \texttt{int ->}\ T
\end{aligned}
$$

Example

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

Step 3: Solve constraints

$$
\begin{array}{rcl}
U & = & V \\
R & = & U \texttt{->} S \\
T & = & V \texttt{->} S \\
\texttt{int -> int -> int} & = & \texttt{int ->}\ T
\end{array}
$$

Example

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

Step 3: Solve constraints

$$R = U \texttt{->} S$$

$$T = U \texttt{->} S$$

$$\texttt{int -> int -> int} = \texttt{int ->} T$$

Example

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

Step 3: Solve constraints

$$R = U \texttt{->} S$$

$$T = U \texttt{->} S$$

$$\texttt{int -> int -> int} = \texttt{int ->}\ T$$

Example

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

Step 3: Solve constraints

$$R = U \texttt{->} S$$

$$\texttt{int -> int -> int} = \texttt{int ->}\ U \texttt{->} S$$

Example

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

Step 3: Solve constraints

$$R = U \texttt{->} S$$

$$\texttt{int -> int -> int} = \texttt{int ->} U \texttt{->} S$$

Example

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

Step 3: Solve constraints

R = **`int -> int`**

Example

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

Step 3: Solve constraints

R = `int -> int`

Done: type of **`g`** is **`int -> 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 **`e1 e2`**:

  U(**`e1`**) = U(**`e2`**) **`->`** U(**`e1 e2`**)

- At an anonymous function **`fun x -> e`**:

  U(**`fun x -> e`**) = D(**`x`**) **`->`** 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: **`fun x -> (fun y -> x)`**
- From **`x`**, constraint is $Y = R$
- From **`fun y -> x`**, constraint is $X = S \mathbin{\texttt{->}} Y$
- From **`fun x -> (fun y -> x)`**, constraint is $T = R \mathbin{\texttt{->}} X$

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*

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 SOLVING

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 t1=t2 from set
- Reduce based on t1 and t2:
  - 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
- `t1 -> t2 = t3 -> t4`
  - no change to solution, add two new constraints: `t1 = t3` and `t2 = t4`
- `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