CSCI 334:
Principles of Programming Languages

Lecture 20: Type inference

Instructor: Dan Barowy

Williams

Topics

Type inference

Your to-dos

1. Project checkpoint #2, **due Sunday 12/4**.
2. Last quiz, **due Wednesday 12/7**.

Type checking & type inference

Cool things made possible by
the lambda calculus!

type inference

Not everybody loves this part of PL.

I hope that you can appreciate the absence of magic.

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 refresher on "curried" expressions

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

`f` is `a:int * b:int * c:char -> float`

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

`f` is `int -> int -> 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 $+ = \lambda x.\lambda y.(x + y)$

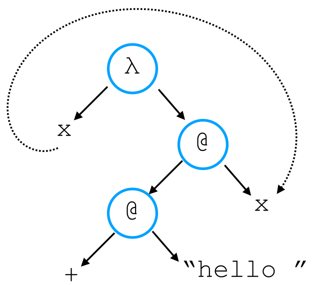
The purpose of this step is to make all of the parts of an expression clear

Type checking

step 2: generate parse tree

`f = λx.((+ "hello ") x)`

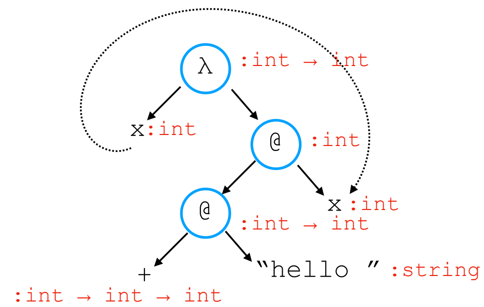
`f` has form `λx.((EE)E)`



Type checking

step 3: label parse tree with types

read ":" as "has type"



Type checking

step 4: check that types are used consistently

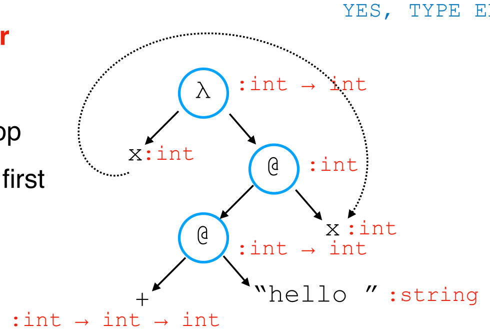
1. Start at the leaves
2. Do type mismatches arise?

   Yes = **error**

   No = **ok**
3. if **error**, stop and report first mismatch

`int → int → int @ string`

`YES, TYPE ERROR`



Type inference

notice that we had a typed expression

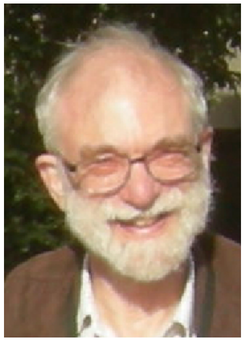
`let f(x:int) : int = "hello " + x`

what if, instead, we had

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

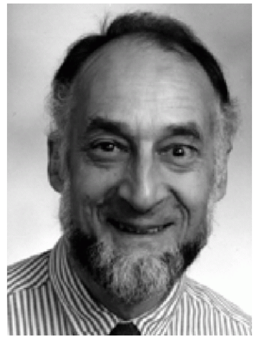
?

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

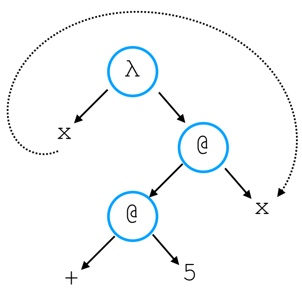
Has three main phases:

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: convert to lambda AST

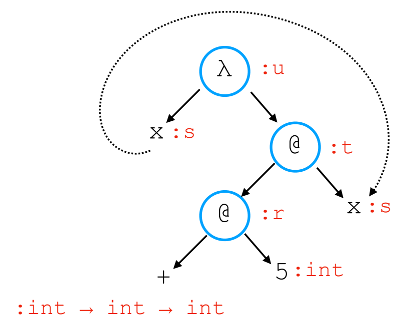
```
let f(x) = 5 + x
f = λx.((+ 5) x)
```



Type inference

step 2: 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 3: generate constraints

```
<expr> ::= <var>            variable
        | λ<var>.<expr>     abstraction
        | <expr><expr>      function application
```

Three rules, each corresponding to a kind of λ expression.

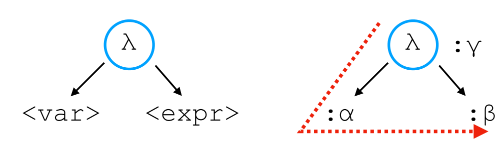
3.1. <var> constraint

No constraint.

3.2. abstraction constraint

λ<var>.<expr>

"left triangle rule"

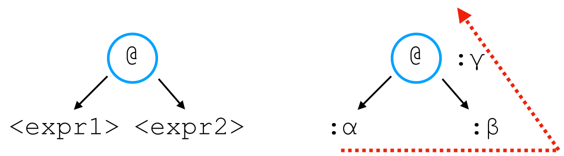


<u>Constraint:</u> If the type of <var> is α and the type of <expr> is β, and the type of λ is γ, then the constraint is $\gamma = \alpha \rightarrow \beta$.

3.3. application constraint

<expr><expr>

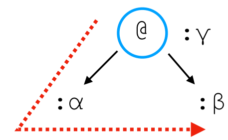
“right triangle rule”



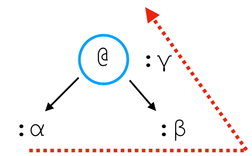
Constraint: If the type of <expr1> is α and the type of <expr2> is β, and the type of @ is γ, then the constraint is α = β → γ.

constraints summary

Abstraction: If the type of <var> is a and the type of <expr> is b, and the type of λ is c, then the constraint is c = a → b.



Application: If the type of <expr1> is a and the type of <expr2> is b, and the type of @ 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

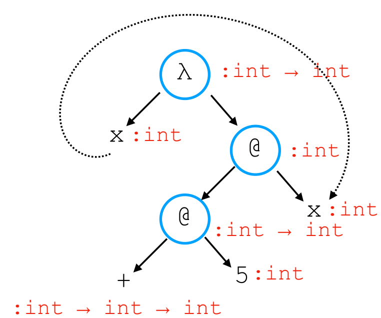
Done when there is nothing left to do.

Sometimes unknown types remain.

An unknown type means that the function is polymorphic.

Completed type inference

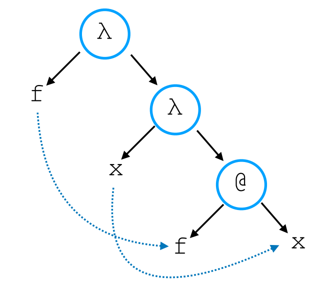
```
let f x = 5 + x
f = λx.((+ 5) x)
```



Let's try one together

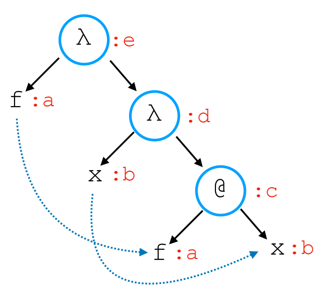
1. convert to λ expression

```
let apply f x = f x
apply = λf.λx.f x
```



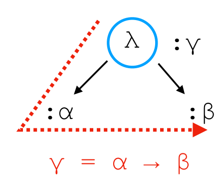
2. label with type variables

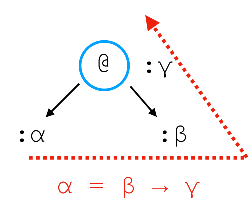
```
let apply f x = f x
apply = λf.λx.f x
```



3. generate constraints

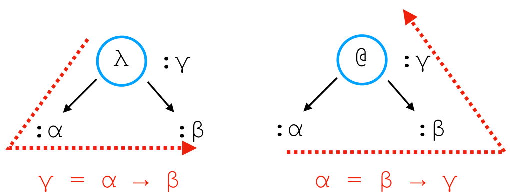
subexpression	type	constraint
f	a	n/a
x	b	n/a
f x	c	a = b → c
λx.f x	d	d = b → c
λf.λx.f x	e	e = a → d





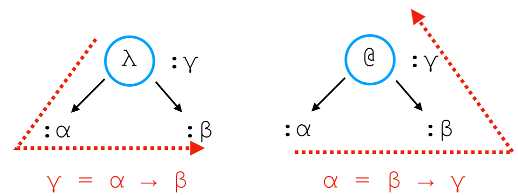
4. unify

subexpression	type	constraint
f	a	n/a
x	b	n/a
f x	c	a = b → c
λx.f x	d	d = b → c
λf.λx.f x	e	e = a → d



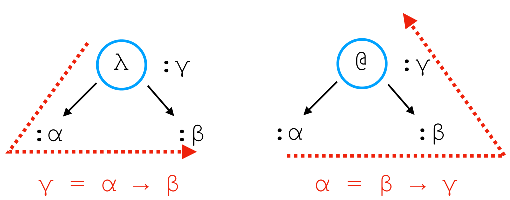
4. unify

subexpression	type	constraint
f	b → c	n/a
x	b	n/a
f x	c	
λx.f x	d	d = b → c
λf.λx.f x	e	e = b → c → d



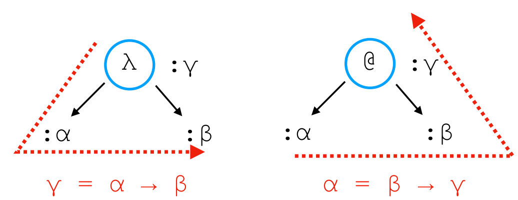
4. unify

subexpression	type	constraint
f	b → c	n/a
x	b	n/a
f x	c	
λx.f x	b → c	
λf.λx.f x	e	e = b → c → b → c



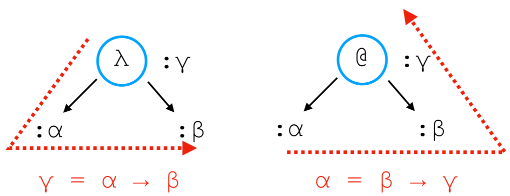
4. unify

subexpression	type	constraint
f	b → c	n/a
x	b	n/a
f x	c	
λx.f x	b → c	
λf.λx.f x	b → c → b → c	



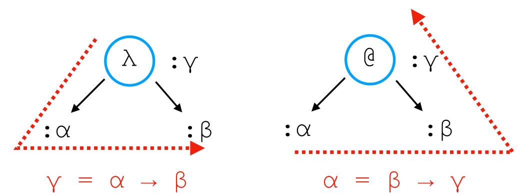
5. rename variables in alpha order

subexpression	type	constraint
f	'a → c	n/a
x	'a	n/a
f x	c	
λx.f x	'a → c	
λf.λx.f x	'a → c → 'a → c	



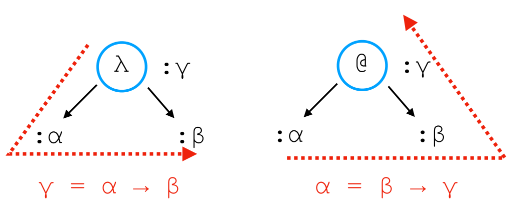
5. rename variables in alpha order

subexpression	type	constraint
f	'a → 'b	n/a
x	'a	n/a
f x	'b	
λx.f x	'a → 'b	
λf.λx.f x	'a → 'b → 'a → 'b	



5. rename variables in alpha order

subexpression	type	constraint
f	'a → 'b	n/a
x	'a	n/a
f x	'b	
λx.f x	'a → 'b	
λf.λx.f x	'a → 'b → 'a → 'b	



Is this the right answer?

'a → 'b → 'a → 'b

```
> let apply f x = f x;;
val apply : f:('a -> 'b) -> x:'a -> 'b
```

Lookin' good!

activity

```
let f g x = g (g x)
```

Recap & Next Class

Today:

Type inference

Next class:

Object Oriented Programming