A Fresh View at Type Inference

Yin Wang
October 19, 2012

Location: LH 101
Time: 4:15 PM

What is λ-calculus?

What is λ-calculus?

Three kinds of terms of λ-calculus

What is λ-calculus?

Three kinds of terms of λ-calculus

variable	x
function (lambda)	λx.E
function application	E_1E_2

What are they?

variable

x y z

function

λx. E

function application

E1E2

What are they?

wires

x y z

function

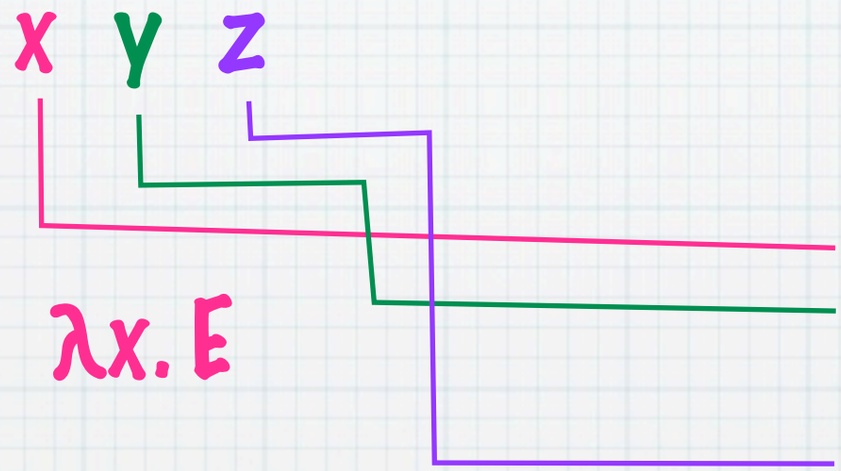
λx. E

function application

E1E2

What are they?

wires



function

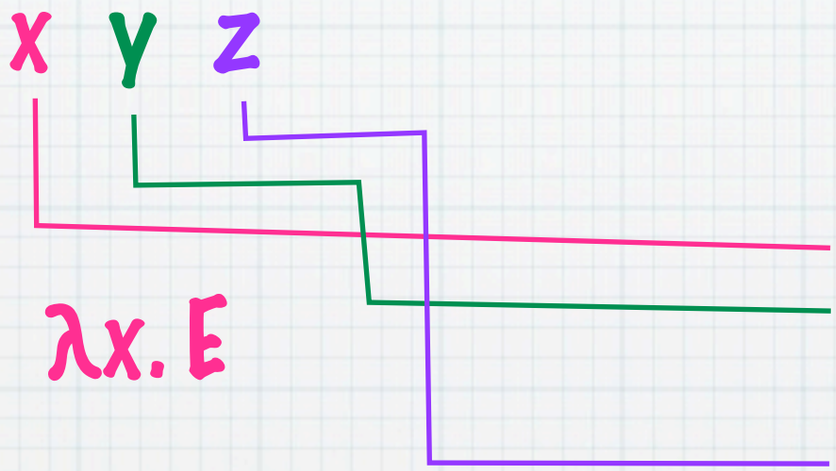
$\lambda x.E$

function application

$E_1 E_2$

What are they?

wires

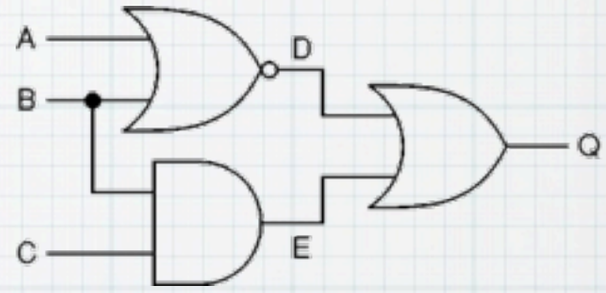


circuit templates

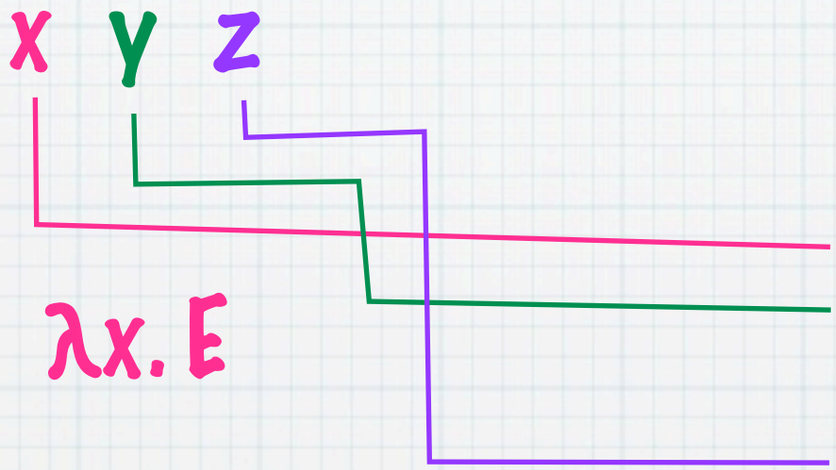
function application

$E_1 E_2$

What are they?



wires



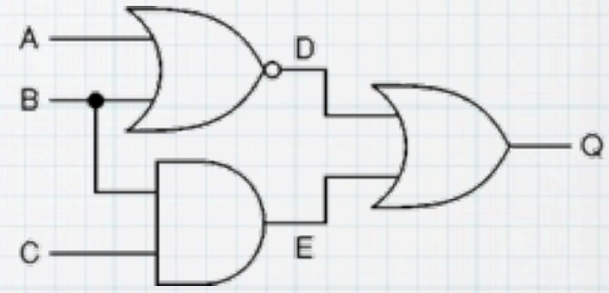
circuit templates

$\lambda x.E$

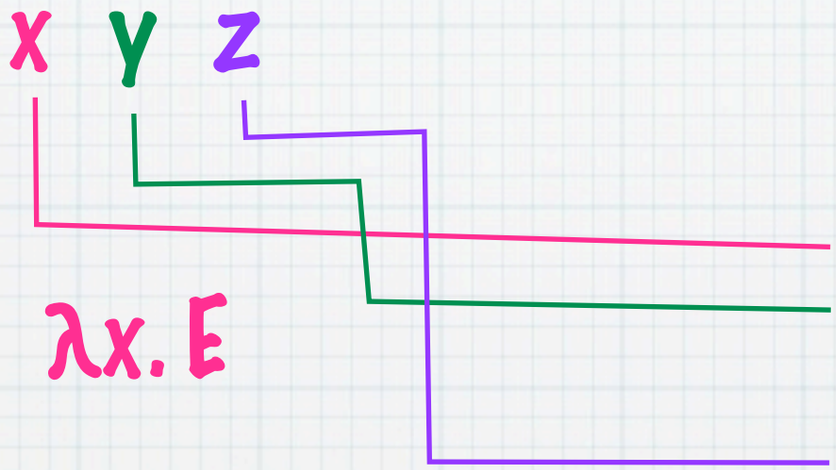
function application

$E_1 E_2$

What are they?



wires



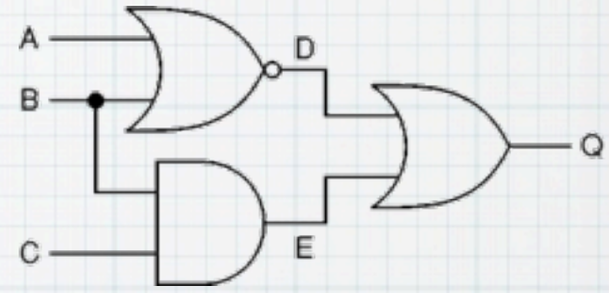
circuit templates

$\lambda x.E$

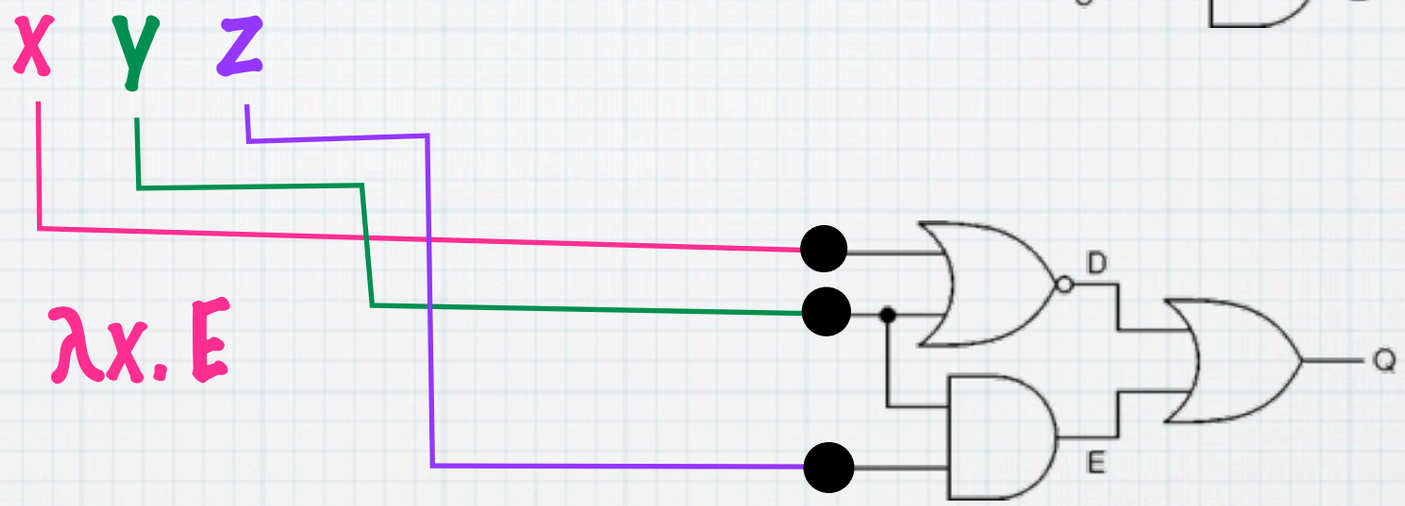
circuit template instantiation and connection

$E_1 E_2$

What are they?



wires



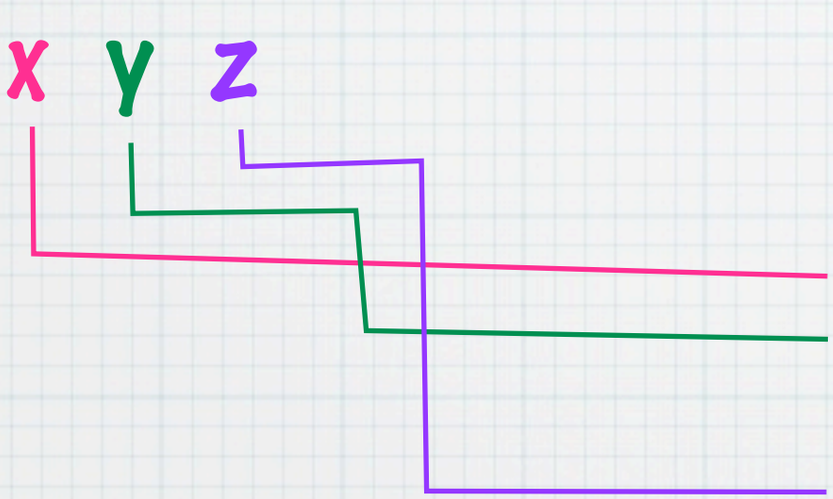
circuit templates

$\lambda x.E$

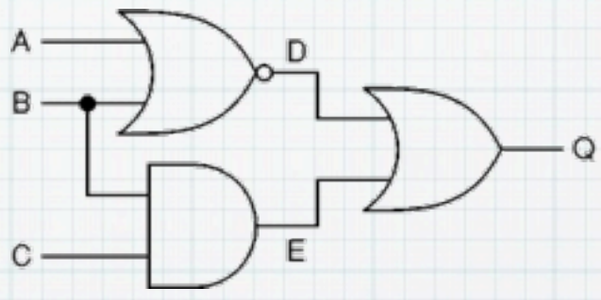
circuit template instantiation and connection

$E_1 E_2$

What are Types?

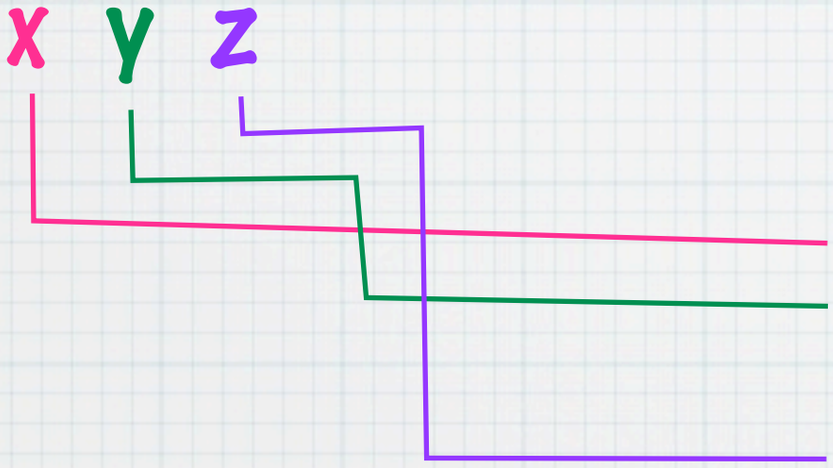


F=λxyz.E

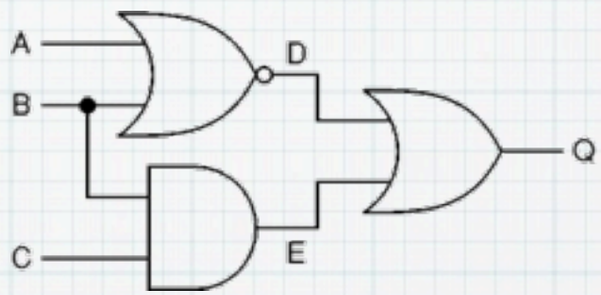


What are Types?

Types are "interface specifications"



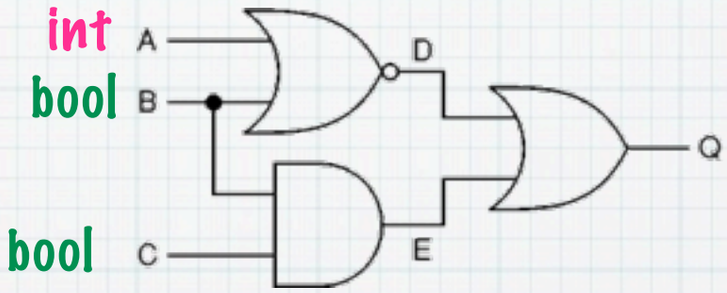
F=λxyz.E

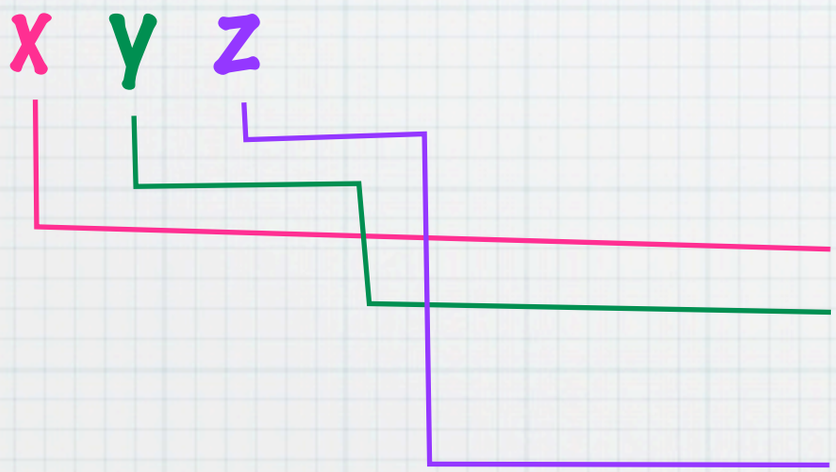


What are Types?

Types are "interface specifications"

F=λxyz.E

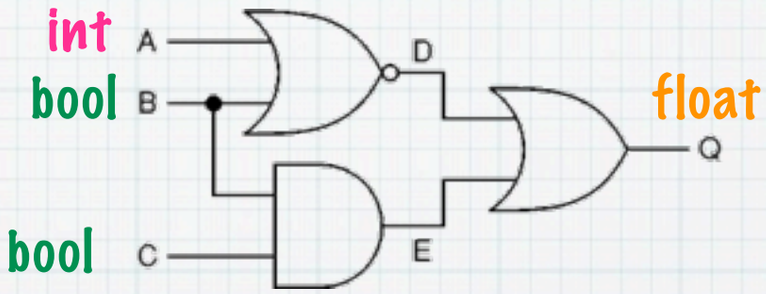


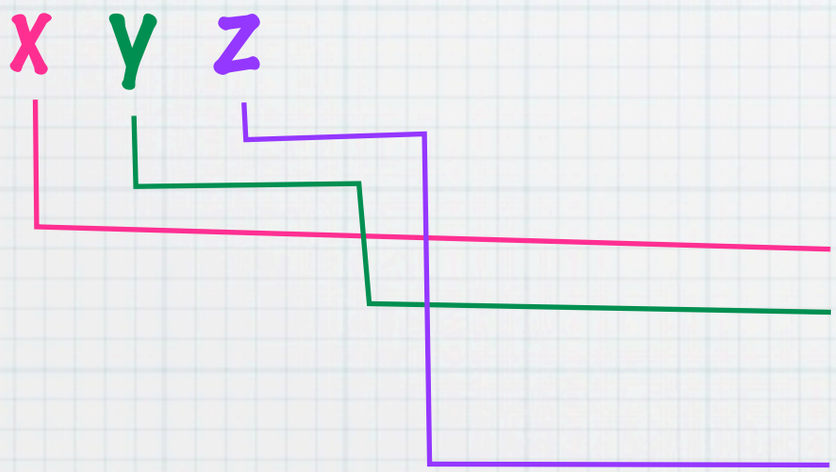


What are Types?

Types are "interface specifications"

F=λxyz.E

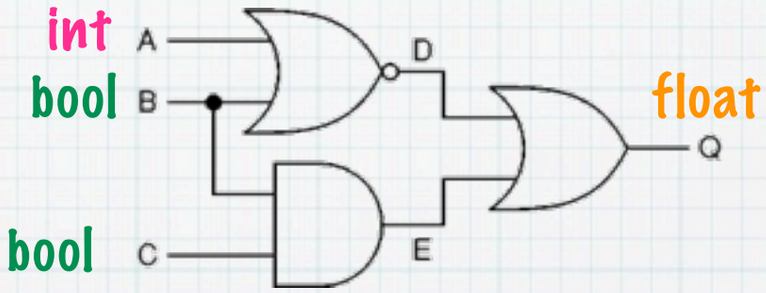




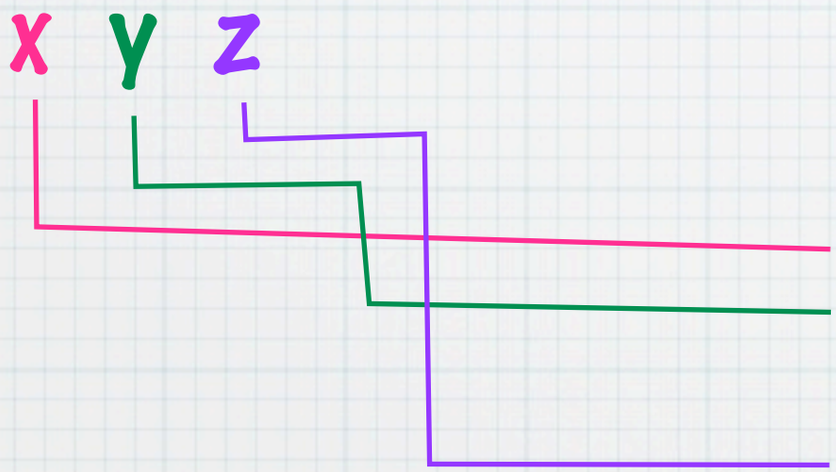
What are Types?

Types are "interface specifications"

F=λxyz.E

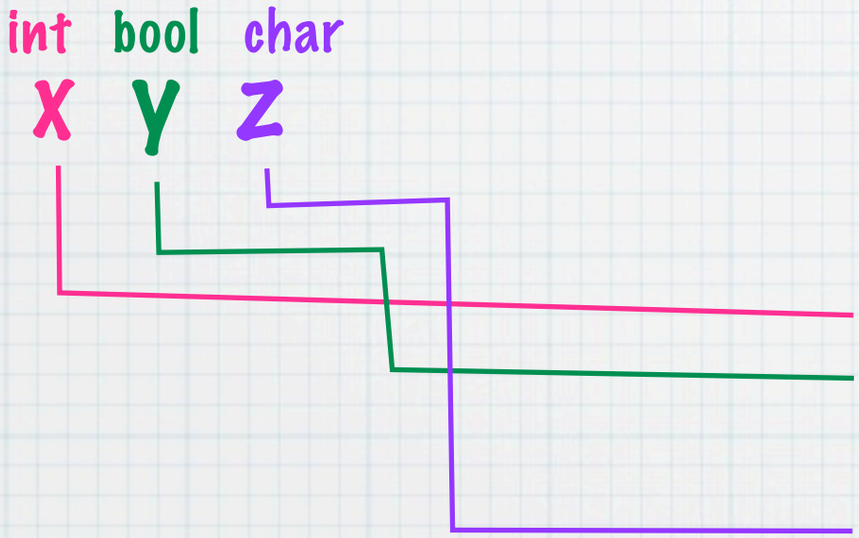


In type theoretical notation:
(int, bool, bool) → float

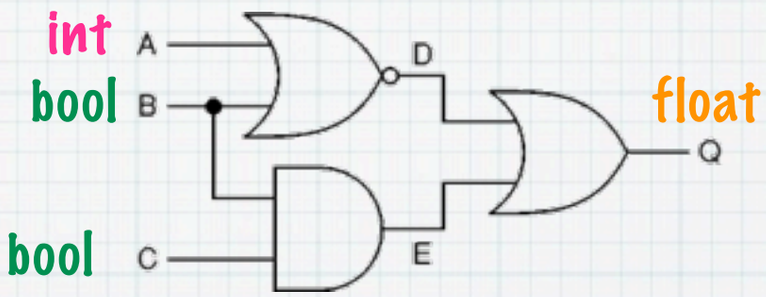


What are Types?

Types are "interface specifications"



F=λxyz.E

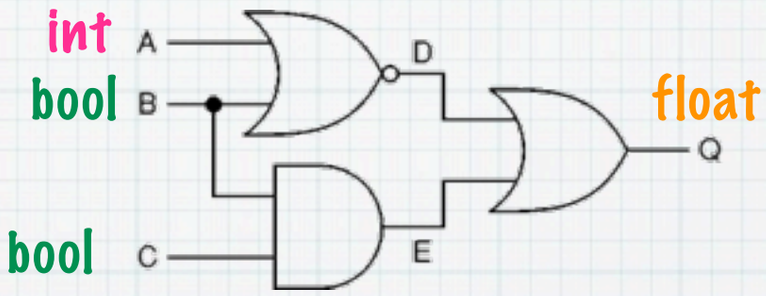


In type theoretical notation:
(int, bool, bool) → float

What are Types?

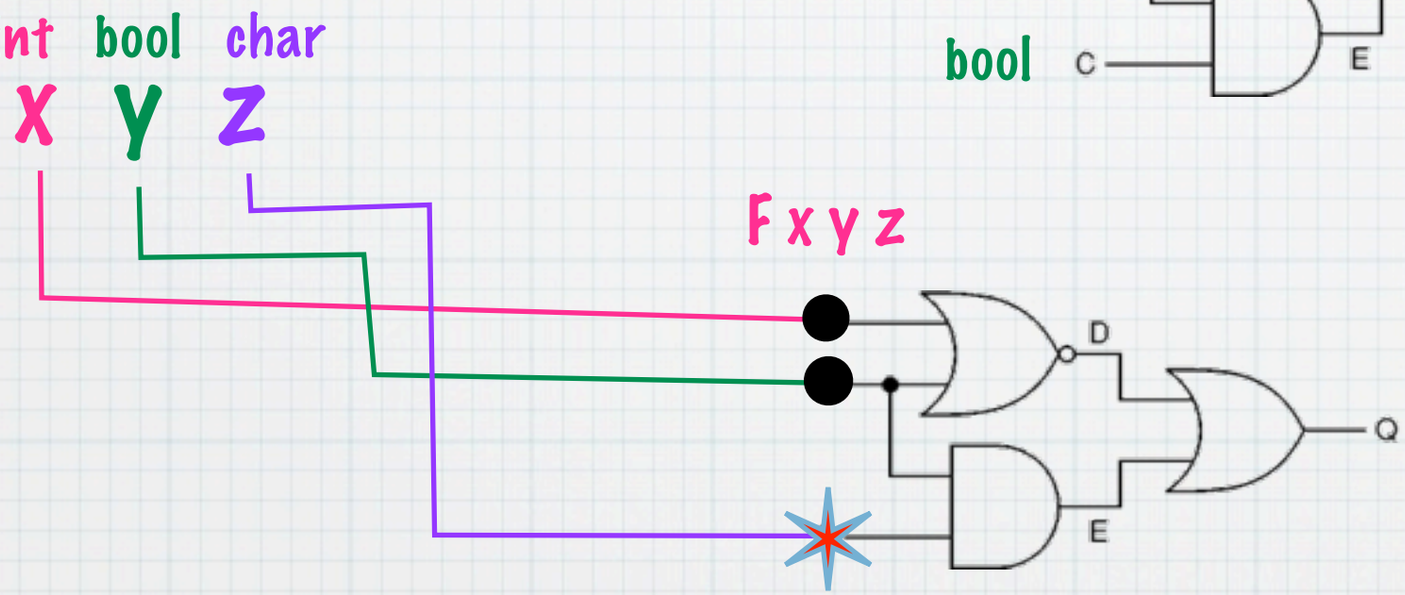
Types are "interface specifications"

F=λxyz.E



In type theoretical notation:
(int, bool, bool) → float

int bool char
X Y Z

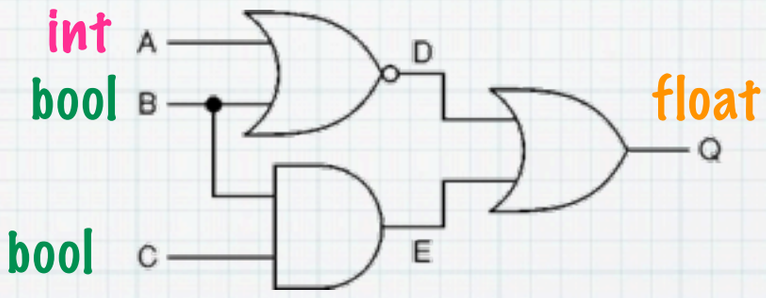


What are Types?

Types are "interface specifications"

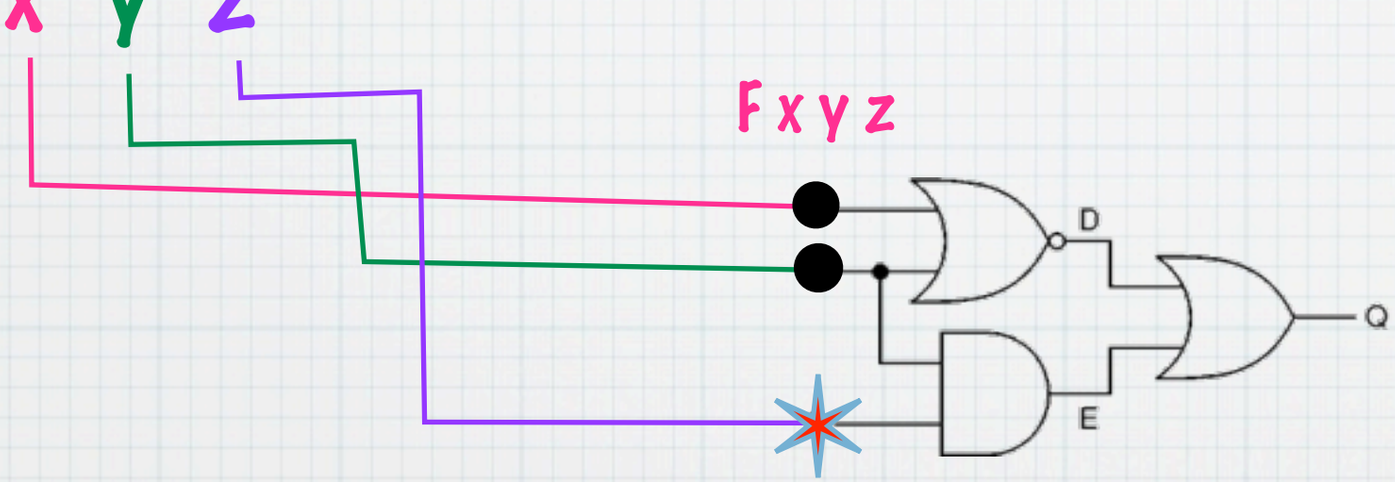
F=λxyz.E

In type theoretical notation:
(int, bool, bool) → float



int bool char
X Y Z

F x y z

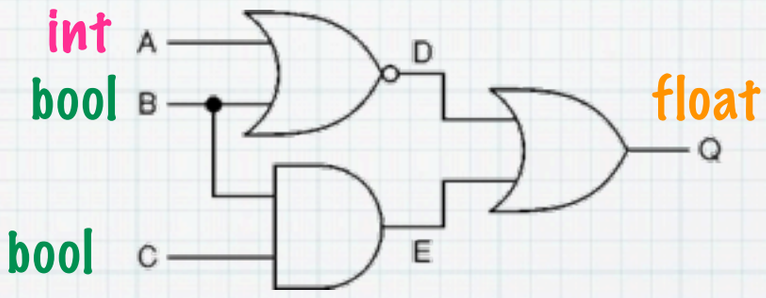


Type Error!

What are Types?

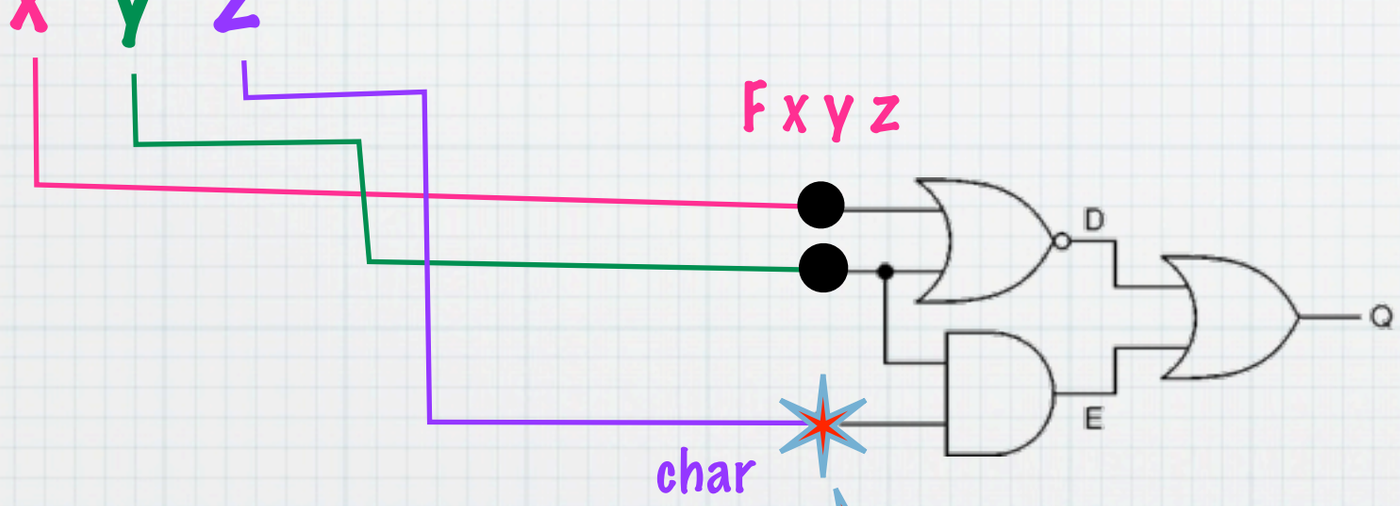
Types are "interface specifications"

F=λxyz.E



In type theoretical notation:
(int, bool, bool) → float

int bool char
X Y Z

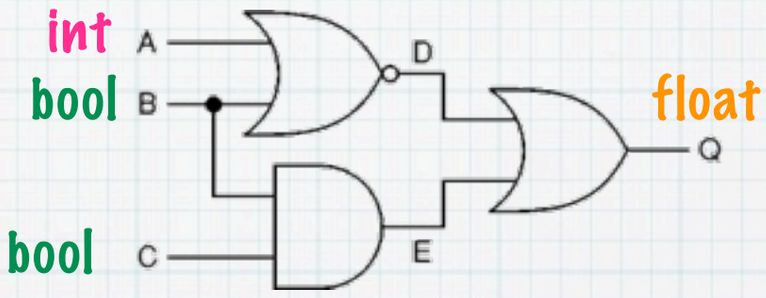


Type Error!

What are Types?

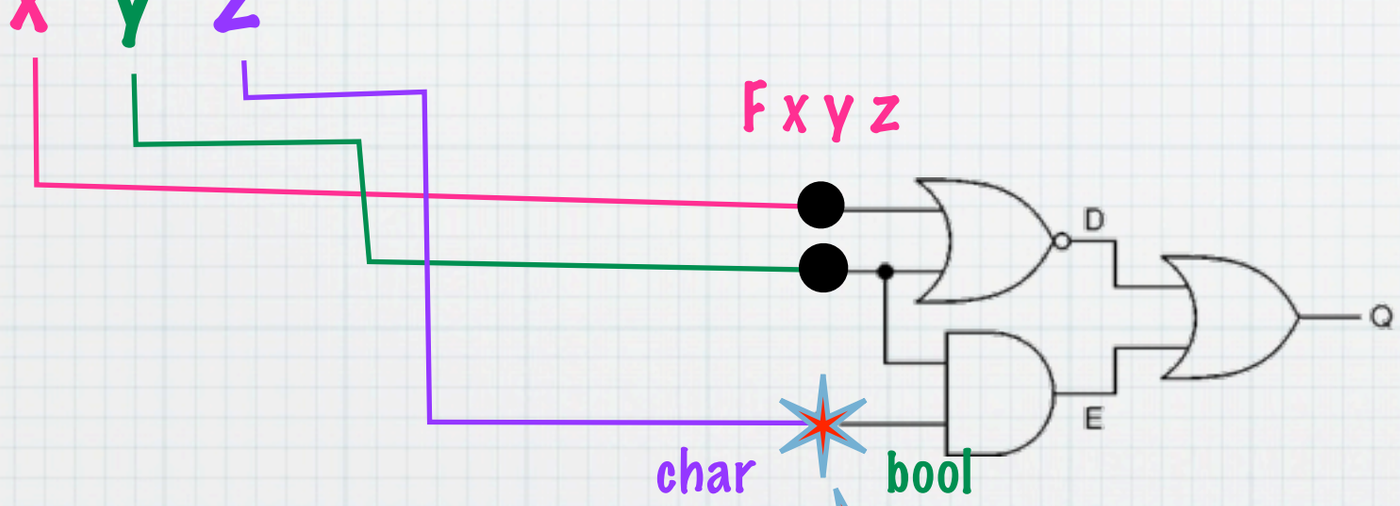
Types are "interface specifications"

F=λxyz.E



In type theoretical notation:
(int, bool, bool) → float

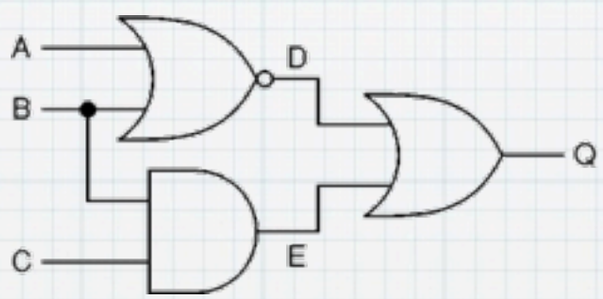
int bool char
X Y Z



Type Error!

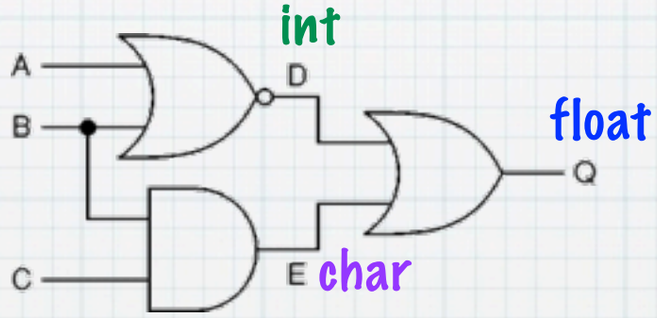
What is Type Inference?

Given circuit with **partially labeled** pins,
infer the label for other pins



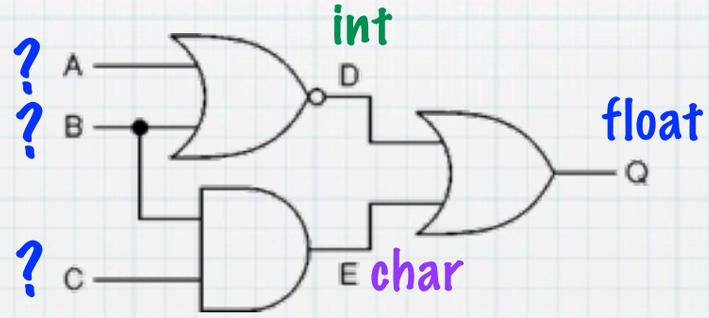
What is Type Inference?

Given circuit with **partially labeled** pins,
infer the label for other pins



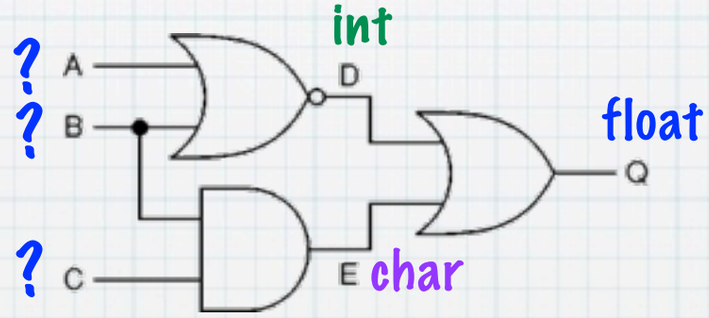
What is Type Inference?

Given circuit with **partially labeled** pins,
infer the label for other pins



What is Type Inference?

Given circuit with **partially labeled** pins,
infer the label for other pins



In essence a logic programming problem

In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* $\lambda f.\lambda x.f\ x$
* $(a \to b) \to a \to b$

In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* $\lambda f.\lambda x.f\ x$
* $(a \to b) \to a \to b$

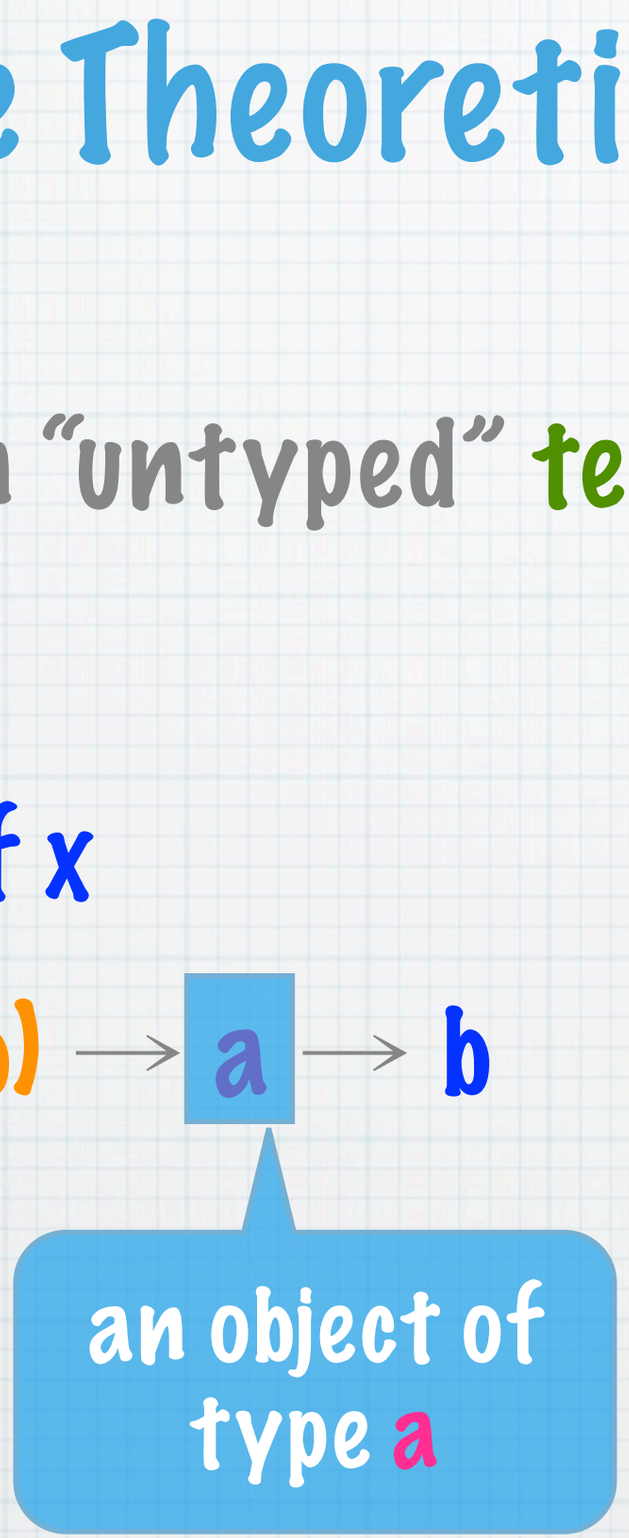
a function from a to b

In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* $\lambda f.\lambda x.f\ x$
* $(a \to b) \to a \to b$

In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* λf.λx.f x
* (a → b) → a → b



In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* $\lambda f.\lambda x.f\ x$
* $(a \to b) \to a \to b$

In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* λf.λx.f x
* (a → b) → a → b

returns an object of type b

In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* $\lambda f.\lambda x.f\ x$
* $(a \to b) \to a \to b$

In Type Theoretical Notations

* Given an "untyped" term, infer its type
* Example
* λf.λx.f x
* (a → b) → a → b

a and b are called "type variables" (white lie)

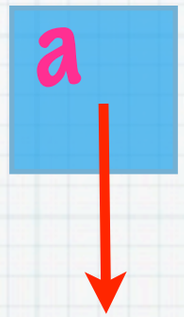
What are Type Variables?

What are Type Variables?

A type variable
is really just a
reference cell
(a pointer in C)

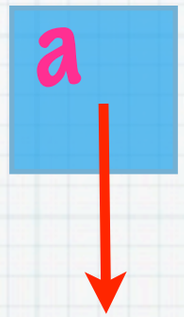
What are Type Variables?

A type variable
is really just a
reference cell
(a pointer in C)



What are Type Variables?

A type variable is really just a **reference cell** (a pointer in C)

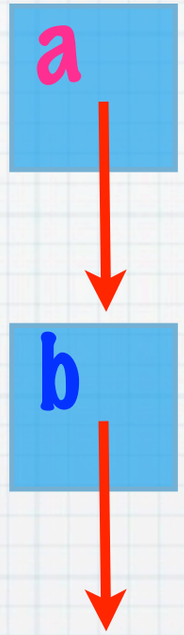


They may point to other type variables or types

What are Type Variables?

A type variable is really just a **reference cell** (a pointer in C)

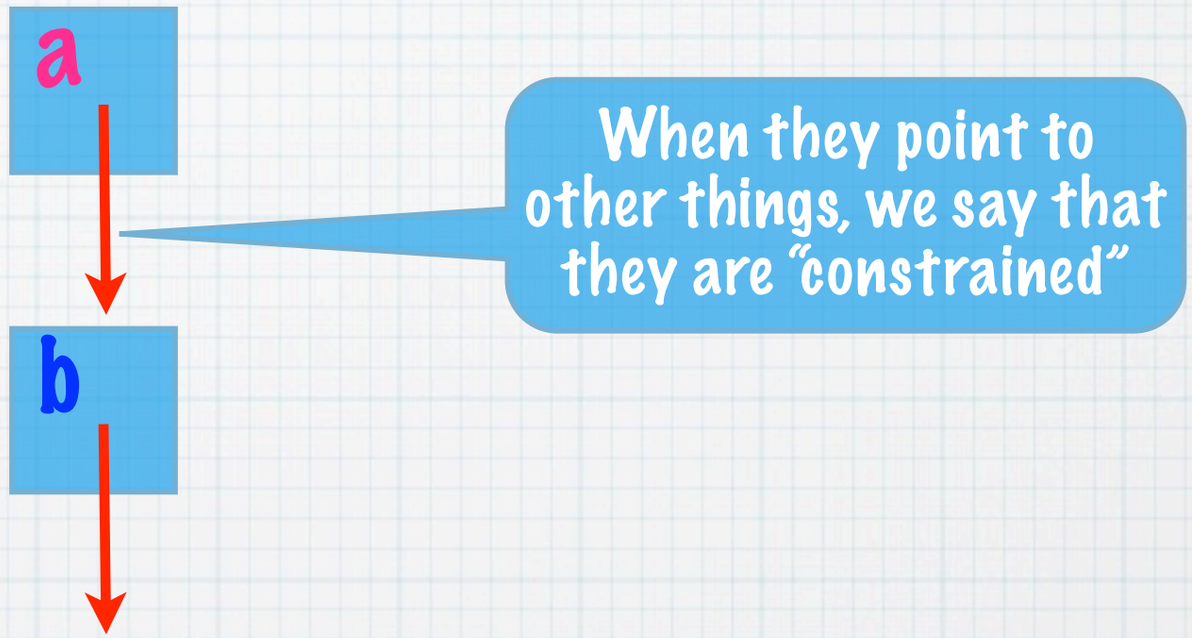
They may point to other type variables or types



What are Type Variables?

A type variable is really just a **reference cell** (a pointer in C)

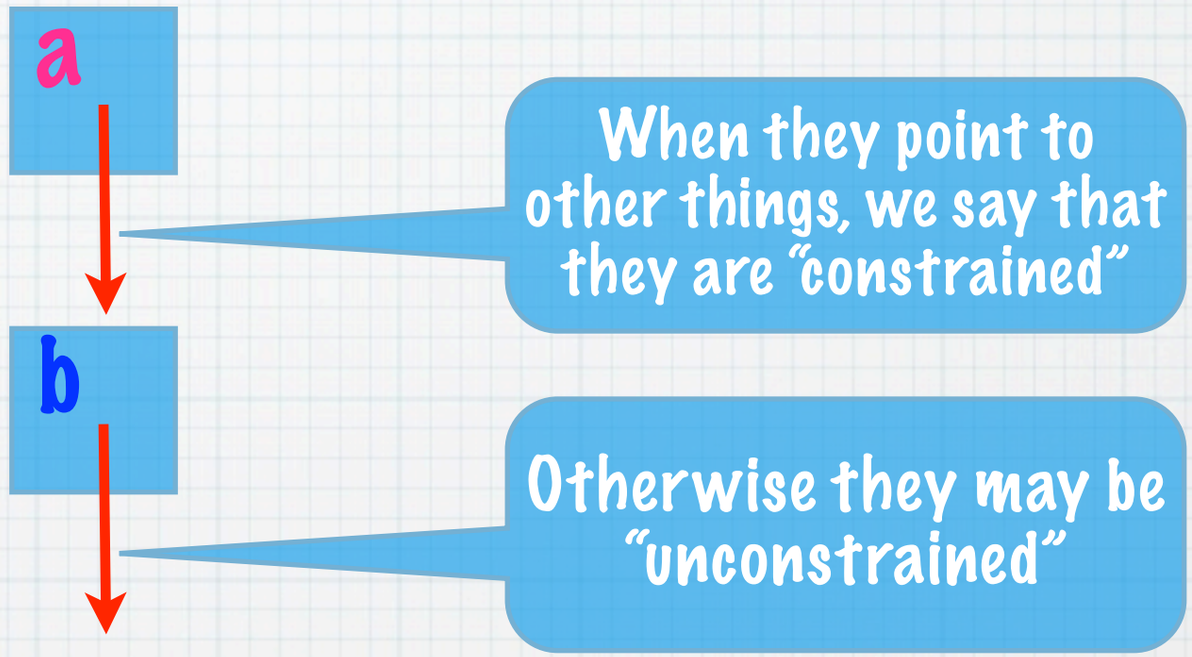
They may point to other type variables or types



What are Type Variables?

A type variable is really just a **reference cell** (a pointer in C)

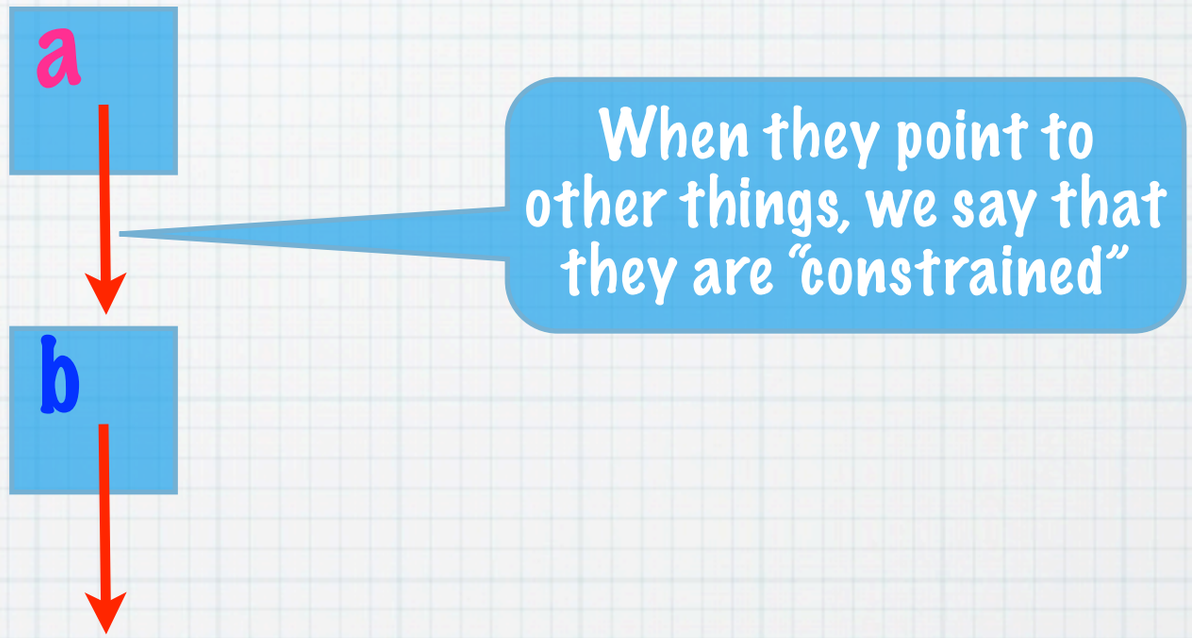
They may point to other type variables or types



What are Type Variables?

A type variable is really just a **reference cell** (a pointer in C)

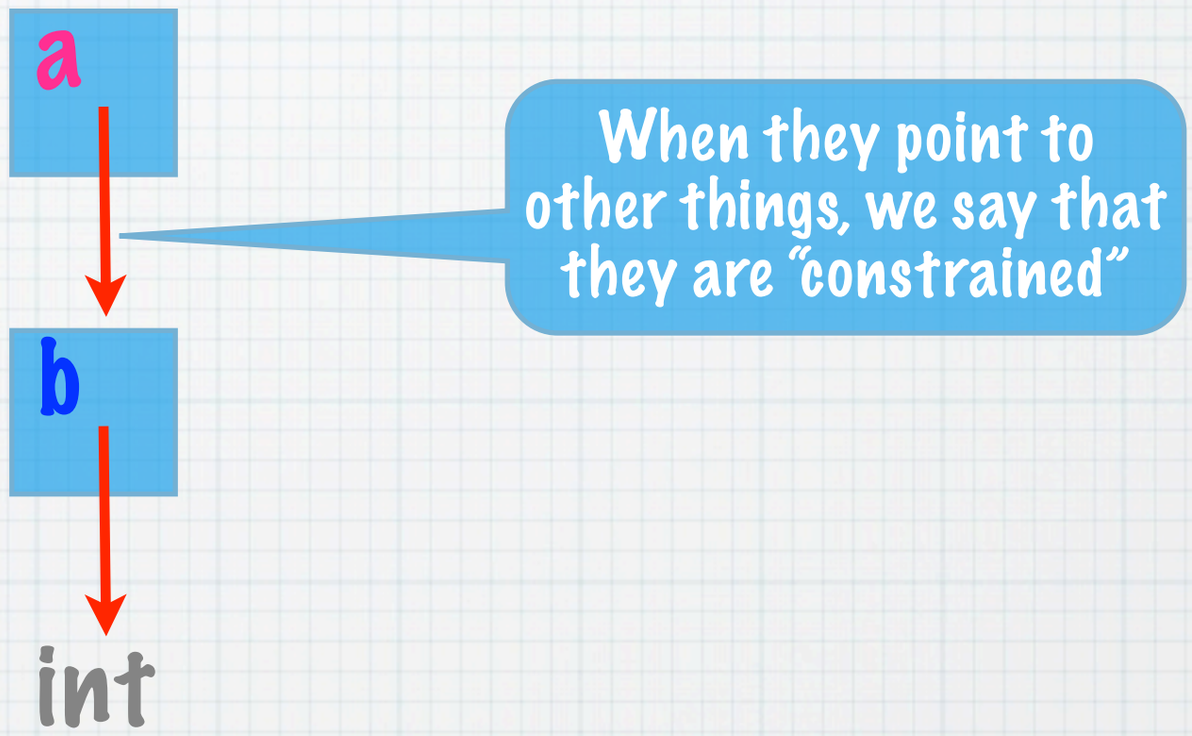
They may point to other type variables or types



What are Type Variables?

A type variable is really just a **reference cell** (a pointer in C)

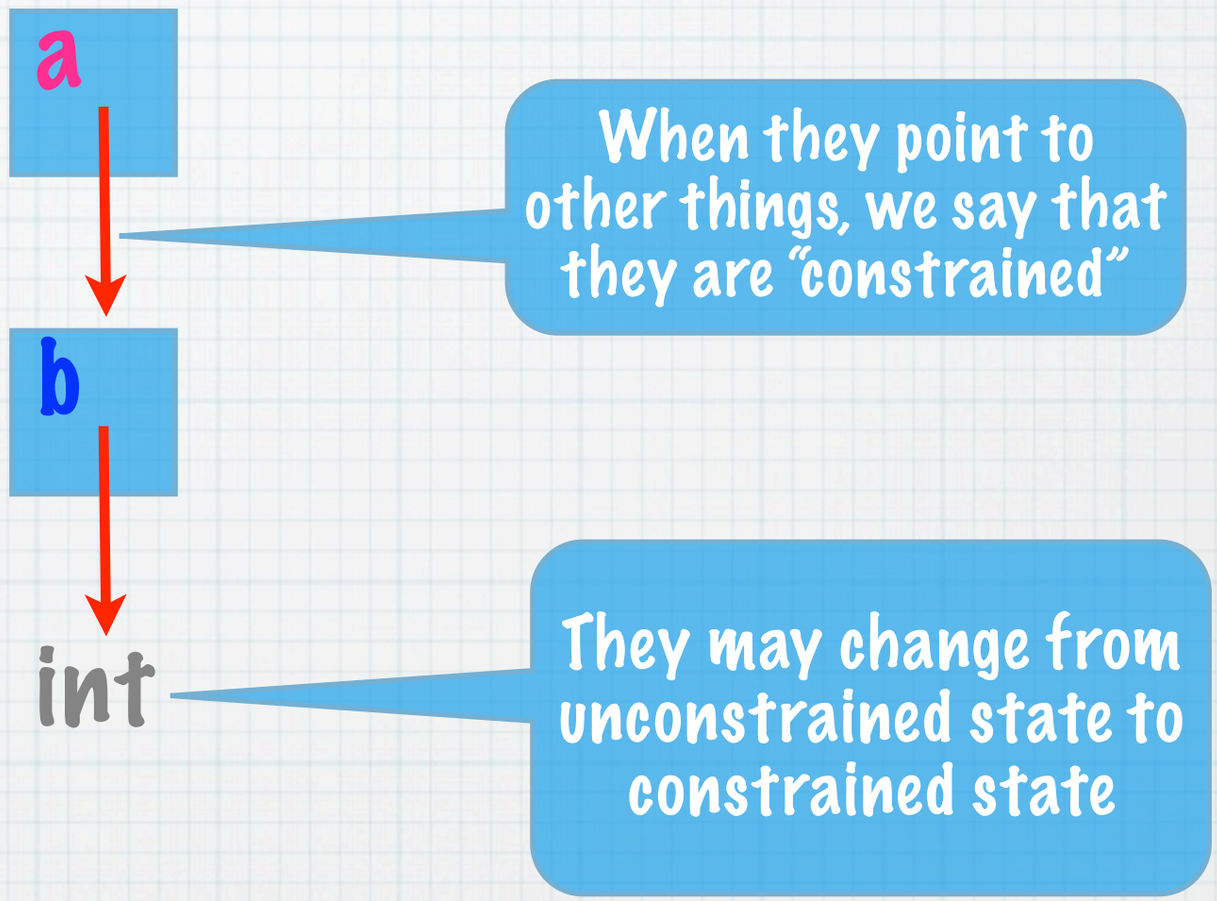
They may point to other type variables or types



What are Type Variables?

A type variable is really just a **reference cell** (a pointer in C)

They may point to other type variables or types



Hindley-Milner System (HMS)

$\lambda f.\ \lambda x.\ f\ x$

Hindley-Milner System (HMS)

$$\lambda f.\ \lambda x.\ f\ x$$

Hindley-Milner System (HMS)

Process the term "recursively", building a **data structure** (using pointers) from it

$$\lambda f.\ \lambda x.\ f\ x$$

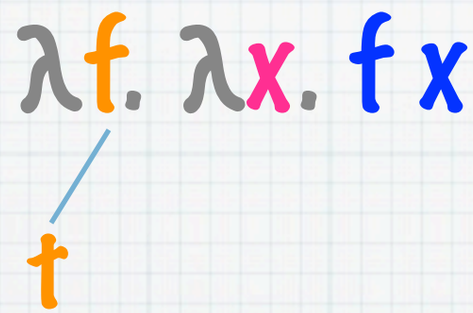
Hindley-Milner System (HMS)

Process the term "recursively", building a data structure (using pointers) from it

$$\lambda f.\ \lambda x.\ f\ x$$

Hindley-Milner System (HMS)

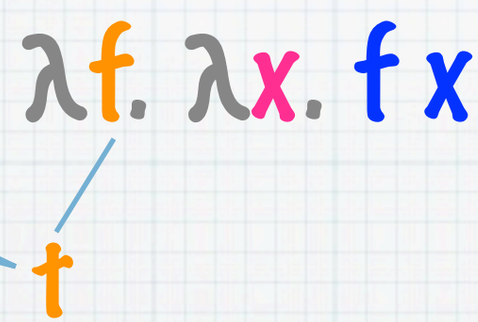
Process the term "recursively", building a data structure (using pointers) from it



Hindley-Milner System (HMS)

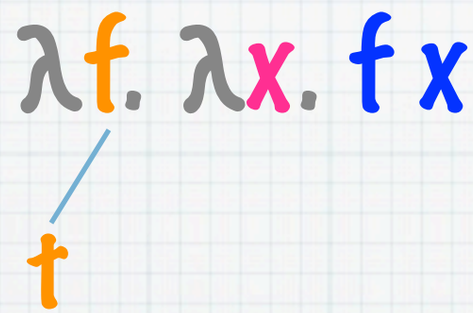
Process the term "recursively", building a data structure (using pointers) from it

Associate with each parameter a "type variable" when going down



Hindley-Milner System (HMS)

Process the term "recursively", building a data structure (using pointers) from it



Hindley-Milner System (HMS)

Process the term "recursively", building a **data structure** (using pointers) from it

$$\lambda f.\ \lambda x.\ f\ x$$

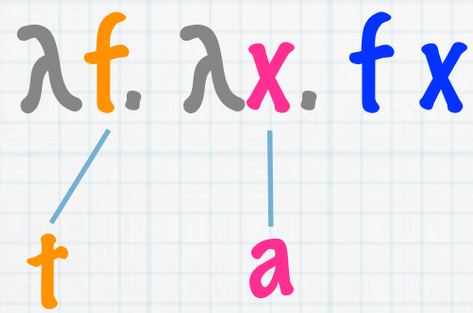
t

Hindley-Milner System (HMS)

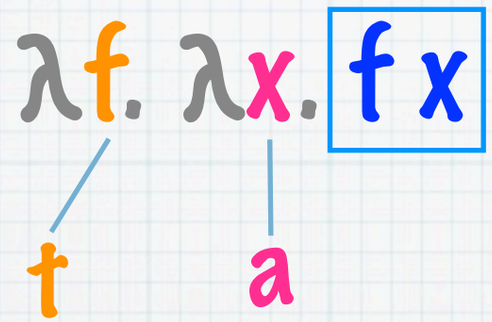
$$\lambda f.\ \lambda x.\ f\ x$$

$$t$$

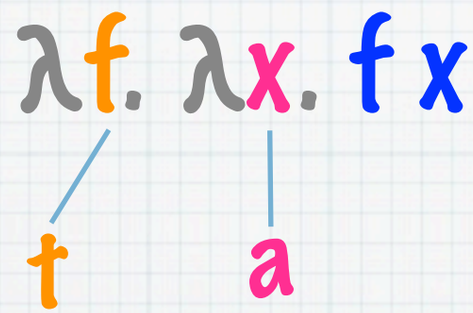
Hindley-Milner System (HMS)



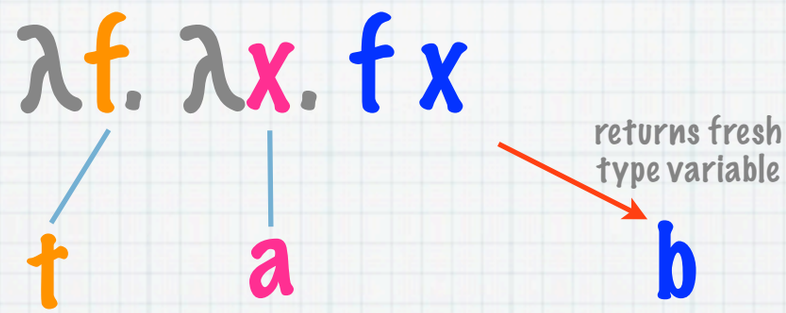
Hindley-Milner System (HMS)



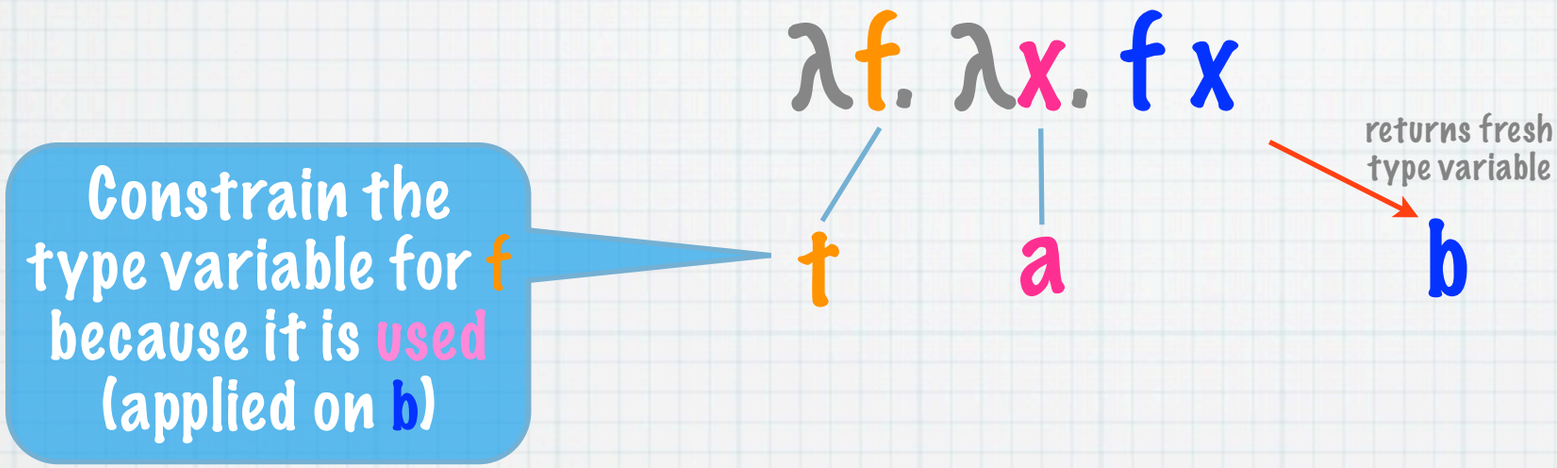
Hindley-Milner System (HMS)



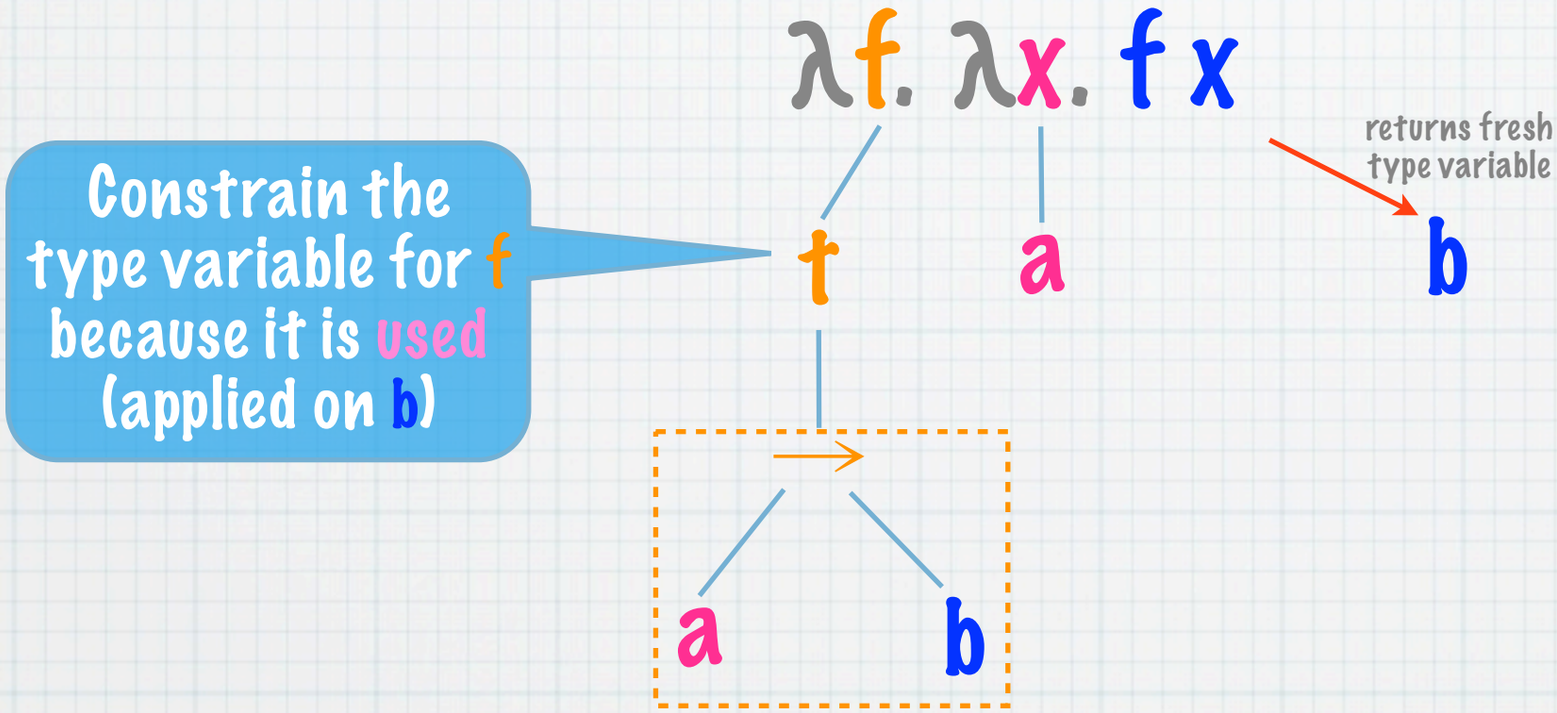
Hindley-Milner System (HMS)



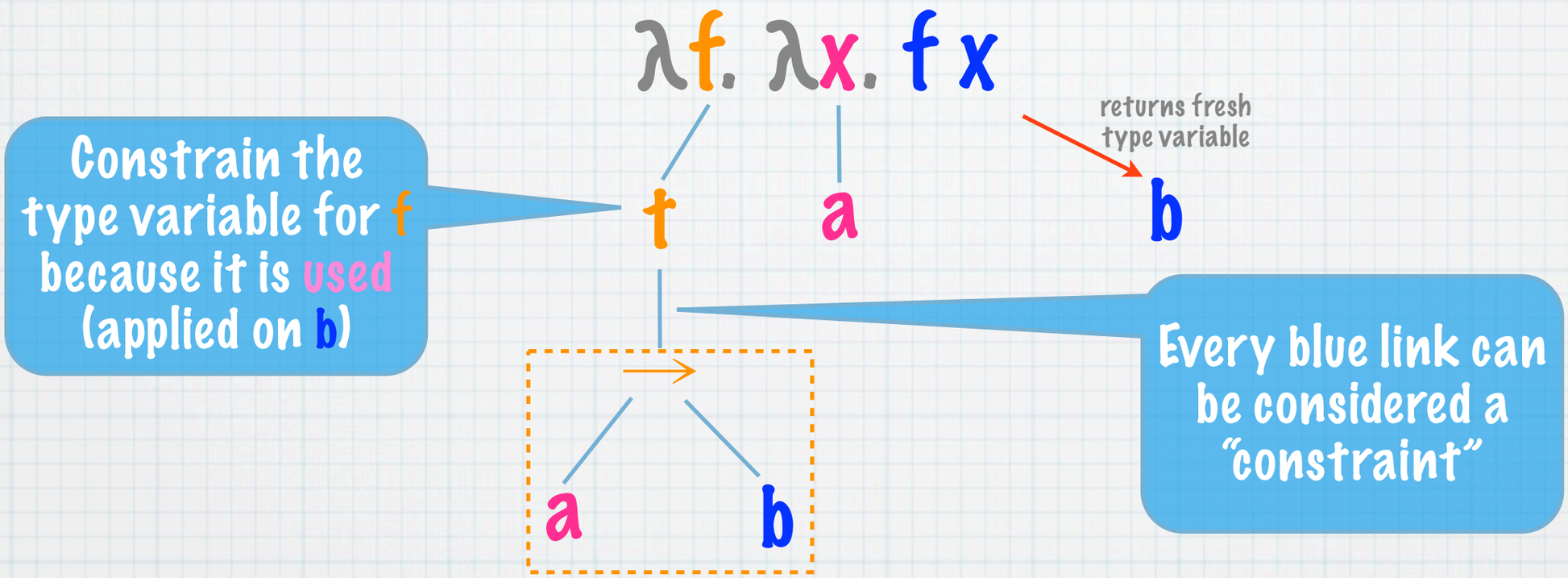
Hindley-Milner System (HMS)



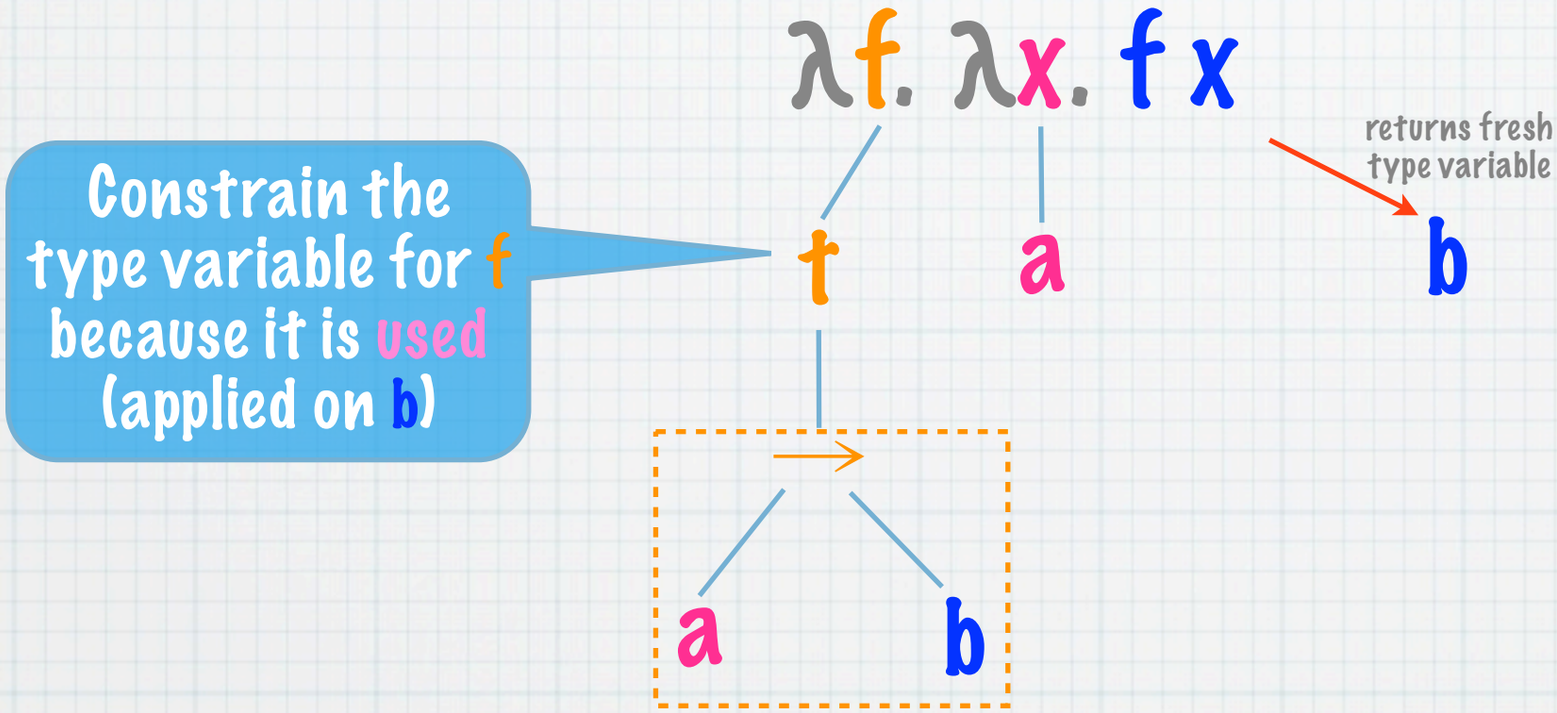
Hindley-Milner System (HMS)



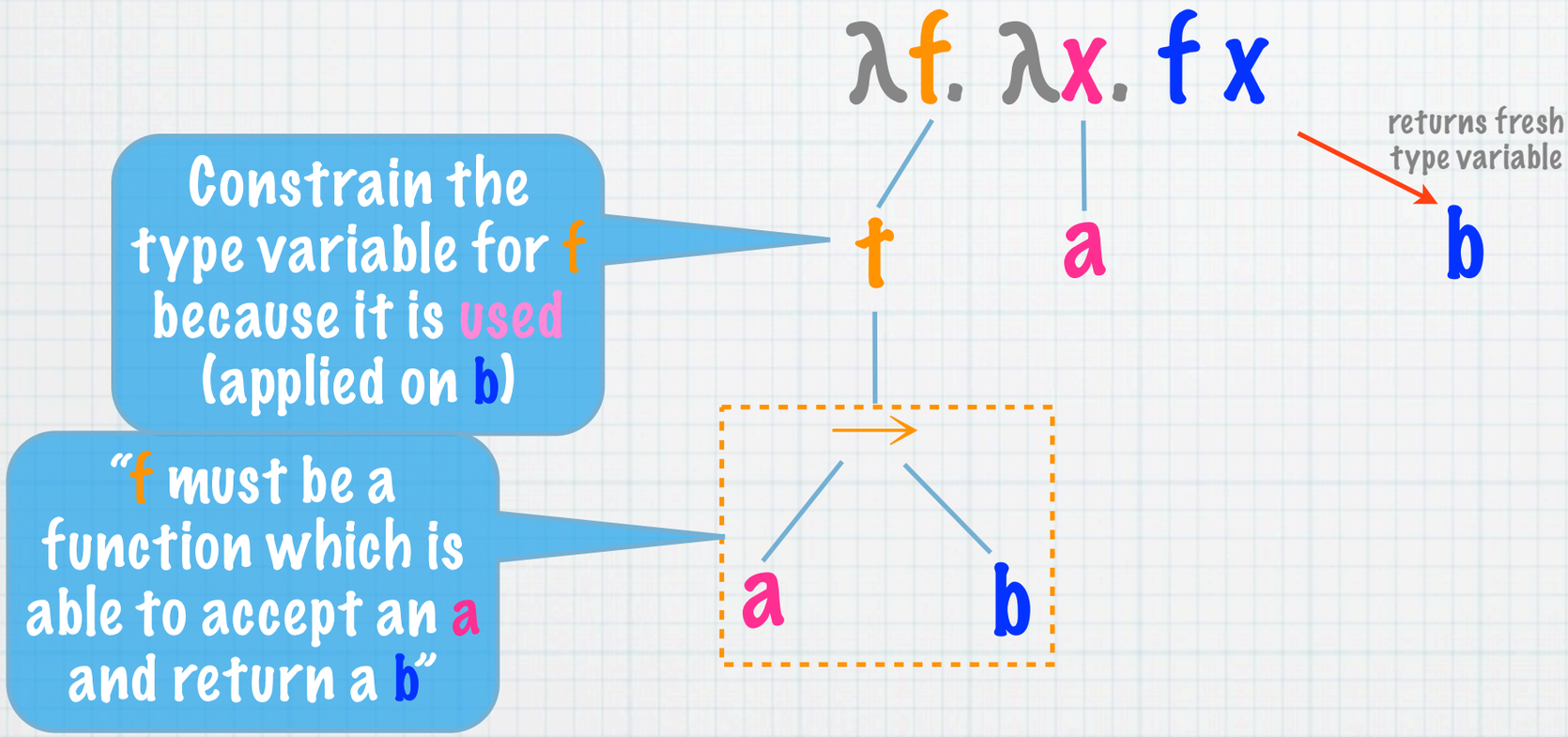
Hindley-Milner System (HMS)



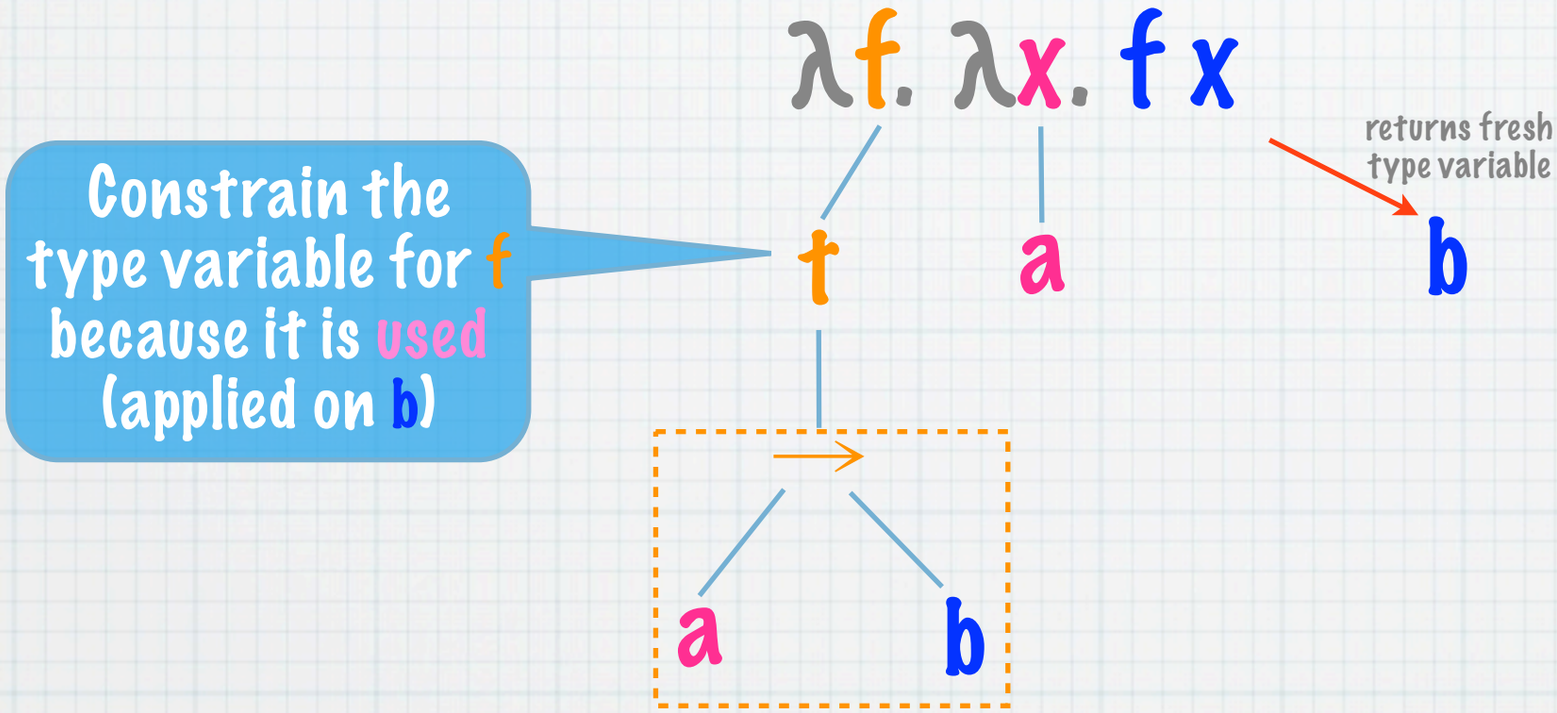
Hindley-Milner System (HMS)



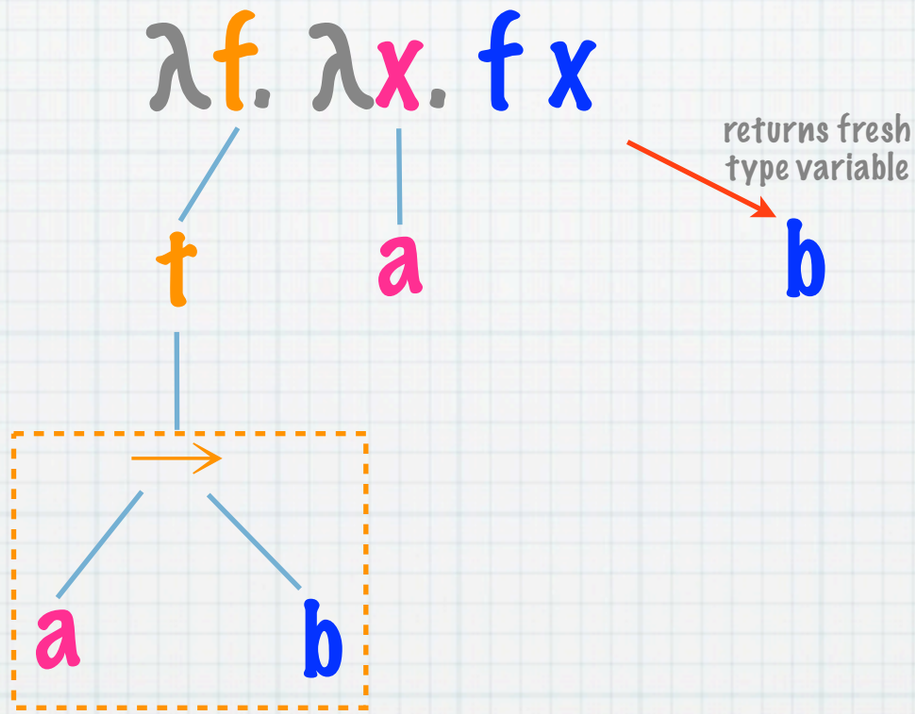
Hindley-Milner System (HMS)



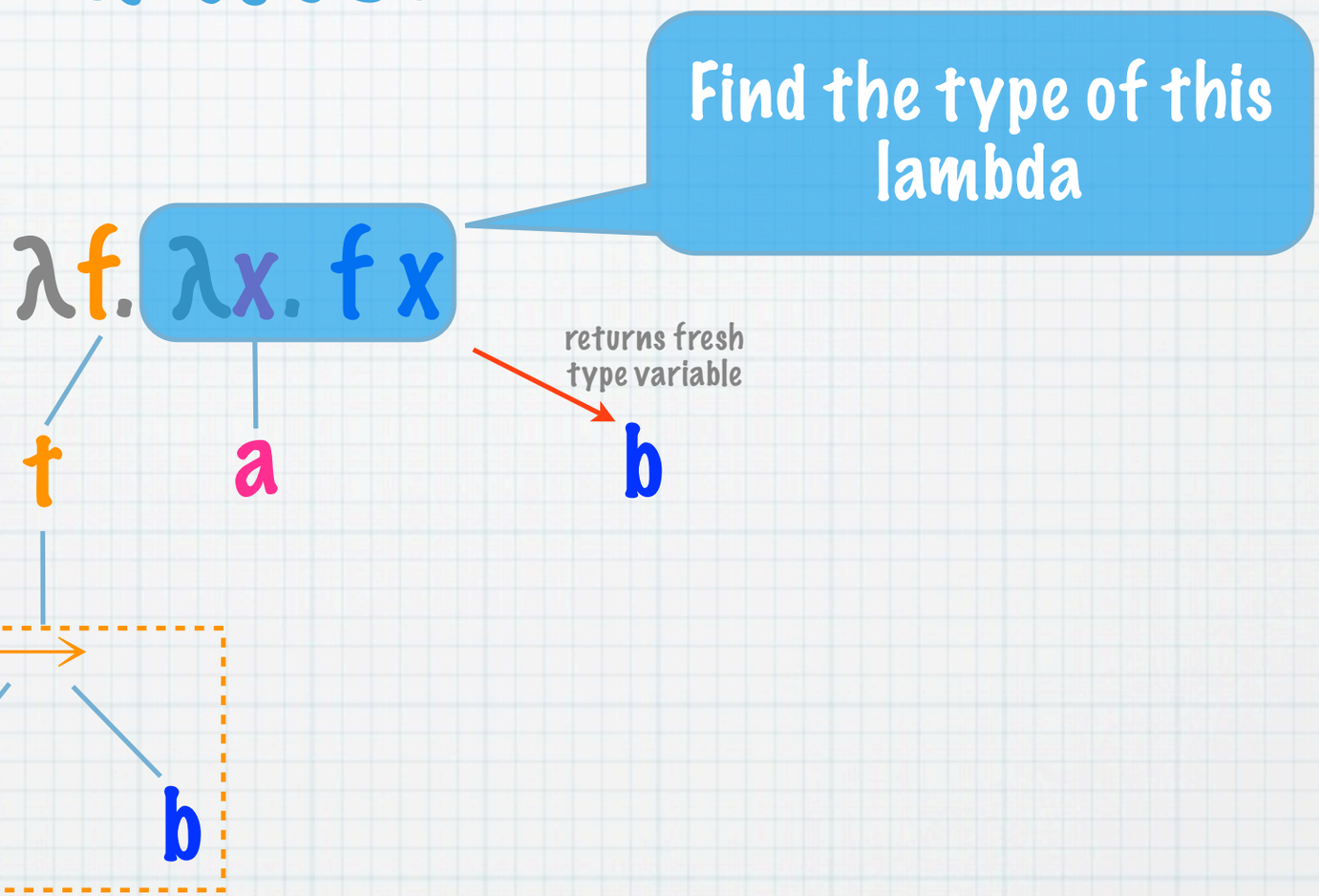
Hindley-Milner System (HMS)



Hindley-Milner System (HMS)



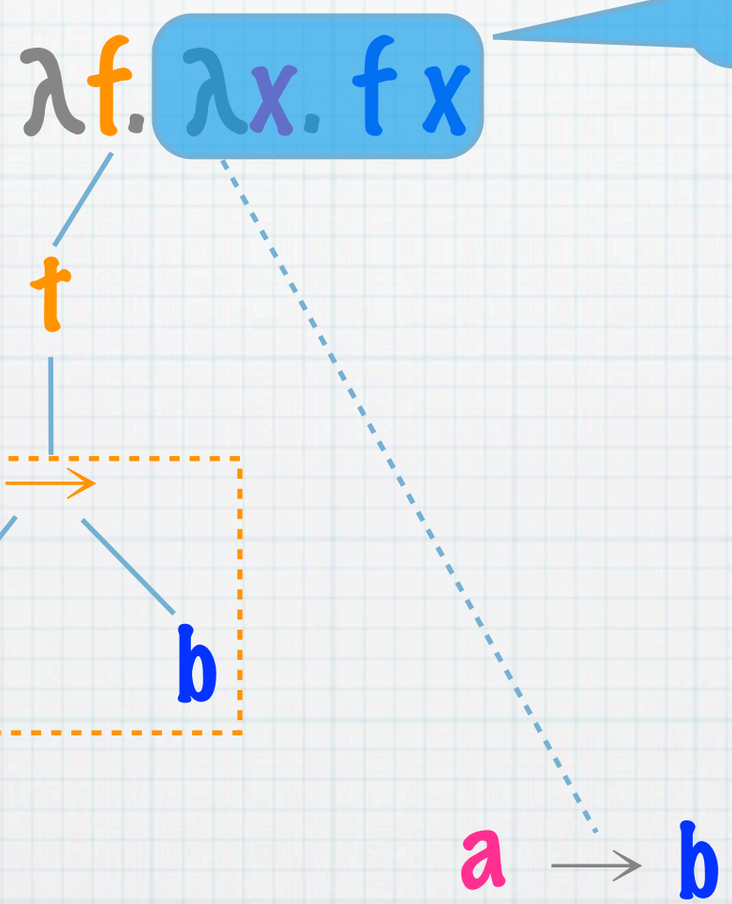
Hindley-Milner System (HMS)



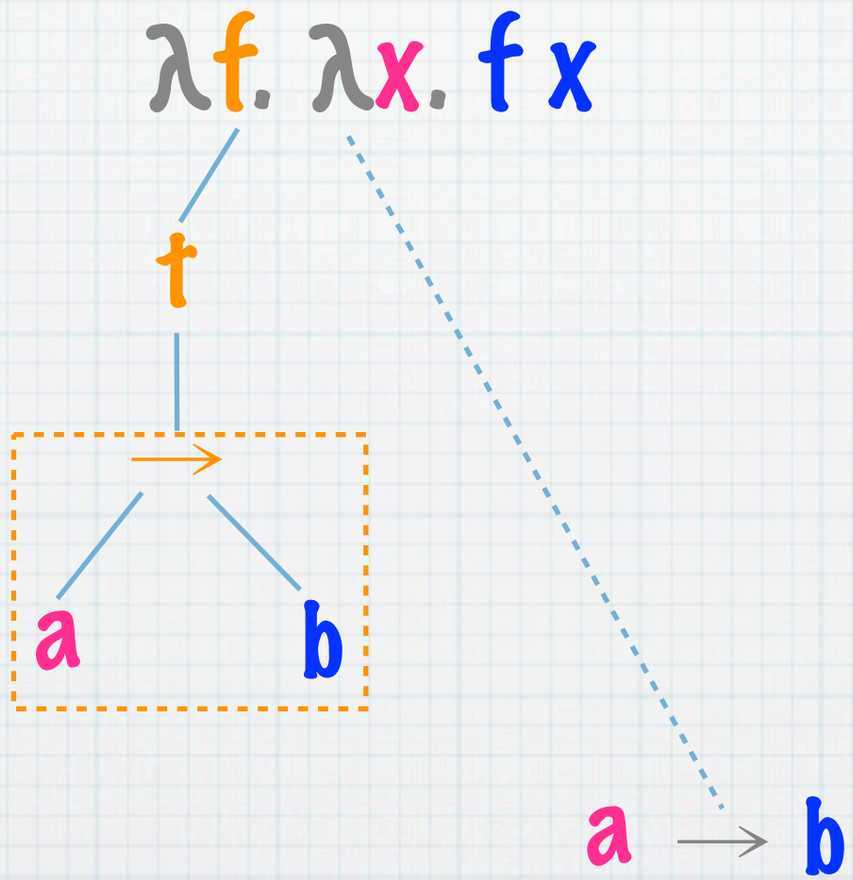
Hindley-Milner System (HMS)

Find the type of this lambda

λf. λx. f x



Hindley-Milner System (HMS)



Hindley-Milner System (HMS)

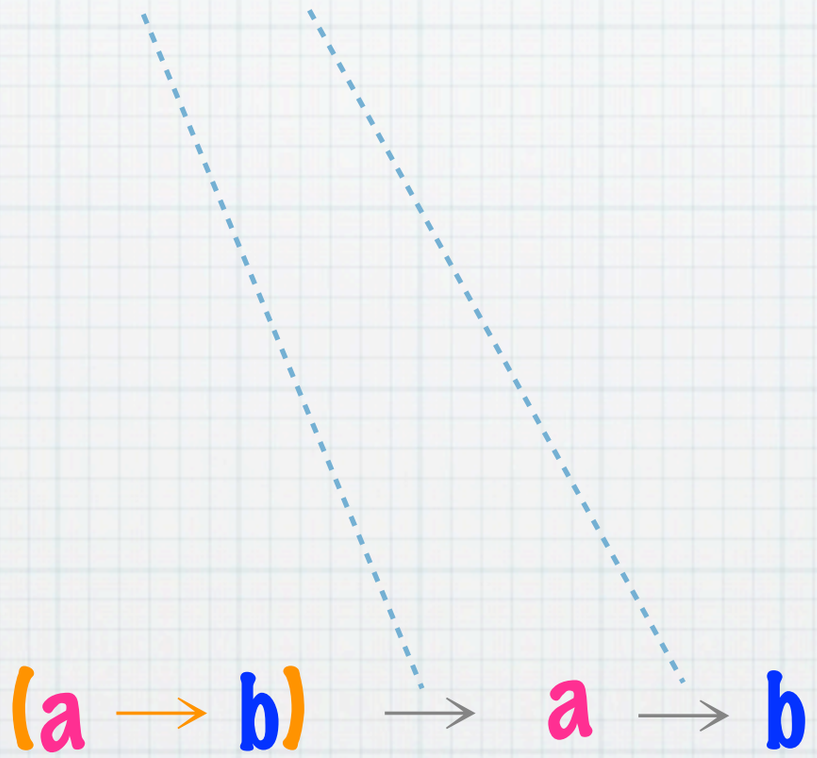
Find the type of this lambda

$$\lambda f.\ \lambda x.\ f\ x$$

$$(a \to b) \to a \to b$$

Hindley-Milner System (HMS)

$$\lambda f.\ \lambda x.\ f\ x$$



Hindley-Milner System (HMS)

$$\lambda f.\ \lambda x.\ f\ x$$

$$(a \to b) \to a \to b$$

How many kinds of arrows are there?

Hindley-Milner System (HMS)

$$\lambda f.\ \lambda x.\ f\ x$$

$$(a \to b) \to a \to b$$

Are a and b constrained?

Hindley-Milner System (HMS)

$$\lambda f.\ \lambda x.\ f\ x$$

$$(a \to b) \to a \to b$$

Can a and b be constrained?

Why do we need polymorphism?

```
let f =λx.x in
(f 1, f True)
```

Why do we need polymorphism?

If we don't have polymorphism ...

```
let f =λx.x in
(f 1, f True)
```

Why do we need polymorphism?

If we don't have polymorphism ...

```
let f = λx.x in
(f 1, f True)
```

Why do we need polymorphism?

If we don't have polymorphism ...

What is f's type using our method?

```
let f = λx.x in
(f 1, f True)
```

Why do we need polymorphism?

If we don't have polymorphism ...

What is f's type using our method?

```
let f = λx.x in
(f 1, f True)
```

a → a

Why do we need polymorphism?

If we don't have polymorphism ...

What is f's type using our method?

```
let f = λx.x in
(f 1, f True)
```

$a \rightarrow a$

What is f's type after the application (f 1)?

Why do we need polymorphism?

If we don't have polymorphism ...

What is f's type using our method?

```
let f = λx.x in
(f 1, f True)
```

What is f's type after the application (f 1)?

int → int

a → a

Why do we need polymorphism?

If we don't have polymorphism ...

What is f's type using our method?

```
let f = λx.x in
(f 1, f True)
```

What is f's type after the application (f 1)?

int → int

Why?

a → a

Why do we need polymorphism?

If we don't have polymorphism ...

What is f's type using our method?

```
let f = λx.x in
(f 1, f True)
```

What is f's type after the application (f 1)?

a → a

int → int

Why?

Because a was constrained to int when we apply f to 1. Once constrained, type variables stay constrained forever.

Why do we need polymorphism?

If we don't have polymorphism ...

What is f's type using our method?

```
let f = λx.x in
(f 1, f True)
```

What is f's type after the application (f 1)?

$a \rightarrow a$

int

$int \rightarrow int$

Why?

Because a was constrained to int when we apply f to 1. Once constrained, type variables stay constrained forever.

Why do we need polymorphism?

If we don't have polymorphism ...

```
let f = λx.x in
(f 1, f True)
```

What is f's type using our method?

$a \rightarrow a$

a

int

Because a was constrained to int when we apply f to 1. Once constrained, type variables stay constrained forever.

Type Error !

What is f's type after the application (f 1)?

int → int

Why?

How to implement polymorphism (the HM way)

```
let f =λx.x in
(f 1, f True)
```

How to implement polymorphism (the HM way)

"Generalize" when we finish typing the RHS of let bindings (let-polymorphism)

```
let f = λx.x in
(f 1, f True)
```

How to implement polymorphism (the HM way)

"Generalize" when we finish typing the RHS of let bindings (let-polymorphism)

```
let f = λx.x in
```

a → a

```
(f 1, f True)
```

How to implement polymorphism (the HM way)

```
let f =λx.x in
(f 1, f True)
```

a → a

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f =λx.x in
(f 1, f True)
```

a → a

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

Each application gets a "fresh instance" of $a \rightarrow a$

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

$a_1 \rightarrow a_1$

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \to a$

$a_1 \to a_1$

$a_2 \to a_2$

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \to a$

What is f's type after the application (f 1)?

$a_1 \to a_1$

$a_2 \to a_2$

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

What is f's type after the application (f 1)?

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \to a$

$a_1 \to a_1$

$a_2 \to a_2$

still $\forall a.\ a \to a$

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \to a$

What is f's type after the application (f 1)?

$a_1 \to a_1$

$a_2 \to a_2$

still $\forall a.\ a \to a$

Why?

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

What is f's type after the application (f 1)?

$a_1 \rightarrow a_1$

$a_2 \rightarrow a_2$

still $\forall a.\ a \rightarrow a$

Why?

Because type variables can be constrained only in their own instances

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

What is f's type after the application (f 1)?

$a_1 \rightarrow a_1$

$a_2 \rightarrow a_2$

still $\forall a.\ a \rightarrow a$

Why?

Because type variables can be constrained only in their own instances

What does this behavior of $\forall a.\ a \rightarrow a$ reminds us of?

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

$\forall a.a \rightarrow a$

What is f's type after the application (f 1)?

```
let f = λx.x in
(f 1, f True)
```

$a_1 \rightarrow a_1$

$a_2 \rightarrow a_2$

still $\forall a.\ a \rightarrow a$

Why?

Because type variables can be constrained only in their own instances

What does this behavior of $\forall a.\ a \rightarrow a$ reminds us of?

This is the behavior of a function!

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

What is f's type after the application (f 1)?

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

still $\forall a.\ a \rightarrow a$

$a_1 \rightarrow a_1$

$a_2 \rightarrow a_2$

Why?

Because type variables can be constrained only in their own instances

What does this behavior of $\forall a.\ a \rightarrow a$ reminds us of?

This is the behavior of a **function**!

Is $\forall a.\ a \rightarrow a$ in fact a function in disguise?

How to implement polymorphism (the HM way)

Basically, consider every unconstrained variable on the RHS as "universally quantified" (white lie)

```
let f = λx.x in
(f 1, f True)
```

$\forall a.a \rightarrow a$

What is f's type after the application (f 1)?

$a_1 \rightarrow a_1$

$a_2 \rightarrow a_2$

still $\forall a.\ a \rightarrow a$

Why?

Because type variables can be constrained only in their own instances

What does this behavior of $\forall a.\ a \rightarrow a$ reminds us of?

This is the behavior of a **function**!

Is $\forall a.\ a \rightarrow a$ in fact a function in disguise? **We will see!**

What is wrong with let-polymorphism?

For any value x of type A

What is wrong with let-polymorphism?

For any value x of type A

Intuitively, we expect [ ] to have type [ ]

What is wrong with let-polymorphism?

For any value x of type A

Intuitively, we expect [x] to have type [ ]

What is wrong with let-polymorphism?

For any value `x` of type `A`

Intuitively, we expect `[x]` to have type `[A]`

What is wrong with let-polymorphism?

For any value `x` of type `A`

Intuitively, we expect `[x]` to have type `[A]`

This is called "closed under composition"

What is wrong with let-polymorphism?

For any value `x` of type `A`

Intuitively, we expect `[x]` to have type `[A]`

This is called "**closed under composition**"

Why is this correct?

What is wrong with let-polymorphism?

For any value x of type A

Intuitively, we expect [x] to have type [A]

This is called "closed under composition"

Why is this correct?

- When the user writes [x], she expects to be able to take x out and use it as its original type A

What is wrong with let-polymorphism?

For any value x of type A

Intuitively, we expect [x] to have type [A]

This is called "closed under composition"

Why is this correct?

- When the user writes [x], she expects to be able to take x out and use it as its original type A
- So whatever you put in [ ], it must be compatible with A (i.e. a subtype of A)

What is wrong with let-polymorphism?

For any value x of type A

Intuitively, we expect [x] to have type [A]

This is called "closed under composition"

Why is this correct?

- When the user writes [x], she expects to be able to take x out and use it as its original type A
- So whatever you put in [ ], it must be compatible with A (i.e. a subtype of A)
- The type [A] is for this check

What we get from let-polymorphism

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [ ]
  expect: [ ]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [ ]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [ ]
  expect: [ ]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [ ]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [∀a.[a -> a]]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

✳ Notice the positions of the ∀ quantifiers

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

✳ Notice the positions of the ∀ quantifiers

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

✳ Notice the positions of the ∀ quantifiers

✳ They are are not determined by the **contents** of RHS terms

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

✳ Notice the positions of the ∀ quantifiers

✳ They are are not determined by the **contents** of RHS terms

✳... But the **position** of the RHS terms

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

  ▼

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

  ▼

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

✳ Notice the positions of the ∀ quantifiers

✳ They are are not determined by the **contents** of RHS terms

✳... But the **position** of the RHS terms

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

  ▼

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

  ▼

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

✳ Notice the positions of the ∀ quantifiers

✳ They are are not determined by the **contents** of RHS terms

✳... But the **position** of the RHS terms

✳ What does this remind us of?

What we get from let-polymorphism

* let x = [1]
  x is typed [int] (okay)

* let id = λx.x
  id is typed ∀a.a->a

* let y = [id]
  expect: [∀a.a->a] but get: ∀a.[a -> a]

* let y = [[id]]
  expect: [∀a.[a -> a]] but get: ∀a.[[a -> a]]

✳ Notice the positions of the ∀ quantifiers

✳ They are are not determined by the **contents** of RHS terms

✳... But the **position** of the RHS terms

✳ What does this remind us of?

dynamic scoping

If anything feels wrong, it will go wrong ...

```
let r = ref (λx.x) in
    (r := λx.x+1; (!r)true)
```

If anything feels wrong, it will go wrong ...

What is the type of this?

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

If anything feels wrong, it will go wrong ...

```
let r = ref (λx.x) in
    (r := λx.x+1; (!r)true)
```

If anything feels wrong, it will go wrong ...

∀a.(ref (a->a))

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

If anything feels wrong, it will go wrong ...

Remember what the quantifier ∀a mean?

∀a.(ref (a->a))

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

If anything feels wrong, it will go wrong ...

Remember what the quantifier ∀a mean?

∀a.(ref (a->a))

It means make an **instance** of its body at every use

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

If anything feels wrong, it will go wrong ...

∀a.(ref (a->a))

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

If anything feels wrong, it will go wrong ...

∀a.(ref (a->a))

let r = ref (λx.x) in
(r := λx.x+1; (!r)true)

ref (a_1->a_1)

If anything feels wrong, it will go wrong ...

$\forall a.(\text{ref } (a \to a))$

$\text{let } r = \text{ref } (\lambda x.x) \text{ in}$

$(r := \lambda x.x+1;\ (!r)\text{true})$

$\text{ref } (a_1 \to a_1)$

$\text{ref } (a_2 \to a_2)$

If anything feels wrong, it will go wrong ...

∀a.(ref (a->a))

let r = ref (λx.x) in
(r := λx.x+1; (!r)true)

ref (a_1->a_1)

ref (a_2->a_2)

Passed type check

If anything feels wrong, it will go wrong ...

$\forall a.(\text{ref } (a \to a))$

$\text{let } r = \text{ref } (\lambda x.x) \text{ in}$

$(r := \lambda x.x+1;\ (!r)\text{true})$

$\text{ref } (a_1 \to a_1)$

$\text{ref } (a_2 \to a_2)$

Unsound!

ML's Solution
(value restriction)

∀a.(ref (a->a))

let r = ref (λx.x) in
(r := λx.x+1; (!r)true)

ML's Solution
(value restriction)

No generalization for "non-values" (**ref** is a case of non-value)

∀a.(ref (a->a))

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

ML's Solution
(value restriction)

No generalization for "non-values" (**ref** is a case of non-value)

(ref (a->a))

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

ML's Solution
(value restriction)

No generalization for "non-values" (ref is a case of non-value)

(ref (a->a))

No longer make instances

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

ML's Solution
(value restriction)

(ref (a->a))

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

Type changes to int->int because **a** was constrained to **int**

Fails type check because int->int cannot be applied to **bool**

But ... This solves only one symptom of the problem

* There are more of them:
  * Interference with **first-class continuations**
  * Interference with **intersection types**
  * And so on ...

But ... This solves only one symptom of the problem

* There are more of them:
  * Interference with **first-class continuations**
  * Interference with **intersection types**
  * And so on ...

Can we get rid of the root of the problem?

Mental Burden on the Programmer

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?
  * variables: YES

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?
  * variables: YES
  * functions: YES

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?
  * variables: YES
  * functions: YES
  * Applications: NO

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?
  * variables: YES
  * functions: YES
  * Applications: NO
  * Constructor calls: YES

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?
  * variables: YES
  * functions: YES
  * Applications: NO
  * Constructor calls: YES
    * Ah wait... except ref ... NO

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?
  * variables: YES
  * functions: YES
  * Applications: NO
  * Constructor calls: YES
    * Ah wait... except ref ... NO
* There can be a lot more to remember ...

Mental Burden on the Programmer

- Value Restriction = Only "values" are generalized at LET
- What are "values"?
  - variables: YES
  - functions: YES
  - Applications: NO
  - Constructor calls: YES
    - Ah wait... except ref ... NO
  - There can be a lot more to remember ...
- The trouble of remembering those can be a lot more than that of writing type annotations!

Mental Burden on the Programmer

* Value Restriction = Only "values" are generalized at LET
* What are "values"?
  * variables: YES
  * functions: YES
  * Applications: NO
  * Constructor calls: YES
    * Ah wait... except ref ... NO
  * There can be a lot more to remember ...
* The trouble of remembering those can be a lot more than that of writing type annotations!
* They have nothing to do with the programmer's problem solving !

Mental Burden on the Programmer

- Value Restriction = Only "values" are generalized at LET
- What are "values"?
  - variables: YES
  - functions: YES
  - Applications: NO
  - Constructor calls: YES
    - Ah wait... except ref ... NO
  - There can be a lot more to remember ...
- The trouble of remembering those can be a lot more than that of writing type annotations!
- They have nothing to do with the programmer's problem solving !

Can we do better?

The Real Problem

∀a. (ref(a->a))

let r = ref (λx.x) in
(r := λx.x+1; (!r)true)

The Real Problem

real problem: this type should be ref (∀a.a->a), but let-polymorphism infers ∀a.(ref (a->a))

∀a. (ref(a->a))

```
let r = ref (λx.x) in
(r := λx.x+1; (!r)true)
```

The Real Problem

real problem: this type should be ref (∀a.a->a), but let-polymorphism infers ∀a.(ref (a->a))

ref (∀a.(a->a))

let r = ref (λx.x) in
(r := λx.x+1; (!r)true)

The Real Problem

real problem: this type should be ref (∀a.a->a), but let-polymorphism infers ∀a.(ref (a->a))

ref (∀a.(a->a))

let r = ref (λx.x) in
(r := λx.x+1; (!r)true)

real error:
- trying to assign int->int into ref (∀a.a->a)
- int->int is not a subtype of ∀a.a->a

The Real Problem

real problem: this type should be ref (∀a.a->a), but let-polymorphism infers ∀a.(ref (a->a))

ref (∀a.(a->a))

let r = ref (λx.x) in
(r := λx.x+1; (!r)true)

real error:
- trying to assign int->int into ref (∀a.a->a)
- int->int is not a subtype of ∀a.a->a

Won't even reach this point!

The Real Problem

real problem: this type should be ref (∀a.a->a), but let-polymorphism infers ∀a.(ref (a->a))

ref (∀a.(a->a))

Can we type it as ref (∀a.a->a) ?

let r = ref (λx.x) in

(r := λx.x+1; (!r)true)

real error:
- trying to assign int->int into ref (∀a.a->a)
- int->int is not a subtype of ∀a.a->a

Won't even reach this point!

Generalization at λ (Genλ)

Generalization at λ (Genλ)

* Essence of type constraints: record how parameters are used in the function body

Generalization at λ (Genλ)

* Essence of type constraints: record how parameters are **used** in the function **body**
* If a type variable should not be further constrained, then it should be **generalized**

Generalization at λ (Genλ)

* Essence of type constraints: record how parameters are used in the function body
* If a type variable should not be further constrained, then it should be generalized
* Generalization of a type variable means: the "door" to further constraining this type variable is "closed"

Generalization at λ (Genλ)

* Essence of type constraints: record how parameters are used in the function body
* If a type variable should not be further constrained, then it should be generalized
* Generalization of a type variable means: the "door" to further constraining this type variable is "closed"
* The caller should not constrain the parameter type, only the function definition can

Generalization at λ (Genλ)

* Essence of type constraints: record how parameters are **used** in the function **body**
* If a type variable should not be further constrained, then it should be **generalized**
* **Generalization of a type variable** means: the "**door**" to further constraining this type variable is "**closed**"
* The **caller** should not constrain the parameter type, only the **function definition** can
* So the door to constraining the type of parameters should be "closed" at **function boundaries**

Generalization at λ (Genλ)

* Essence of type constraints: record how parameters are **used** in the function **body**
* If a type variable should not be further constrained, then it should be **generalized**
* **Generalization of a type variable** means: the "**door**" to further constraining this type variable is "**closed**"
* The **caller** should not constrain the parameter type, only the **function definition** can
* So the door to constraining the type of parameters should be "closed" at **function boundaries**
* This means: generalize should happen at **λ**, and not **LET**

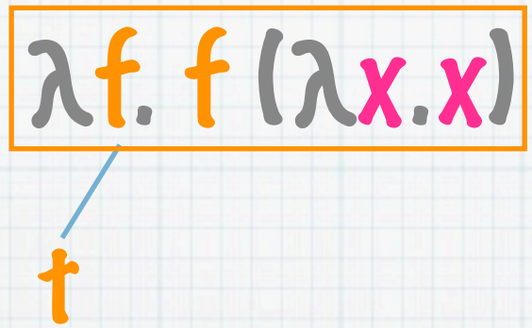
How does Genλ happen?

λf. f (λx.x)

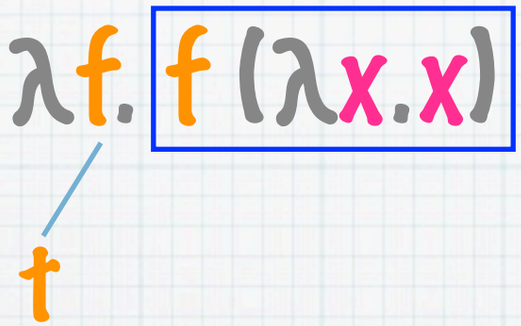
How does Genλ happen?

$$\lambda f.\ f\ (\lambda x.x)$$

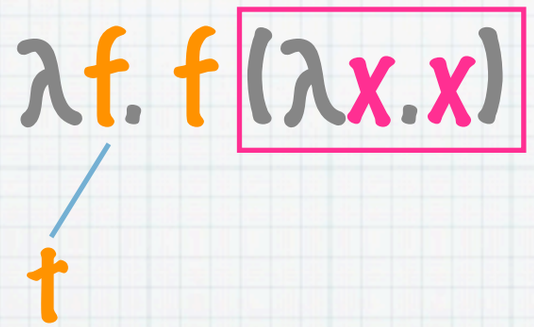
How does Genλ happen?



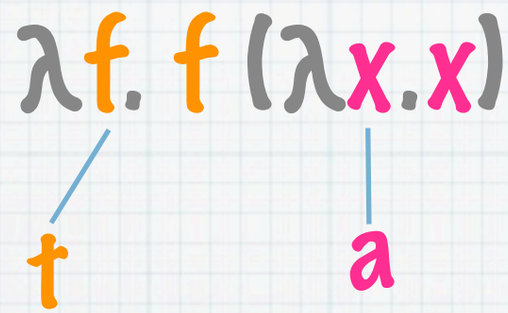
How does Genλ happen?



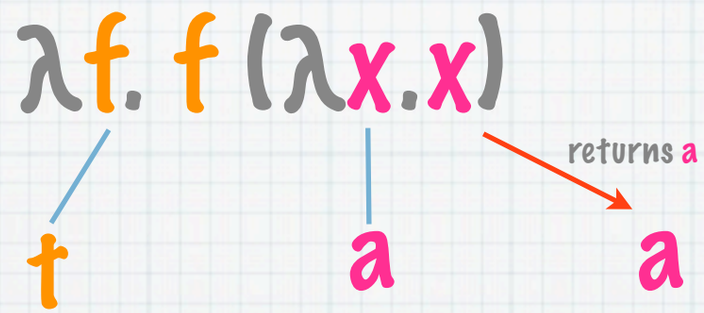
How does Genλ happen?



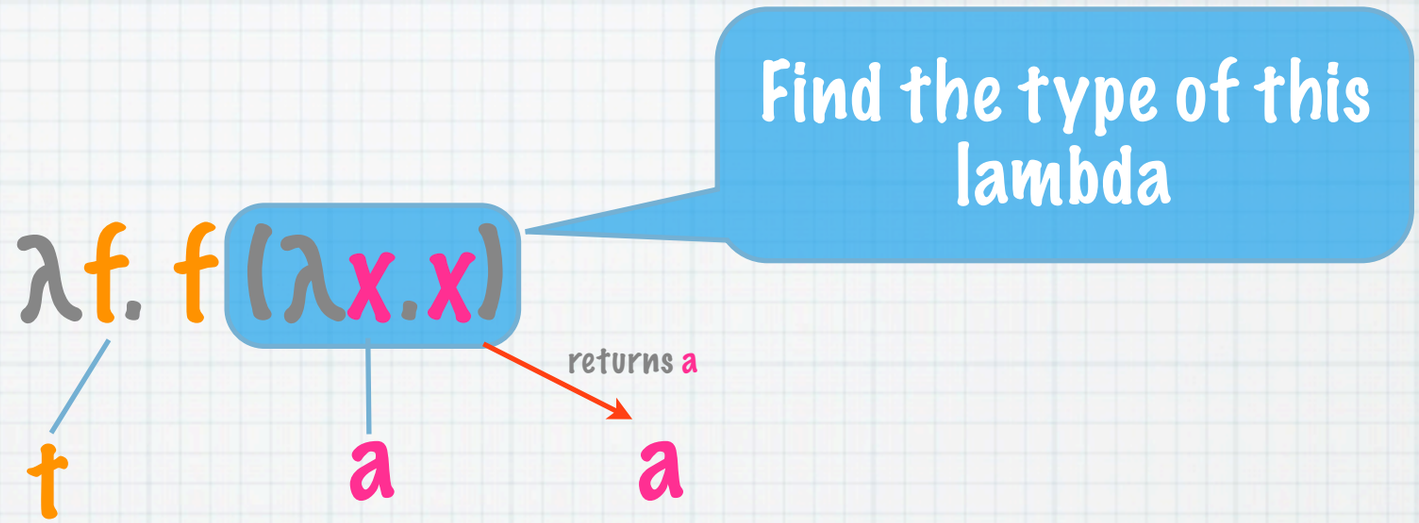
How does Genλ happen?



How does Genλ happen?



How does Genλ happen?



How does Genλ happen?

Find the type of this lambda

λf. f (λx.x)

f

a → a

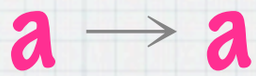
How does Gen happen?

Backing out of the scope of x

Find the type of this lambda

$$\lambda f.\ f\ (\lambda x.x)$$

$$f$$



How does Gen happen?

Backing out of the scope of x

Find the type of this lambda

$\lambda f.\ f\ (\lambda x.x)$

t

a can no longer be constrained because x is going out of scope and no outer references to a exists, so generalization happens

$a \rightarrow a$

How does Gen happen?

Backing out of the scope of x

Find the type of this lambda

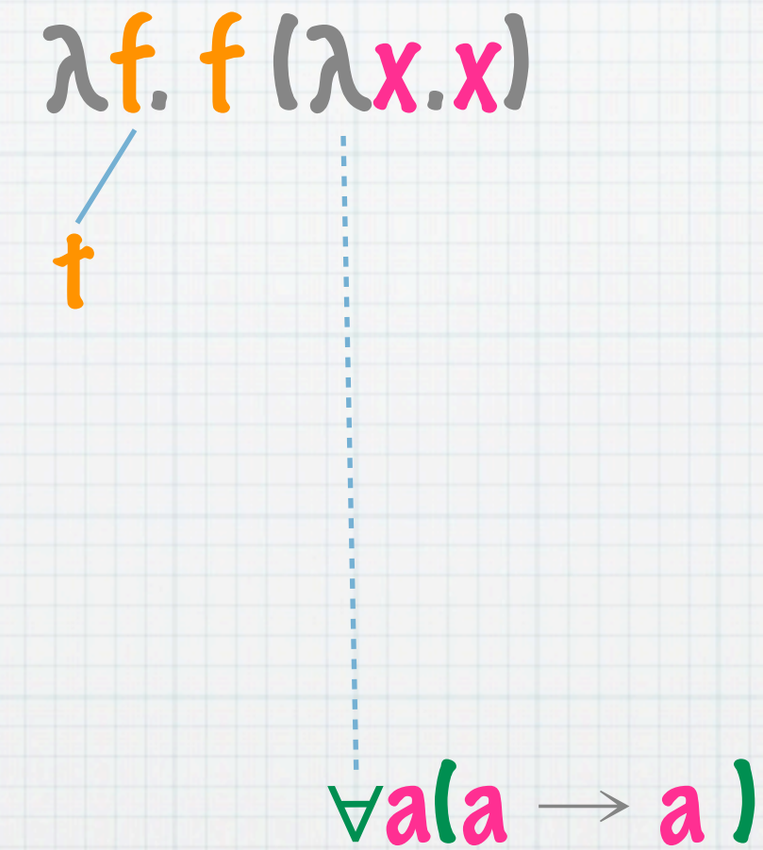
$\lambda f.\ f\ (\lambda x.x)$

f

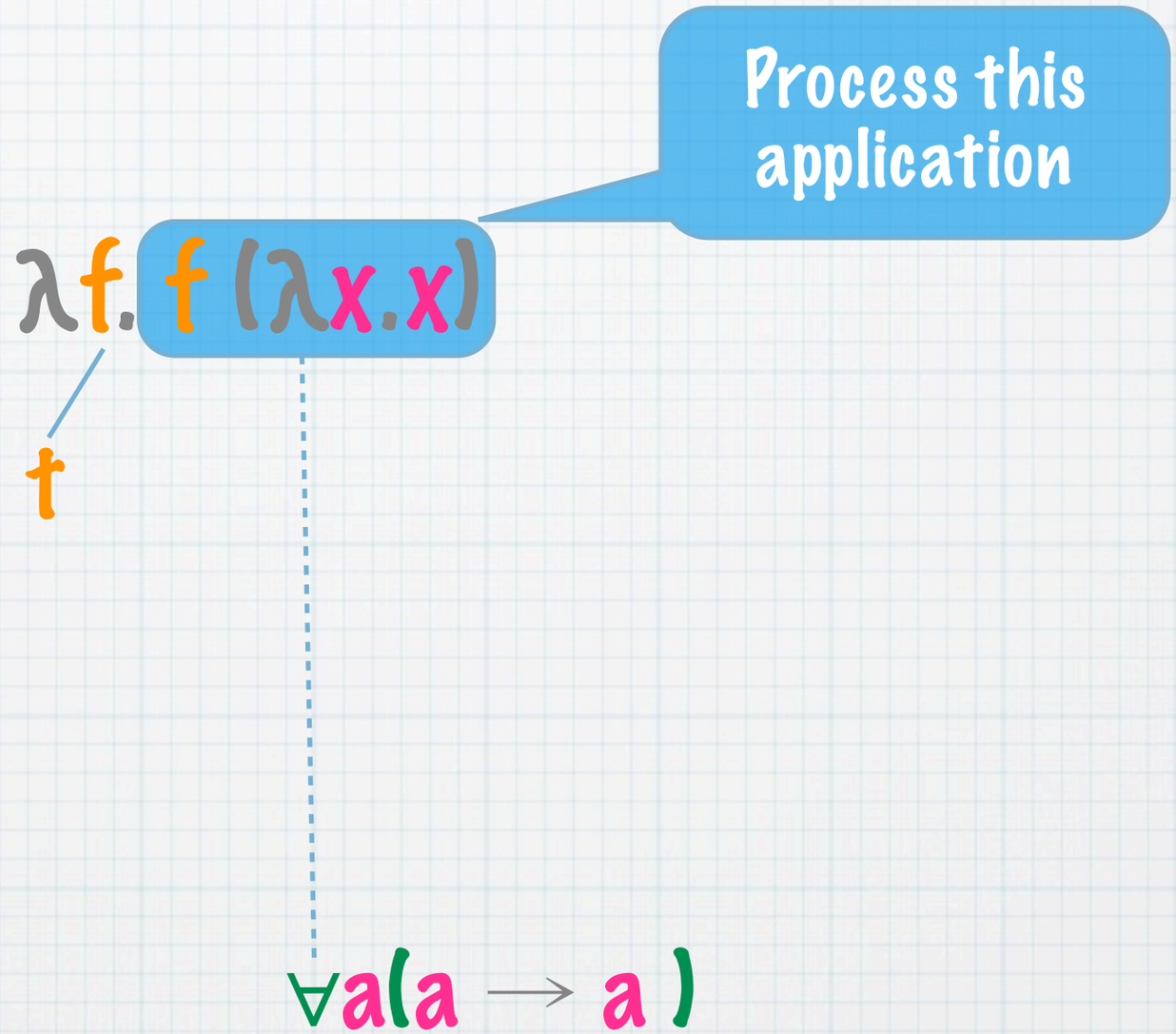
a can no longer be constrained because x is going out of scope and no outer references to a exists, so generalization happens

$\forall a (a \rightarrow a)$

How does Genλ happen?



How does Genλ happen?

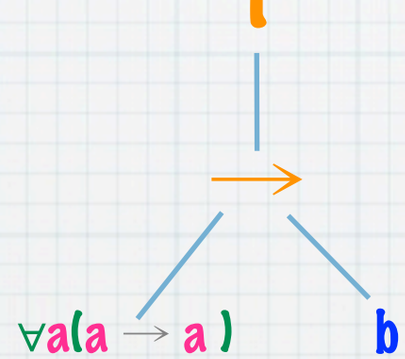


How does Genλ happen?

Process this application

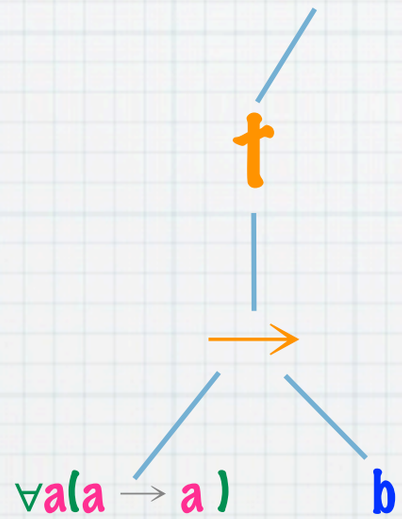
λf. f (λx.x)

t



How does Genλ happen?

λf. f (λx.x)



How does Genλ happen?

$$\lambda f.\ f\ (\lambda x.x)$$

$$((\forall a.a \to a) \to b) \to b$$

How does Genλ happen?

$$\lambda f.\ f\ (\lambda x.x)$$

b can no longer be constrained

$$((\forall a.a \to a) \to b) \to b$$

How does Genλ happen?

$$\lambda f.\ f\ (\lambda x.x)$$

$$\forall b.((((\forall a.a \to a) \to b) \to b) \to b)$$

How does Genλ happen?

λf. f (λx.x)

∀b.(((∀a.a → a) → b) → b)

This is a "higher ranked" type

How does Genλ happen?

$$\lambda f.\ f\ (\lambda x.x)$$

$$\forall b.((((\forall a.a \to a) \to b) \to b) \to b)$$

How does Genλ happen?

$$\lambda f.\ f\ (\lambda x.x)$$

$$\forall b.(((\forall a.a \to a) \to b) \to b)$$

How will HM type this?

$$\forall b.(((\forall a.a \to a) \to b) \to b)$$

How does Genλ happen?

$$\lambda f.\ f\ (\lambda x.x)$$

$$\forall b.((((\forall a.a \to a) \to b) \to b) \to b)$$

How will HM type this?

$$\forall a.\forall b.((((a \to a) \to b) \to b) \to b)$$

The Difference of Two Types

Gen_λ

$$\forall b.(((\forall a.a \to a) \to b) \to b)$$

ML

$$\forall a.\forall b.(((a \to a) \to b) \to b)$$

The Difference of Two Types

Gen$_\lambda$

$$\forall b.(((\forall a.a \to a) \to b) \to b)$$

ML

$$\forall a.\forall b.(((a \to a) \to b) \to b)$$

★ Gen$_\lambda$ type contains more information

★ We can know from the type that the argument will be applied to a polymorphic function

The Difference of Two Types

Gen$_\lambda$

$$\forall b.(((\forall a.a \to a) \to b) \to b)$$

ML

$$\forall a.\forall b.(((a \to a) \to b) \to b)$$

★ Gen$_\lambda$ type contains **more information**

★ We can know **from the type** that the argument **will be** applied to a **polymorphic function**

Is "more information" always good?

The Conflict between Expressiveness and Effectiveness

The Conflict between Expressiveness and Effectiveness

* Type inference is putting partial information about the program into its type

The Conflict between Expressiveness and Effectiveness

* Type inference is putting partial information about the program into its type
* We can always put more information of the program into its type

The Conflict between Expressiveness and Effectiveness

* Type inference is putting partial information about the program into its type
* We can always put more information of the program into its type
* The more information the type contains, the more likely the program can be typed (expressiveness)

The Conflict between Expressiveness and Effectiveness

* Type inference is putting partial information about the program into its type
* We can always put more information of the program into its type
* The more information the type contains, the more likely the program can be typed (expressiveness)
* The more information the types contain, the more computing power is needed to verify the compatibility of the types (effectiveness)

The Conflict between Expressiveness and Effectiveness

* Type inference is putting partial information about the program into its type
* We can always put more information of the program into its type
* The more information the type contains, the more likely the program can be typed (expressiveness)
* The more information the types contain, the more computing power is needed to verify the compatibility of the types (effectiveness)
* In the extreme case, the type can contain nearly all the program's information, and type checking becomes as expensive as running the program

The Conflict between Expressiveness and Effectiveness

* Type inference is putting partial information about the program into its type
* We can always put more information of the program into its type
* The more information the type contains, the more likely the program can be typed (expressiveness)
* The more information the types contain, the more computing power is needed to verify the compatibility of the types (effectiveness)
* In the extreme case, the type can contain nearly all the program's information, and type checking becomes as expensive as running the program

An example of such a system is intersection types

What are Intersection Types?

λf. (f 1, f true)

Is typed as: (→ a ∧ → b) → (a, b)

What are Intersection Types?

λf. (f 1, f true)

Is typed as: (int → a ∧ → b) → (a, b)

What are Intersection Types?

λf. (f 1, f true)

Is typed as: $(\mathsf{int} \to a \wedge \mathsf{bool} \to b) \to (a, b)$

What are Intersection Types?

λf. (f 1, f true) "f is simultaneously int → a and bool → b"

Is typed as: (int → a ∧ bool → b) → (a, b)

What are Intersection Types?

λf. (f 1, f true) "f is simultaneously int → a and bool → b"

Is typed as: (int → a ∧ bool → b) → (a, b)

When applied on λx.x : (λf. (f 1, f true)) (λx.x)

What are Intersection Types?

λf. (f 1, f true) "f is simultaneously int → a and bool → b"

Is typed as: (int → a ∧ bool → b) → (a, b)

When applied on λx.x : (λf. (f 1, f true)) (λx.x)

... type checked as (int, bool)

What are Intersection Types?

λf. (f 1, f true) "f is simultaneously int → a and bool → b"

Is typed as: (int → a ∧ bool → b) → (a, b)

When applied on λx.x : (λf. (f 1, f true)) (λx.x)

... type checked as (int, bool)

supposed to produce (1, true)

What are Intersection Types?

$\lambda f.\,(f\ 1,\ f\ \text{true})$ "f is simultaneously $\text{int} \to a$ and $\text{bool} \to b$"

Is typed as: $(\ \text{int} \to a \wedge \text{bool} \to b) \to (a, b)$

When applied on $\lambda x.x$: $(\lambda f.\,(f\ 1,\ f\ \text{true}))\ (\lambda x.x)$

The problem is, how to get (int, bool)
given $(\ \text{int} \to a \wedge \text{bool} \to b) \to (a, b)$
and $\forall a.a \to a$

What are Intersection Types?

Answer: "apply" $\forall a.a \to a$ twice,
once to **int**, the other time to **bool**

When applied on $\lambda x.x$: $(\lambda f.\ (f\ 1,\ f\ true))\ (\lambda x.x)$

The problem is, how to get (**int**, **bool**)
given (**int** $\to$ a $\wedge$ **bool** $\to$ b) $\to$ (a, b)
and $\forall a.a \to a$

The Rest of the Story

* I have only shown intersection types which doesn't capture the **control flow** information
* We can capture control flow information in the type, removing **idempotence**
* The resulting systems will be even more powerful, able to type self applications like $\lambda x.xx$ or $\lambda x.xxx$
* To be continued ...