CSCI 334:
Principles of Programming Languages

Lecture 17: Polymorphic Type Inference

Instructor: Dan Barowy

Williams

Announcements

Announcements

No “assignment”.

Announcements

No “assignment”.

A short reading (Cardelli) for Monday.

Announcements

No “assignment”.

A short reading (Cardelli) for Monday.

Otherwise, work on your project.

Announcements

No “assignment”.

A short reading (Cardelli) for Monday.

Otherwise, work on your project.

Next week: “programming in the large”

Announcements

No “assignment”.

A short reading (Cardelli) for Monday.

Otherwise, work on your project.

Next week: “programming in the large”

`http://catb.org/jargon/html/F/foo.html`

Project Timeline

Project Timeline

Project checkin: mostly complete by Nov 29

Project Timeline

Project checkin: mostly complete by Nov 29

Project done: complete by Dec 6

Project Timeline

Project checkin: mostly complete by Nov 29

Project done: complete by Dec 6

Project presentation (5-10 minutes): Dec 11

Topics

Topics

Type inference

Topics

Type inference

Project Q & A

Type Inference

type inference

```
let apply f x = f x
```

type inference

```
let apply f x = f x
```

1. convert to λ expression

type inference

```
let apply f x = f x
```

1. convert to λ expression
2. label with type variables

type inference

```
let apply f x = f x
```

1. convert to λ expression
2. label with type variables
3. generate constraints

type inference

```
let apply f x = f x
```

1. convert to λ expression
2. label with type variables
3. generate constraints
4. unify

type inference

```
let apply f x = f x
```

1. convert to λ expression
2. label with type variables
3. generate constraints
4. unify
5. rename variables

type inference

type inference

Not everybody loves this part of PL.

type inference

Not everybody loves this part of PL.

I hope that you can appreciate the absence of magic.

1. convert to λ expression

1. convert to λ expression

```
let apply f x = f x
```

1. convert to λ expression

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

1. convert to λ expression

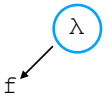
```
let apply f x = f x
apply = λf.λx.f x
```

1. convert to λ expression

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

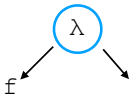
1. convert to λ expression

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



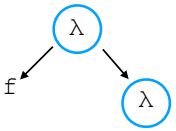
1. convert to λ expression

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



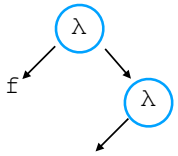
1. convert to λ expression

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



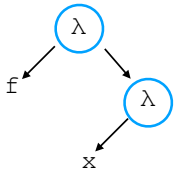
1. convert to λ expression

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



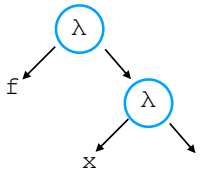
1. convert to λ expression

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



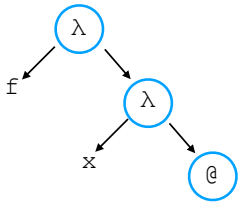
1. convert to λ expression

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



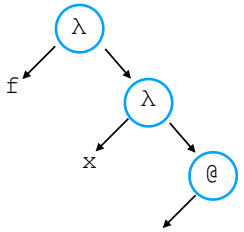
1. convert to λ expression

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



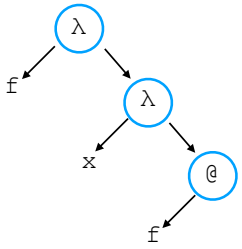
1. convert to λ expression

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



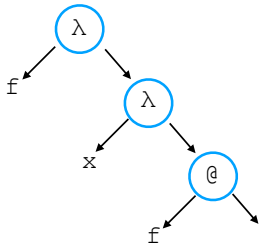
1. convert to λ expression

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



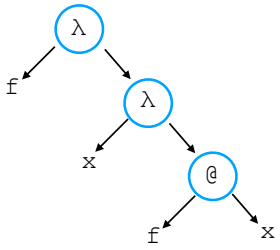
1. convert to λ expression

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



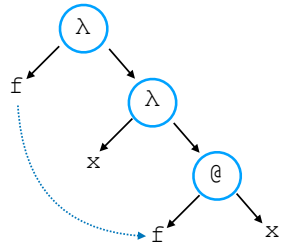
1. convert to λ expression

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



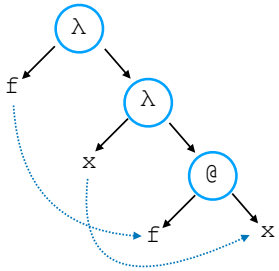
1. convert to λ expression

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



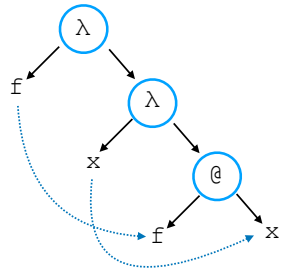
1. convert to λ expression

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



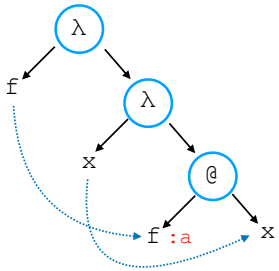
2. label with type variables

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



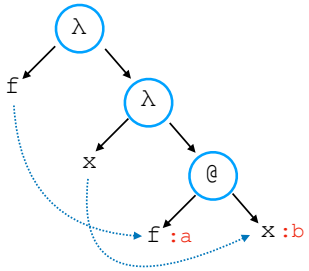
2. label with type variables

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



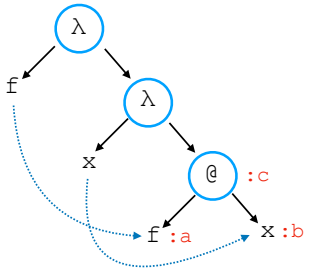
2. label with type variables

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



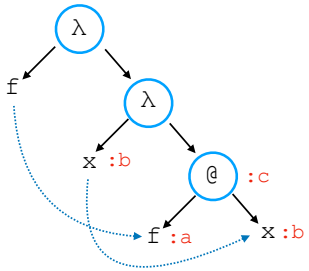
2. label with type variables

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



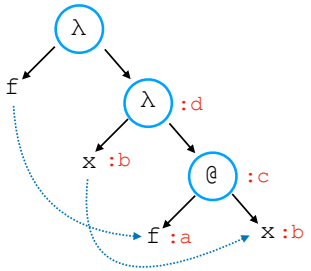
2. label with type variables

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



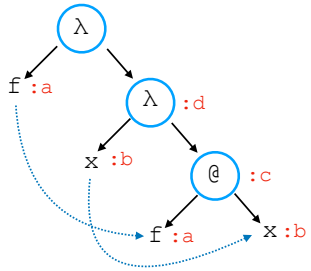
2. label with type variables

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



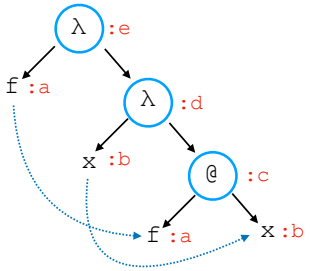
2. label with type variables

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



2. label with type variables

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



3. generate constraints

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

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

3.1. <var> constraint

No constraint.

3.2. abstraction constraint

`λ<var>.<expr>`

3.2. abstraction constraint

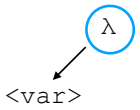
`λ<var>.<expr>`

3.2. abstraction constraint

λ<var>.<expr>

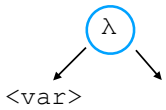
3.2. abstraction constraint

λ<var>.<expr>



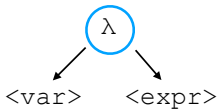
3.2. abstraction constraint

λ<var>.<expr>



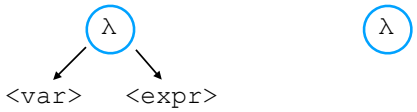
3.2. abstraction constraint

λ<var>.<expr>



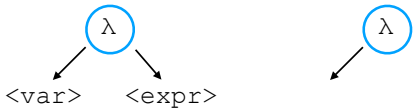
3.2. abstraction constraint

λ<var>.<expr>



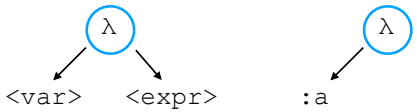
3.2. abstraction constraint

λ<var>.<expr>



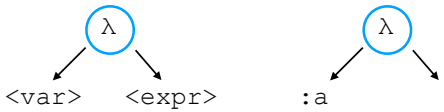
3.2. abstraction constraint

λ<var>.<expr>



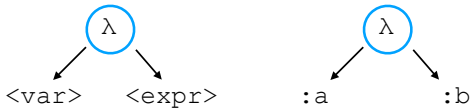
3.2. abstraction constraint

λ<var>.<expr>



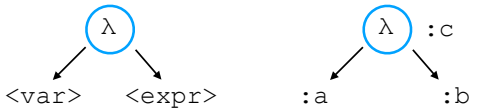
3.2. abstraction constraint

λ<var>.<expr>



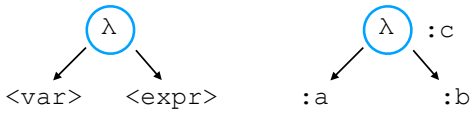
3.2. abstraction constraint

λ<var>.<expr>



3.2. abstraction constraint

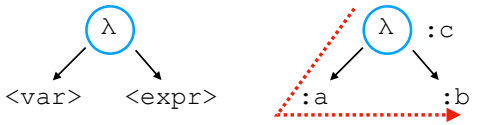
λ<var>.<expr>



Constraint: 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.

3.2. abstraction constraint

λ<var>.<expr>

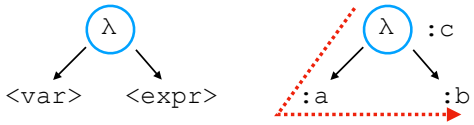


Constraint: If the type of <var> is a and the type of <expr> is b, and the type of λ is c, then the constraint is c = a → b.

3.2. abstraction constraint

λ<var>.<expr>

"left triangle rule"



Constraint: 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.

3.3. application constraint

<expr><expr>

3.3. application constraint

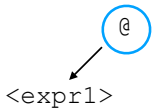
<expr><expr>

3.3. application constraint

<expr><expr>

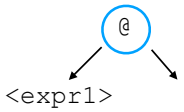
3.3. application constraint

<expr><expr>



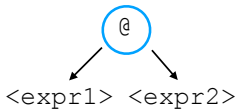
3.3. application constraint

<expr><expr>



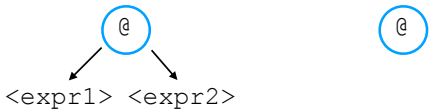
3.3. application constraint

<expr><expr>



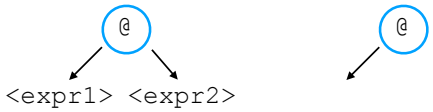
3.3. application constraint

<expr><expr>



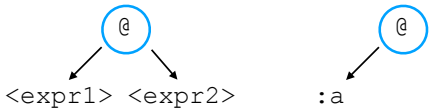
3.3. application constraint

<expr><expr>



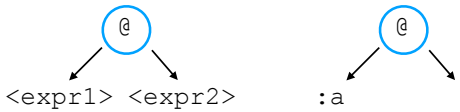
3.3. application constraint

<expr><expr>



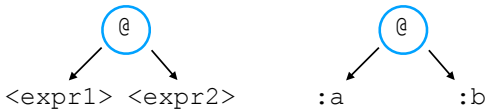
3.3. application constraint

<expr><expr>



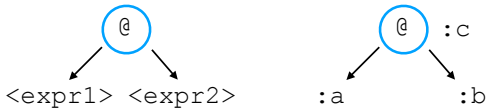
3.3. application constraint

<expr><expr>



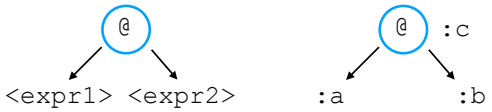
3.3. application constraint

<expr><expr>



3.3. application constraint

<expr><expr>



Constraint: 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.

3.3. application constraint

<expr><expr>

Constraint: 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.

3.3. application constraint

<expr><expr>

"right triangle rule"

Constraint: 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.

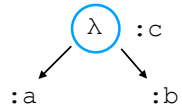
3. constraints summary

3. 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`.

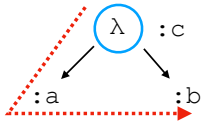
3. constraints summary

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`.



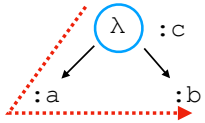
3. constraints summary

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`.



3. constraints summary

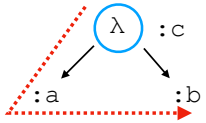
Abstraction: If the type of <var> is a and the type of <expr> is b, and the type of λ is c, then the constraint is c = a → b.



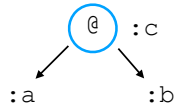
Application: If the type of <expr1> is a and the type of <expr2> is b, and the type of @ is c, then the constraint is a = b → c.

3. 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.

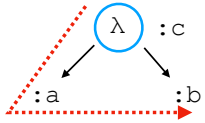


Application: If the type of <expr1> is a and the type of <expr2> is b, and the type of @ is c, then the constraint is a = b → c.

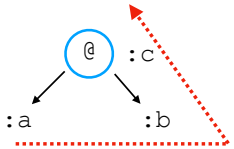


3. constraints summary

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.

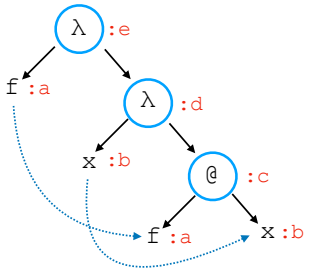


Application: If the type of <expr1> is a and the type of <expr2> is b, and the type of @ is c, then the constraint is a = b → c.

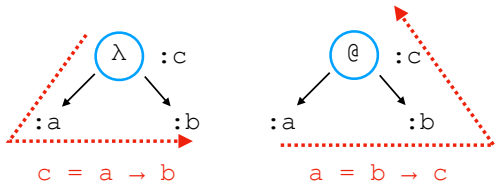


2. label with type variables

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

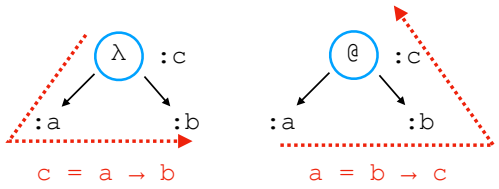


3. generate constraints



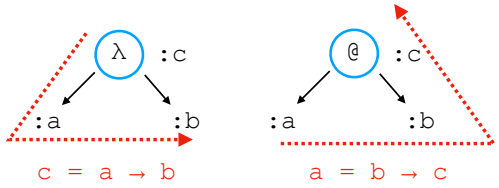
3. generate constraints

subexpression	type	constraint



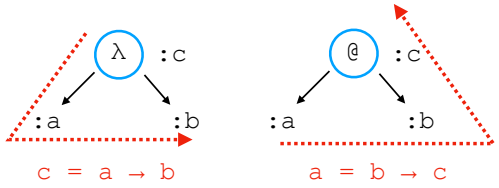
3. generate constraints

subexpression	type	constraint
`f`		



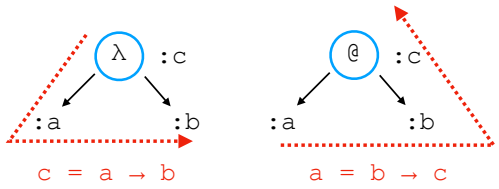
3. generate constraints

subexpression	type	constraint
`f`		
`x`		



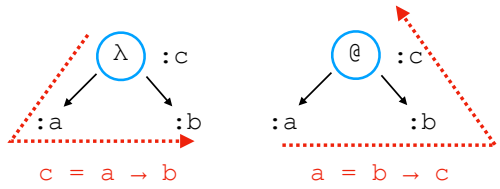
3. generate constraints

subexpression	type	constraint
f		
x		
f x		



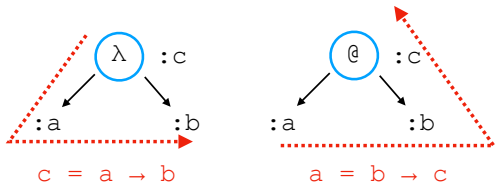
3. generate constraints

subexpression	type	constraint
f		
x		
f x		
λx.f x		



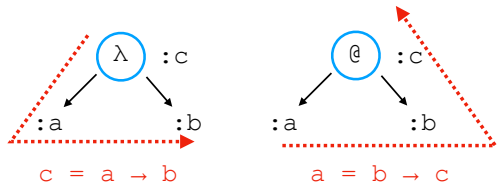
3. generate constraints

subexpression	type	constraint
f		
x		
f x		
λx.f x		
λf.λx.f x		



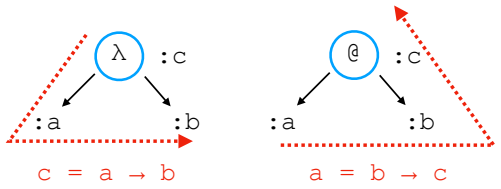
3. generate constraints

subexpression	type	constraint
f	a	
x		
f x		
λx.f x		
λf.λx.f x		



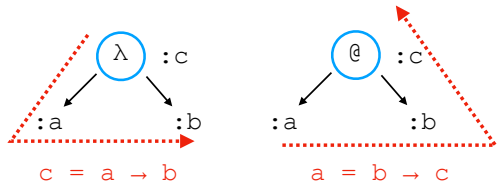
3. generate constraints

subexpression	type	constraint
f	a	
x	b	
f x		
λx.f x		
λf.λx.f x		



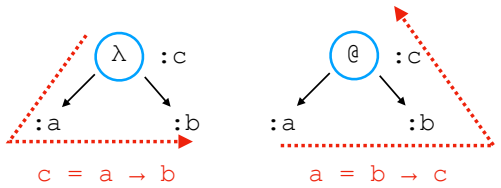
3. generate constraints

subexpression	type	constraint
f	a	
x	b	
f x	c	
λx.f x		
λf.λx.f x		



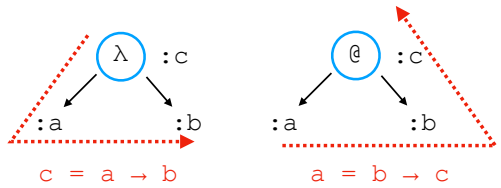
3. generate constraints

subexpression	type	constraint
f	a	
x	b	
f x	c	
λx.f x	d	
λf.λx.f x		



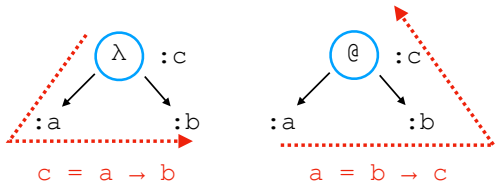
3. generate constraints

subexpression	type	constraint
f	a	
x	b	
f x	c	
λx.f x	d	
λf.λx.f x	e	



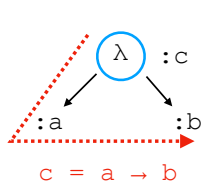
3. generate constraints

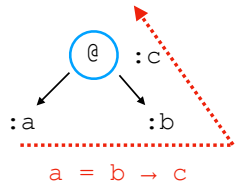
subexpression	type	constraint
f	a	n/a
x	b	
f x	c	
λx.f x	d	
λf.λx.f x	e	



3. generate constraints

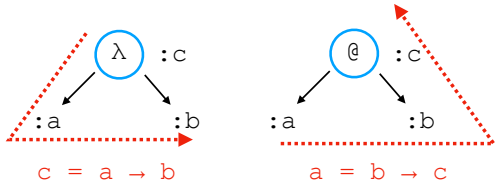
subexpression	type	constraint
f	a	n/a
x	b	n/a
f x	c	
λx.f x	d	
λf.λx.f x	e	





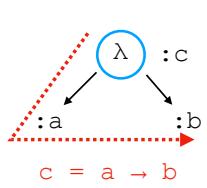
3. generate constraints

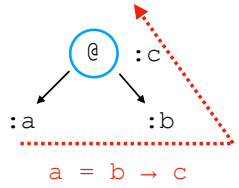
subexpression	type	constraint
f	a	n/a
x	b	n/a
f x	c	a = b → c
λx.f x	d	
λf.λx.f x	e	



3. generate constraints

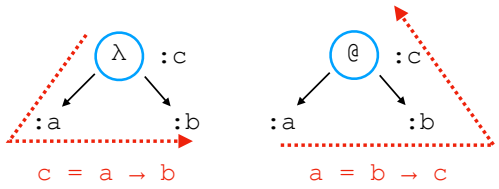
subexpression	type	constraint
f	a	n/a
x	b	n/a
f x	c	a = b → c
λx.f x	d	d = b → c
λf.λx.f x	e	





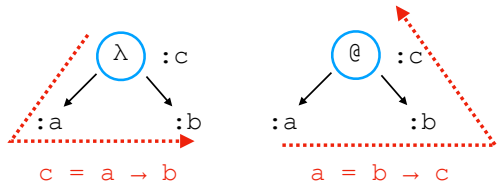
3. generate constraints

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



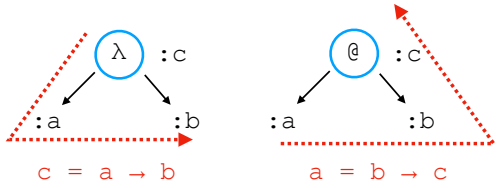
4. unify

subexpression	type	constraint
f	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



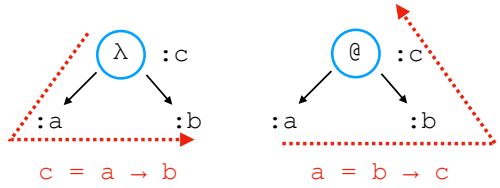
4. unify

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



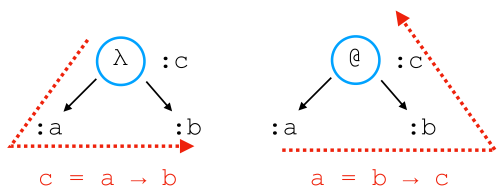
4. unify

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	



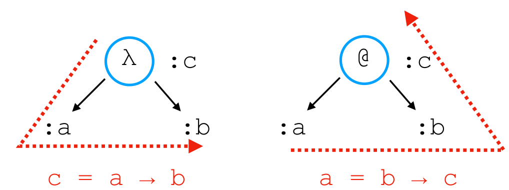
5. rename variables in alpha order

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	



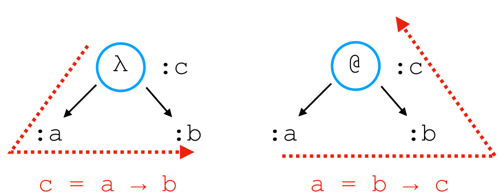
5. rename variables in alpha order

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	



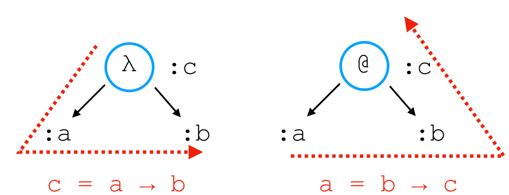
5. rename variables in alpha order

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	



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`

Is this the right answer?

`'a → 'b → 'a → 'b`

Is this the right answer?

`'a → 'b → 'a → 'b`

```
> let apply f x = f x;;
```

Is this the right answer?

`'a → 'b → 'a → 'b`

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

Project Q & A

Next week

Next week

"programming in the large"

Next week

"programming in the large"

object-oriented programming