CSCI 334:
Principles of Programming Languages

Lecture 15: Type Inference

Instructor: Dan Barowy

Williams

Announcements

Announcements

No class Thursday

Announcements

No class Thursday

I am out all week; email me if you need anything, but expect delays in my responses

Announcements

No class Thursday

I am out all week; email me if you need anything, but expect delays in my responses

HW7 solutions

Announcements

No class Thursday

I am out all week; email me if you need anything, but expect delays in my responses

HW7 solutions

Exam resubmissions: no later than tomorrow

Topics

Topics

Discuss: Graham / Gabriel readings

Topics

Discuss: Graham / Gabriel readings

Review alpha normal form

Topics

Discuss: Graham / Gabriel readings

Review alpha normal form

The importance of technical interviews

Topics

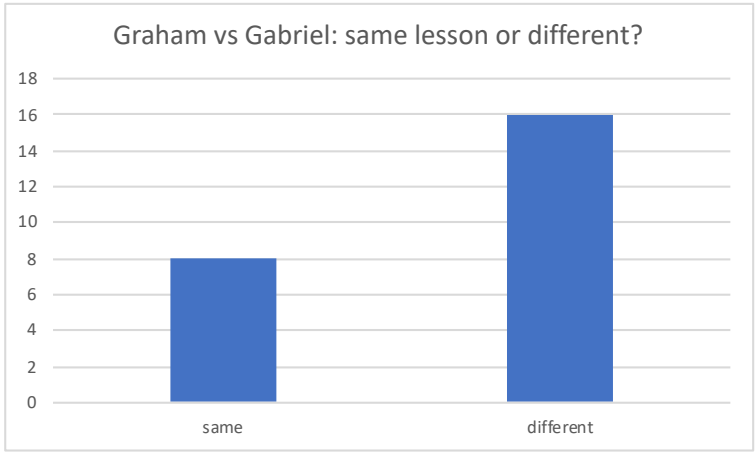
Discuss: Graham / Gabriel readings

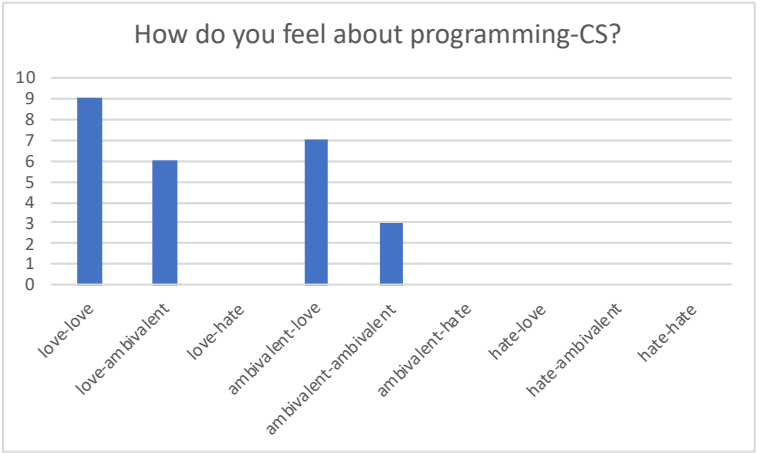
Review alpha normal form

The importance of technical interviews

Type checking and type inference

Discussion





Alpha Normal Form

$(\lambda x.\lambda x.x)(\lambda x.y)$

$(\lambda x.\lambda x.x)\,(\lambda x.y)$

$\Downarrow \alpha$

$(\lambda x.\lambda x.x)\,(\lambda x.y)$

$\Downarrow \alpha$

$(\lambda x.\lambda a.a)\,(\lambda b.y)$

Goal: alpha-normal form

Goal: alpha-normal form

1. No bound variable uses the same name as any free variable.

Goal: alpha-normal form

1. No bound variable uses the same name as any free variable.
2. No bound variable uses the same name as any other bound variable.

Goal: alpha-normal form

1. No bound variable uses the same name as any free variable.
2. No bound variable uses the same name as any other bound variable.

In other words, all variable names are unique.

Parts

```
e: Expr             An expression
b: Set<char>        Variable bindings
r: Map<char,char>   Renamings
```

```
alphanorm(e: Expr)(b: Set<char>)(r: Map<char,char>)
                  : Expr*Set<char>
```

What is passed in; returned

```
alphanorm(e: Expr)(b: Set<char>)(r: Map<char,char>)
                  : Expr*Set<char>
```

What is passed in; returned

```
alphanorm(e: Expr)(b: Set<char>)(r: Map<char,char>)
                  : Expr*Set<char>
```

Note that we want the set of bindings to persist, therefore it is both *passed in* and *out*.

What is passed in; returned

```
alphanorm(e: Expr)(b: Set<char>)(r: Map<char,char>)
                  : Expr*Set<char>
```

Note that we want the set of bindings to persist, therefore it is both *passed in* and *out*.

But the set of renamings is *scoped*: it is only passed in.

Algorithm

Algorithm

Var(v):

Algorithm

Var(v):
if there is a renaming rule, rename and return renamed Var;

Algorithm

Var(v):
if there is a renaming rule, rename and return renamed Var;
otherwise, return original Var

Algorithm

Var(v):
if there is a renaming rule, rename and return renamed Var;
otherwise, return original Var

App(e1, e2):

Algorithm

Var(v):
if there is a renaming rule, rename and return renamed Var;
otherwise, return original Var

App(e1, e2):
α-norm e1 & e2 and return App(e1, e2)

Algorithm

Var(v):
if there is a renaming rule, rename and return renamed Var;
otherwise, return original Var

App(e1, e2):
α-norm e1 & e2 and return App(e1, e2)

Abs(v,e):

Algorithm

Var(v):
if there is a renaming rule, rename and return renamed Var;
otherwise, return original Var

App(e1, e2):
α-norm e1 & e2 and return App(e1, e2)

Abs(v,e):
if v is already bound, add renaming rule, α-norm e, then return Abs(v', e');

Algorithm

Var(v):
if there is a renaming rule, rename and return renamed Var;
otherwise, return original Var

App(e1, e2):
α-norm e1 & e2 and return App(e1, e2)

Abs(v,e):
if v is already bound, add renaming rule, α-norm e, then return Abs(v', e');
otherwise, return α-norm e and return Abs(v, e')

Technical Interviews

Read: interview excerpt with Peter Norvig.

Technical Interviews

"One of the interesting things we found, when trying to predict how well somebody we've hired is going to perform when we evaluate them a year or two later, is one of the best indicators of success within the company was getting the worst possible score on one of your interviews."

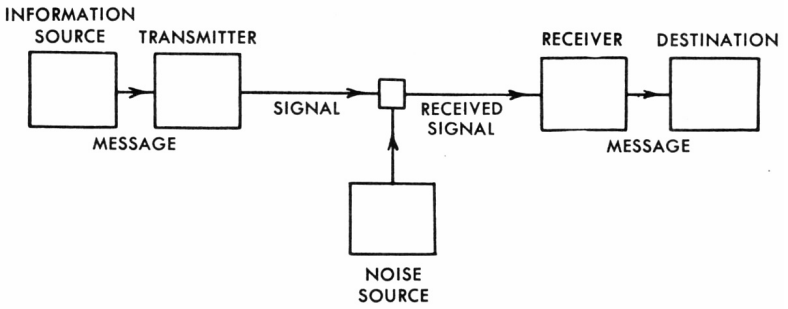


Fig. 1. — Schematic diagram of a general communication system.

34 The Mathematical Theory of Communication

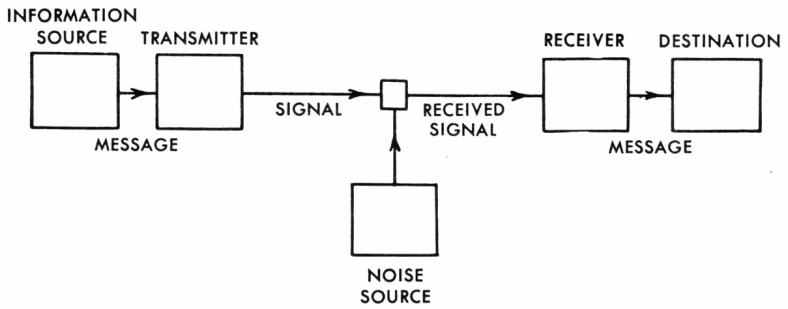


Fig. 1. — Schematic diagram of a general communication system.

34 The Mathematical Theory of Communication

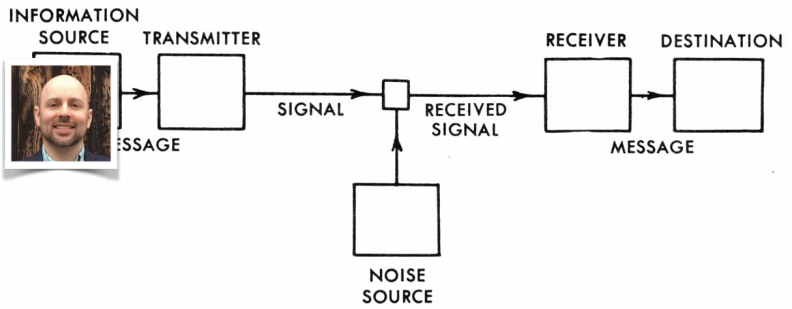


Fig. 1. — Schematic diagram of a general communication system.

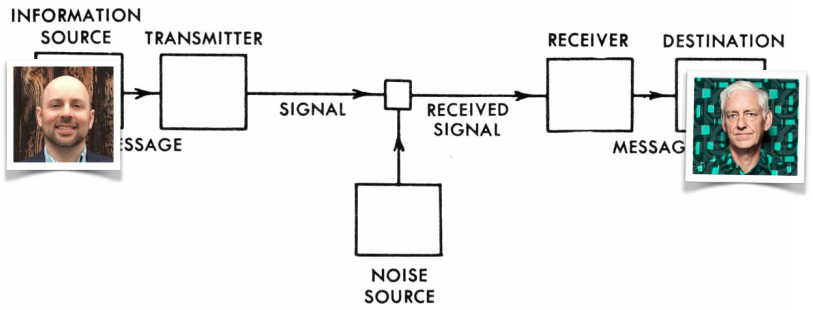


Fig. 1. — Schematic diagram of a general communication system.

34 The Mathematical Theory of Communication

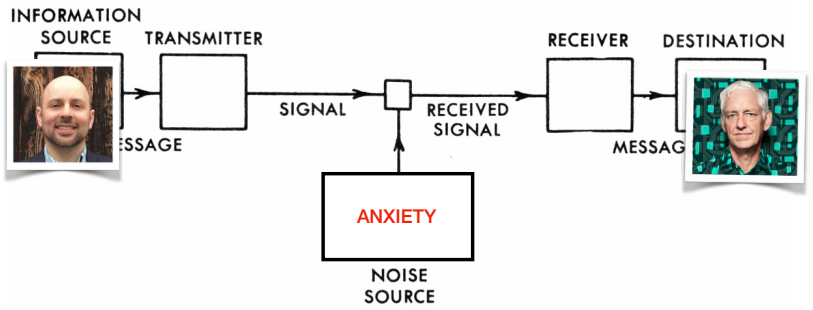


Fig. 1. — Schematic diagram of a general communication system.

34 The Mathematical Theory of Communication

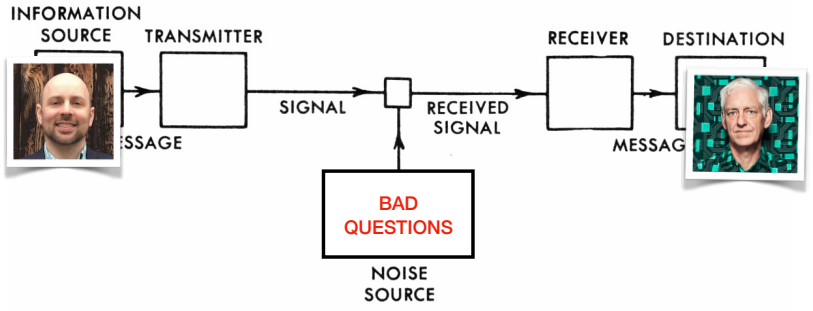


Fig. 1. — Schematic diagram of a general communication system.

34 The Mathematical Theory of Communication

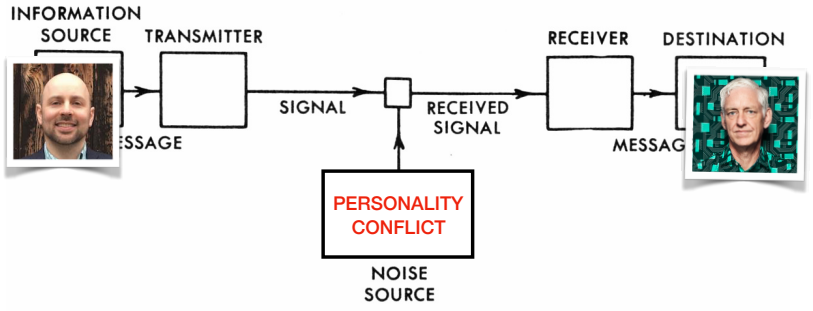


Fig. 1. — Schematic diagram of a general communication system.

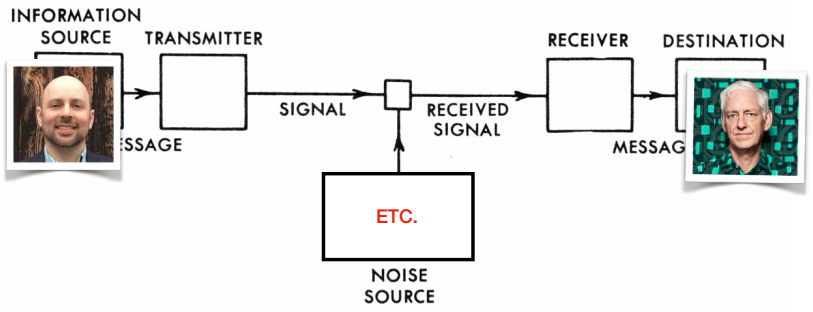
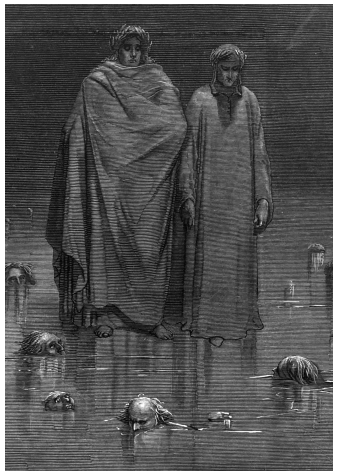


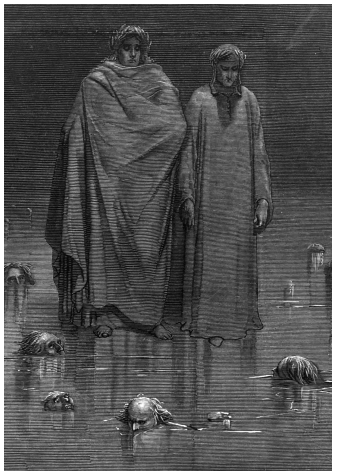
Fig. 1. — Schematic diagram of a general communication system.

Abandon all hope ye who enter here?



Abandon all hope ye who enter here?

"It's more you want to get a feeling for how this person thinks and how they work together, so do they know the basic ideas? Can they say, **"Well, in order to solve this, I need to know A, B, and C,"** and they start putting it together. And I think you can demonstrate that while still failing on a puzzle. You can say, **"Well, here's how I attack this puzzle. Well, I first think about this. Then I do that. Then I do that, but geez, here's this part I don't quite understand."** For some people that little part clicks and for some it doesn't. **And you can do fine if it doesn't click as long as you've demonstrated the basic competency and fluency in how you think about it."**



This class

This class

Programming languages capture the ***essential*** parts of problem-solving on a machine.

This class

Programming languages capture the ***essential*** parts of problem-solving on a machine.

This class is about learning how to ***talk*** about a problem.

This class

Programming languages capture the ***essential*** parts of problem-solving on a machine.

This class is about learning how to ***talk*** about a problem.

Dirty secret: most other CS classes are about that too.

Technical Interviews

Takeaways

Technical Interviews

Takeaways

- Technical interviews are a noisy process at best.

Technical Interviews

Takeaways

- Technical interviews are a noisy process at best.
- Google—and most other tech firms—know this.

Technical Interviews

Takeaways

- Technical interviews are a noisy process at best.
- Google—and most other tech firms—know this.
- Do your best to eliminate the noise.

Technical Interviews

Takeaways

- Technical interviews are a noisy process at best.
- Google—and most other tech firms—know this.
- Do your best to eliminate the noise.
- If you "fail," remember to *learn from your mistakes* and *try again*.

Technical Interviews

Takeaways

- Technical interviews are a noisy process at best.
- Google—and most other tech firms—know this.
- Do your best to eliminate the noise.
- If you "fail," remember to *learn from your mistakes* and *try again*.
- "Just do good work."

Mental Technique #5

"Just do good work."

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).
- You have **limited control** over what other people think about you.

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).
- You have **limited control** over what other people think about you.
- You have a **lot of control** about what you think about yourself.

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).
- You have **limited control** over what other people think about you.
- You have a **lot of control** about what you think about yourself.
- **Set your own standards** and make an effort to meet or **exceed them**.

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).
- You have **limited control** over what other people think about you.
- You have a **lot of control** about what you think about yourself.
- **Set your own standards** and make an effort to meet or **exceed them**.
- ***Care*** about your work.

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).
- You have **limited control** over what other people think about you.
- You have a **lot of control** about what you think about yourself.
- **Set your own standards** and make an effort to meet or **exceed them**.
- ***Care*** about your work.
- This is a "**gumption-building**" activity.

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).
- You have **limited control** over what other people think about you.
- You have a **lot of control** about what you think about yourself.
- **Set your own standards** and make an effort to meet or **exceed them**.
- ***Care*** about your work.
- This is a "**gumption-building**" activity.
- It makes nearly all work seem "**worth it**" because you're not just doing the work; "**the real cycle you're working on is a cycle called 'yourself.'**"

Mental Technique #5

"Just do good work."

- Advice from a favorite professor of mine (David Jensen).
- You have **limited control** over what other people think about you.
- You have a **lot of control** about what you think about yourself.
- **Set your own standards** and make an effort to meet or **exceed them**.
- ***Care*** about your work.
- This is a "**gumption-building**" activity.
- It makes nearly all work seem "**worth it**" because you're not just doing the work; "**the real cycle you're working on is a cycle called 'yourself.'**"

- People will **inevitably notice** your positive, can-do attitude.

Type checking & type inference

Type checking & type inference

Finally—cool things enabled by the lambda calculus!

Type checking

(or, “how does ML know that my expression is wrong?”)

```
let f(x:int) : int = “hello” + x
```

Type checking

(or, “how does ML 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'
```

Type checking

step 1: convert into lambda form

Type checking

step 1: convert into lambda form

```
let f(x:int) : int = “hello” + x
```

Type checking

step 1: convert into lambda form

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

convert into λ expression

Type checking

step 1: convert into lambda form

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

convert into λ expression

assume + = λx.λy.[[x + y]]

Type checking

step 1: convert into lambda form

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

convert into λ expression

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

Type checking

step 2: generate parse tree

```
f = λx.((+ "hello ") x)
```

Type checking

step 2: generate parse tree

```
f = λx.((+ "hello ") x)
```

`f` has form `λx.((EE)E)`

Type checking

step 2: generate parse tree

```
f = λx.((+ "hello ") x)
```

`f` has form `λx.((EE)E)`

Type checking

step 2: generate parse tree

```
f = λx.((+ "hello ") x)
```

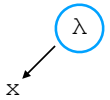
`f` has form `λx.((EE)E)`

Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

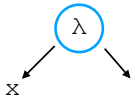


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

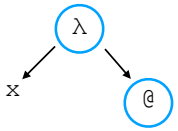


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

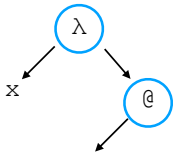


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

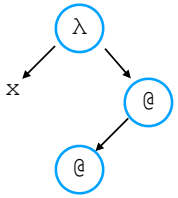


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

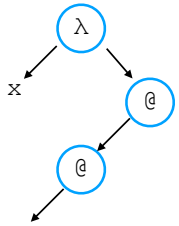


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

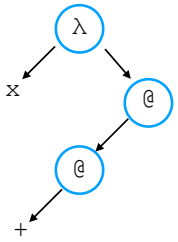


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

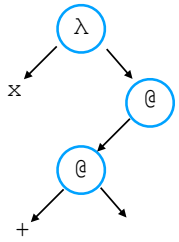


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

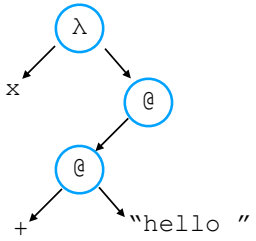


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

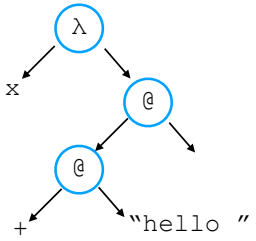


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

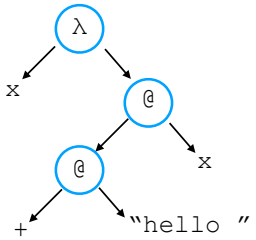


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

`f` has form `λx.((EE)E)`

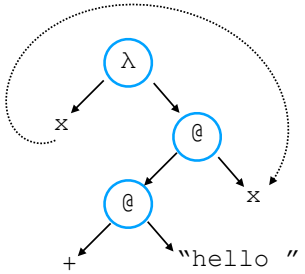


Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

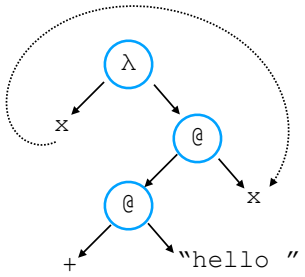
`f` has form `λx.((EE)E)`



Type checking

step 3: label parse tree with types

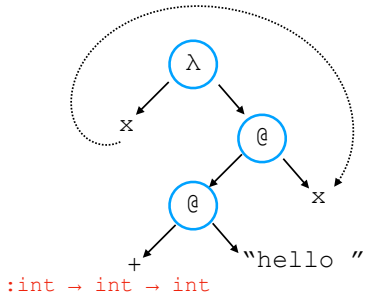
read "**:**" as "has type"



Type checking

step 3: label parse tree with types

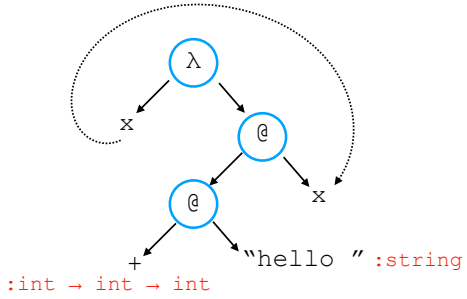
read "**:**" as "has type"



Type checking

step 3: label parse tree with types

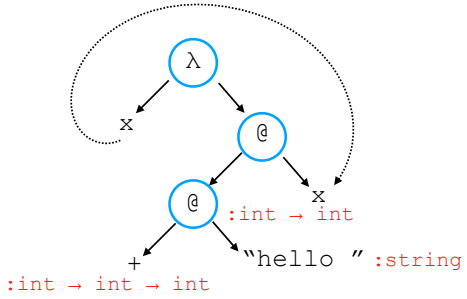
read "**:**" as "has type"



Type checking

step 3: label parse tree with types

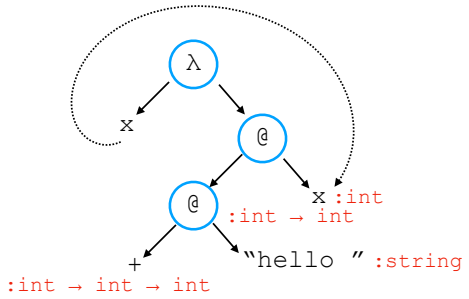
read "**:**" as "has type"



Type checking

step 3: label parse tree with types

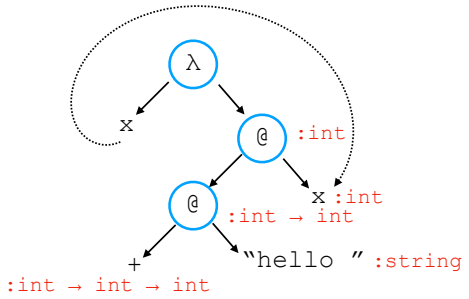
read "**:**" as "has type"



Type checking

step 3: label parse tree with types

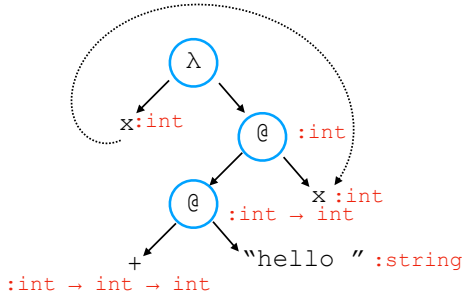
read "**:**" as "has type"



Type checking

step 3: label parse tree with types

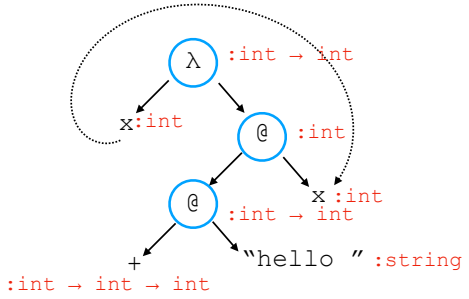
read "**:**" as "has type"



Type checking

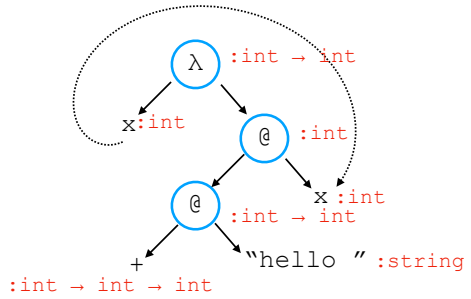
step 3: label parse tree with types

read "**:**" as "has type"



Type checking

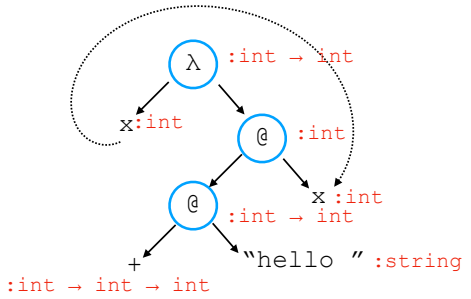
step 4: check that types are used consistently



Type checking

step 4: check that types are used consistently

1. Start at the leaves



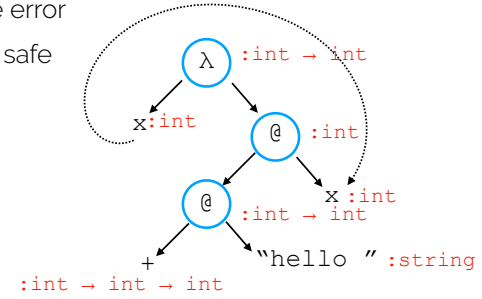
Type checking

step 4: check that types are used consistently

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

   Yes = type error

   No = type safe



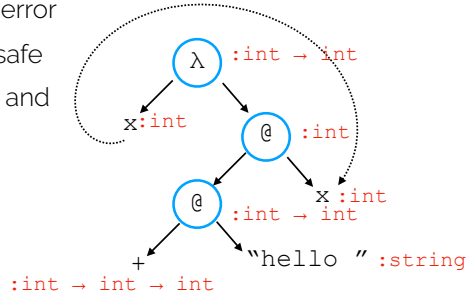
Type checking

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

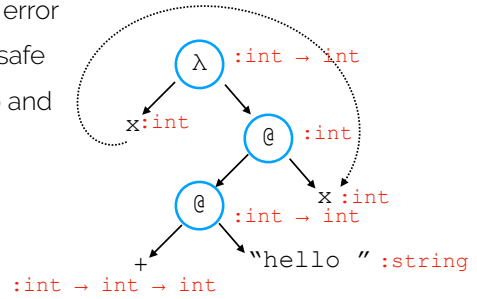


Type checking

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`



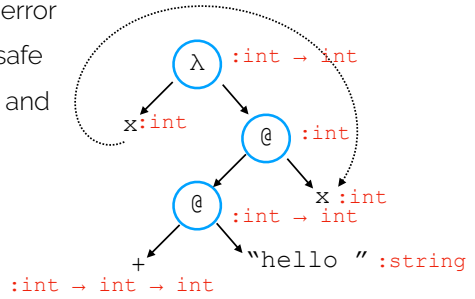
Type checking

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
```

Type inference

notice that we had a typed expression

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

what if, instead, we had

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
```

Type inference

notice that we had a typed expression

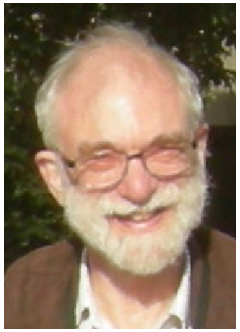
```
let f(x:int) : int = "hello " + x
```

what if, instead, we had

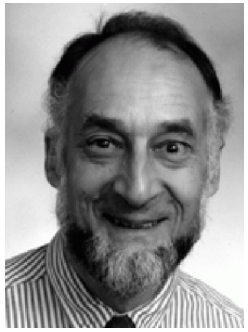
```
let f x = "hello " + x
```

?

Hinley-Milner algorithm

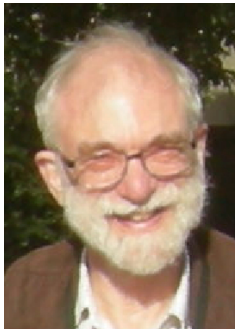


J. Roger Hindley



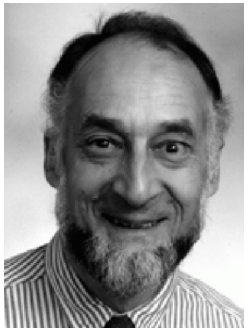
Robin Milner

Hinley-Milner algorithm



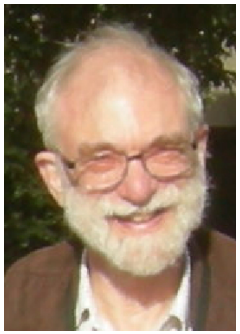
J. Roger Hindley

- Hindley and Milner invented algorithm independently.



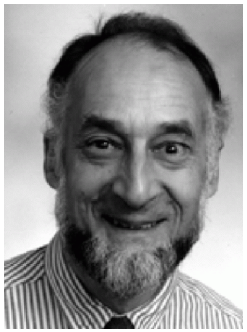
Robin Milner

Hinley-Milner algorithm



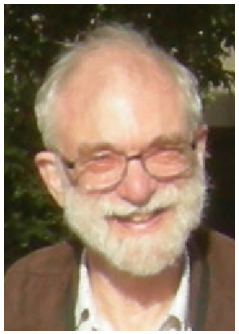
J. Roger Hindley

- Hindley and Milner invented algorithm independently.
- Infers types from known data types and operations used.



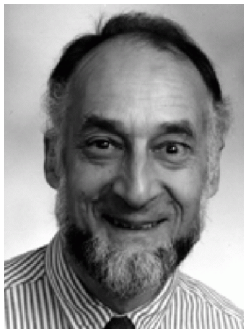
Robin Milner

Hinley-Milner algorithm



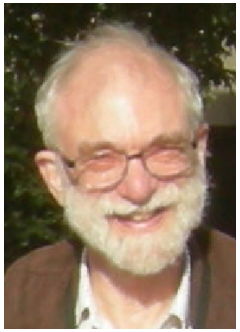
J. Roger Hindley

- Hindley and Milner invented algorithm independently.
- Infers types from known data types and operations used.
- Depends on a step called “unification”.



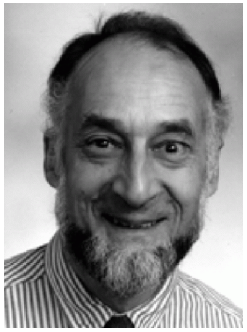
Robin Milner

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:

Hinley-Milner algorithm

Has three main phases:

1. Assign known types to each subexpression

Hinley-Milner algorithm

Has three main phases:

1. Assign known types to each subexpression
2. Generate type constraints based on rules of **λ** calculus:

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

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

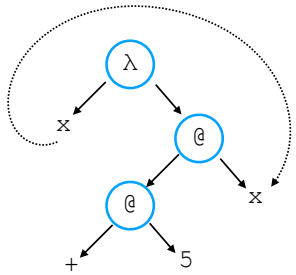
Hinley-Milner algorithm

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

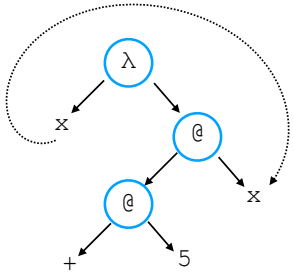
step 1: label parse tree with known/unknown types



Type inference

step 1: label parse tree with known/unknown types

```
let f x = 5 + x
```

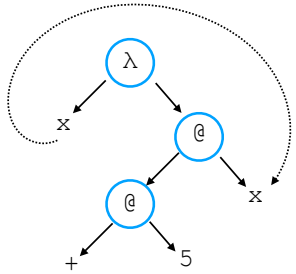


Type inference

step 1: label parse tree with known/unknown types

```
let f x = 5 + x
```

I am using the example from book so that you can follow along at home!

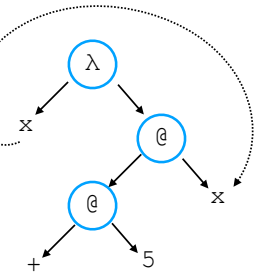


Type inference

step 1: label parse tree with known/unknown types

I am using the example from book so that you can follow along at home!

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

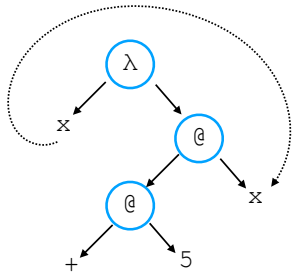


Type inference

step 1: label parse tree with known/unknown types

I am using the example from book so that you can follow along at home!

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

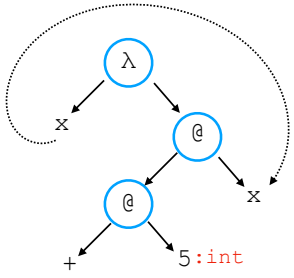
:int → int → int

Type inference

step 1: label parse tree with known/unknown types

I am using the example from book so that you can follow along at home!

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

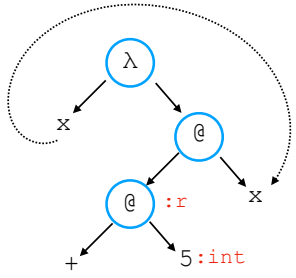
:int → int → int

Type inference

step 1: label parse tree with known/unknown types

I am using the example from book so that you can follow along at home!

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

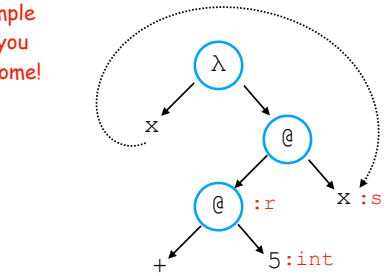
:int → int → int

Type inference

step 1: label parse tree with known/unknown types

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

I am using the example from book so that you can follow along at home!



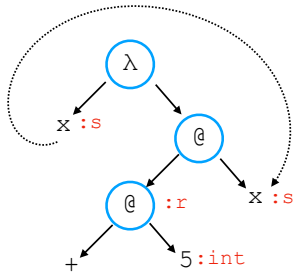
:int → int → int

Type inference

step 1: label parse tree with known/unknown types

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

I am using the example from book so that you can follow along at home!



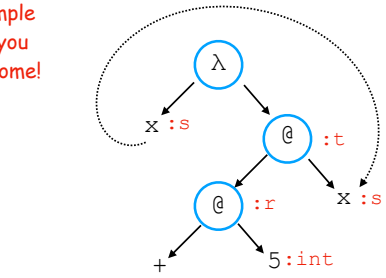
:int → int → int

Type inference

step 1: label parse tree with known/unknown types

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

I am using the example from book so that you can follow along at home!



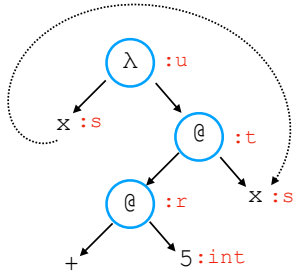
:int → int → int

Type inference

step 1: label parse tree with known/unknown types

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

I am using the example from book so that you can follow along at home!



:int → int → int

Type inference

it is often helpful to have types in tabular form

subexpression	type

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	`int → int → int`

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	`int → int → int`
5	

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	int → int → int
5	int

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	int → int → int
5	int
(+5)	

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	int → int → int
5	int
(+5)	r

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	int → int → int
5	int
(+5)	r
x	

Type inference

it is often helpful to have types in tabular form

subexpression	type
+	int → int → int
5	int
(+5)	r
x	s

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	

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

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)	

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
```

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`.

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

Type inference

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

Type inference

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

Type inference

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

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	
λx.((+ 5) x)	u	

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	

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

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?

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

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

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

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 → 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 → 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

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?

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

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

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

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

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

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 = 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 = 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

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?

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

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

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	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

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.

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.

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. More next class!

Completed type inference

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