CSCI 334:
Principles of Programming Languages

Lecture 9: Type Inference

Instructor: Dan Barowy

Williams

Announcements

- Lab 5 due date is Sunday at 11:59pm. But if you can't get it done, just let me know when you can.
- "Course Changes"

Midterm Exam

- After you return. Date TBD.

Announcements

- We will navigate the chaos together.
  - Be proactive; we understand and we want to help
  - The situation is unreasonable, we are not
- Remember, nothing about this is fair, but nothing about this is anyone's fault. We have to be good to each other and to ourselves.
  - There is more than CS334 in our lives.

Study tip

Grades are important, but they are **not the most important** thing in life.

Study tip

Just do your best.

Remember: **labs are practice**. Practice makes perfect.

Remember: **you can resubmit labs**.

Remember: **you can resubmit the midterm**.

Outline

1. Type inference
2. Q&A

This course is going to change

See the “**Course Changes**” section of the course webpage

Your crazy awesome TAs are actually available this weekend! We will update the schedule soon…

Colin: 11am-8pm on Saturday

Type checking & type inference

Finally—cool things enabled by the lambda calculus!

Type checking

(or, "how does my compiler know that my expression is wrong?")

```
let f(x:int) : int = "hello" + x
```

```
let f(x:int) : int = "hello" + x;;
-------------------------------^

stdin(1,32): error FS0001: The type 'int' does not
match the type 'string'
```

A note about "curried" expressions

let f(a: int, b: int, c: char) : float = …

f is **int -> int -> char -> float**

let f(a: int)(b: int)(c: char) : float = …

let f a b c = …

f = λa.λb.λc.(…)

Type checking

step 1: convert into lambda form

```
let f(x:int) : int = "hello" + x
f = λx."hello " + x             convert into λ expression
f = λx.((+ "hello ") x)         assume + = λx.λy.((x + y))
```

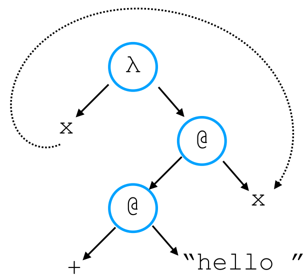
The purpose of this step is to make all of the parts of an expression clear

Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

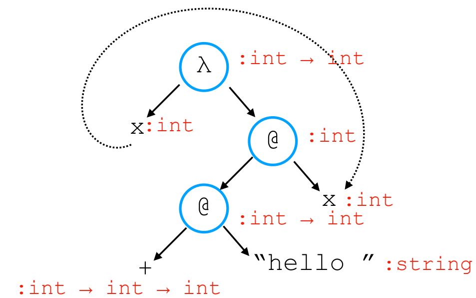
`f` has form `λx.((EE)E)`



Type checking

step 3: label parse tree with types

read ":" as "has type"



Type checking

step 4: check that types are used consistently

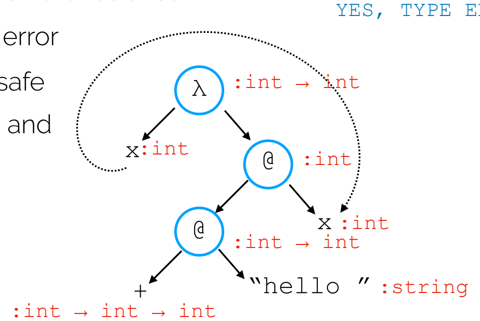
1. Start at the leaves
2. Do type mismatches arise?

   Yes = type error

   No = type safe
3. if yes, stop and report first mismatch

`int → int → int @ string`

`YES, TYPE ERROR`



Type inference

notice that we had a typed expression

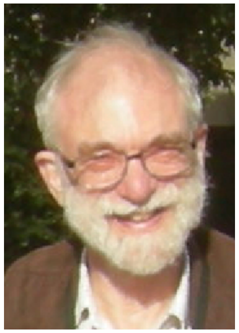
`let f(x:int) : int = "hello " + x`

what if, instead, we had

`let f(x) = "hello " + x`

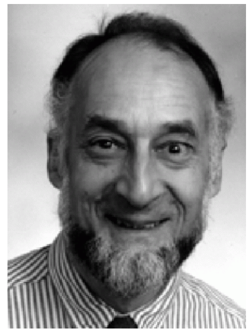
?

Hinley-Milner algorithm



J. Roger Hindley

- Hindley and Milner invented algorithm independently.
- Infers types from known data types and operations used.
- Depends on a step called "unification".
- I will demonstrate informal method for unification; works for small examples



Robin Milner

Hinley-Milner algorithm

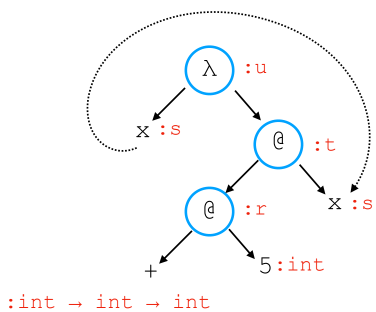
Has three main phases:

1. Assign known types to each subexpression
2. Generate type constraints based on rules of **λ** calculus:
   a. Abstraction constraints
   b. Application constraints
3. Solve type constraints using unification.

Type inference

step 1: label parse tree with known/unknown types

```
let f(x) = 5 + x
f = λx.((+ 5) x)
```



Type inference

it is often helpful to have types in tabular form

subexpression	type
+	int → int → int
5	int
(+5)	r
x	s
(+5)x	t
λx.((+ 5) x)	u

Type inference

step 2: generate type constraints using λ calculus

```
E ::= x        variable
    | λx.E     abstraction
    | EE       function application
```

Abstraction rule: If the type of x is a and the type of E is b, and the type of λx.E is c, then the constraint is c = a → b.

Application rule: If the type of E_1 is a and the type of E_2 is b, and the type of E_1E_2 is c, then the constraint is a = b → c.

Type inference

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r	int → int → int = int → r
x	s	n/a
(+5)x	t	r = s → t
λx.((+ 5) x)	u	u = s → t

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r	int → int → int = int → r
x	s	n/a
(+5)x	t	r = s → t
λx.((+ 5) x)	u	u = s → t

Start with the topmost unknown. What do we know about r?

int → int → int = int → r
r = int → int

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int → int → int = int → r
x	s	n/a
(+5)x	t	r = s → t
λx.((+ 5) x)	u	u = s → t

Eliminate r from the constraint.

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int→int→int = int→int→int
x	s	n/a
(+5) x	t	int → int = s → t
λx.((+ 5) x)	u	u = s → t

Eliminate r from the constraint.

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int→int→int = int→int→int
x	s	n/a
(+5) x	t	int → int = s → t
λx.((+ 5) x)	u	u = s → t

What do we know about s and t?

```
int → int = s → t
s = int
t = int
```

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int→int→int = int→int→int
x	s = int	n/a
(+5) x	t = int	int → int = s → t
λx.((+ 5) x)	u	u = s → t

Eliminate s and t from constraint.

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int→int→int = int→int→int
x	s = int	n/a
(+5) x	t = int	int → int = int → int
λx.((+ 5) x)	u	u = int → int

What do we know about u?

```
u = int → int
```

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int→int→int = int→int→int
x	s = int	n/a
(+5)x	t = int	int → int = int → int
λx.((+ 5) x)	u = int → int	u = int → int

Eliminate u from constraint.

Type inference

step 3: unify

subexpression	type	constraint
+	int → int → int	n/a
5	int	n/a
(+5)	r = int → int	int→int→int = int→int→int
x	s = int	n/a
(+5)x	t = int	int → int = int → int
λx.((+ 5) x)	u = int → int	int → int = int → int

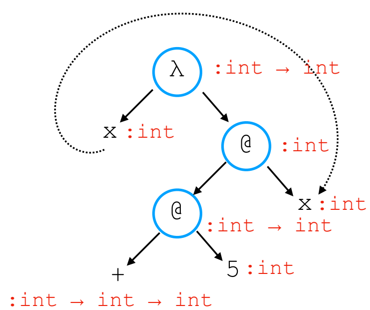
Done when there is nothing left to do.

Sometimes unknown types remain.

This means that the function is polymorphic. We'll talk more later!

Completed type inference

```
let f x = 5 + x
f = λx.((+ 5) x)
```



Wrap up

Stay Safe and Healthy

- It's not going to be easy, but we will work together to make the course a success
  - We want to support you! BUT
  - It is up to you to let us know when things aren't going as planned
- We know what it is like to be stuck and not understand something…
  - Do not accept defeat alone. We are a team.

Stay Safe and Healthy

- If things come up in your life outside of class, let us know
  - We will find ways to accommodate your situation
- If things come up in class, let us know
  - We will find ways to resolve issues on our end

Stay Safe and Healthy

- Find routines and practices that work for you
  - Want a study partner from CS334?
    - Reach out
  - Hard time concentrating?
    - "Work Uniform", mynoise.net, daily planner
  - Get the big picture, but not the details?
    - Teach a friend!
  - Easily distracted?
    - draw pictures on paper, take physical notes, get away from a computer

Remote Access

VPN

SSH

Q&A

Recap & Next Class

Today we covered:

Type inference

Next class:

TBD— enjoy your break!