

Polymorphism: What's the deal with 'a'?

Hype for Types

November 5, 2021

Polymorphism

Identity

Recall lambda abstraction from the Simply Typed Lambda Calculus

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda(x : \tau)e : \tau \rightarrow \tau'}$$

Identity

Recall lambda abstraction from the Simply Typed Lambda Calculus

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda(x : \tau)e : \tau \rightarrow \tau'}$$

Notice, \uparrow we must type annotate every lambda.

Let's write the identity function (assuming some reasonable base types).

Identity

Recall lambda abstraction from the Simply Typed Lambda Calculus

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda(x : \tau)e : \tau \rightarrow \tau'}$$

Notice, \uparrow we must type annotate every lambda.

Let's write the identity function (assuming some reasonable base types).

$id = \lambda(x : Nat)x$

But this only works on Nats!

$id\ true$ (* type error! *)

Identity

Recall lambda abstraction from the Simply Typed Lambda Calculus

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda(x : \tau)e : \tau \rightarrow \tau'}$$

Notice, \uparrow we must type annotate every lambda.

Let's write the identity function (assuming some reasonable base types).

$id = \lambda(x : Nat)x$

But this only works on Nats!

$id\ true$ (* type error! *)

$id2 = \lambda(x : Bool)x$

This seems really annoying >: (

What does SML do?

```
val id = fn (x : 'a) => x
val _ = id 1
val _ = id true
val _ = id "nice"

id : 'a -> 'a
```

What does SML do?

```
val id = fn (x : 'a) => x
val _ = id 1
val _ = id true
val _ = id "nice"

id : 'a -> 'a
```

But what *is* 'a? Is it a type?

What does SML do?

```
val id = fn (x : 'a) => x
val _ = id 1
val _ = id true
val _ = id "nice"

id : 'a -> 'a
```

But what *is* 'a? Is it a type?

If `id 1` type checks then `1 : 'a???`

Polymorphism

Intuitively, we'd like to interpret $'a \rightarrow 'a$ as "for all $'a$, $'a \rightarrow 'a$ "
The "for all" is *implicit*.

This is great for programming, but confusing to formalize.

Let's make it *explicit*!

$$'a \rightarrow 'a \implies \forall a. a \rightarrow a$$

The ticks are no longer needed, as we've explicitly bound a as a type variable.

Polymorphism

How do we construct a value of type $\forall a. a \rightarrow a$ in our new formalism?
We might suggest $\lambda(x : a)x$, but once again the type variable is being bound *implicitly*.

Polymorphism

How do we construct a value of type $\forall a.a \rightarrow a$ in our new formalism?
We might suggest $\lambda(x : a)x$, but once again the type variable is being bound *implicitly*.

Let's bind it *explicitly*!

$\Lambda(a : \text{Type})\lambda(x : a)x : \forall a.a \rightarrow a$

Polymorphism

How do we construct a value of type $\forall a.a \rightarrow a$ in our new formalism?
We might suggest $\lambda(x : a)x$, but once again the type variable is being bound *implicitly*.

Let's bind it *explicitly*!

$\Lambda(a : \text{Type})\lambda(x : a)x : \forall a.a \rightarrow a$

How do we use this?

Polymorphism

How do we construct a value of type $\forall a. a \rightarrow a$ in our new formalism?
We might suggest $\lambda(x : a)x$, but once again the type variable is being bound *implicitly*.

Let's bind it *explicitly*!

$\Lambda(a : \text{Type})\lambda(x : a)x : \forall a. a \rightarrow a$

How do we use this?

$(\Lambda(a : \text{Type})\lambda(x : a)x)[\text{Nat}] \implies \lambda(x : \text{Nat})x$

System F

The polymorphic lambda calculus we've developed is called System F.
Let's write a grammar!

System F

The polymorphic lambda calculus we've developed is called System F.
Let's write a grammar!

$e ::=$	x	term variable
	$ \lambda(x : \tau)e$	term abstraction
	$ \Lambda(t : \text{Type})e$	type abstraction
	$ e_1 e_2$	term application
	$ e_1[\tau]$	type application

$\tau ::=$	t	type variable
	$ \tau_1 \rightarrow \tau_2$	function type
	$ \forall t. \tau$	polymorphic type

System F

And some inference rules!

System F

And some inference rules!

$$\frac{t \in \Delta}{\Delta \vdash t \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type}}$$

$$\frac{\Delta, t \vdash \tau \text{ type}}{\Delta \vdash \forall t. \tau \text{ type}}$$

System F

And some inference rules!

$$\frac{t \in \Delta}{\Delta \vdash t \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type}}$$

$$\frac{\Delta, t \vdash \tau \text{ type}}{\Delta \vdash \forall t. \tau \text{ type}}$$

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta; \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ type}}{\Delta; \Gamma \vdash \lambda(x : \tau) e : \tau \rightarrow \tau'}$$

$$\frac{\Delta, t; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda(t : \text{Type}) e : \forall t. \tau}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Delta; \Gamma \vdash e : \forall t. \tau \quad \Delta \vdash \tau' \text{ type}}{\Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/t]}$$

System F

And some inference rules!

$$\frac{t \in \Delta}{\Delta \vdash t \text{ type}}$$

$$\frac{\Delta \vdash \tau_1 \text{ type} \quad \Delta \vdash \tau_2 \text{ type}}{\Delta \vdash \tau_1 \rightarrow \tau_2 \text{ type}}$$

$$\frac{\Delta, t \vdash \tau \text{ type}}{\Delta \vdash \forall t. \tau \text{ type}}$$

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau}$$

$$\frac{\Delta; \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \text{ type}}{\Delta; \Gamma \vdash \lambda(x : \tau) e : \tau \rightarrow \tau'}$$

$$\frac{\Delta, t; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda(t : \text{Type}) e : \forall t. \tau}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash e_1 e_2 : \tau'}$$

$$\frac{\Delta; \Gamma \vdash e : \forall t. \tau \quad \Delta \vdash \tau' \text{ type}}{\Delta; \Gamma \vdash e[\tau'] : \tau[\tau'/t]}$$

Question

Do we need anything else? What about product types? Sum types?

Some Fing Functions

$$\text{swap} : \forall a b c. (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c) =$$

Some Fing Functions

$$\text{swap} : \forall a b c. (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c) =$$
$$\Lambda(a b c : \text{Type}) \lambda(f : a \rightarrow b \rightarrow c) \lambda(x : b) \lambda(y : a) f y x$$

Some Fing Functions

$$\begin{aligned} \text{swap} &: \forall a b c. (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c) = \\ &\Lambda(a b c : \text{Type}) \lambda(f : a \rightarrow b \rightarrow c) \lambda(x : b) \lambda(y : a) f y x \\ \text{compose} &: \forall a b c. (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c) = \end{aligned}$$

Some Fing Functions

$$\begin{aligned} \text{swap} &: \forall a b c. (a \rightarrow b \rightarrow c) \rightarrow (b \rightarrow a \rightarrow c) = \\ &\Lambda(a b c : \text{Type})\lambda(f : a \rightarrow b \rightarrow c)\lambda(x : b)\lambda(y : a)f y x \\ \text{compose} &: \forall a b c. (a \rightarrow b) \rightarrow (b \rightarrow c) \rightarrow (a \rightarrow c) = \\ &\Lambda(a b c : \text{Type})\lambda(f : a \rightarrow b)\lambda(g : b \rightarrow c)\lambda(x : a)g(f x) \end{aligned}$$

Does SML implement System F?

Is the polymorphism of SML equivalent to the polymorphism of System F?

Is $'a \rightarrow 'a$ always really $\forall a.a \rightarrow a$?

Does SML implement System F?

Is the polymorphism of SML equivalent to the polymorphism of System F?

Is $'a \rightarrow 'a$ always really $\forall a.a \rightarrow a$?

Consider:

```
fun hmm (id : 'a -> 'a) = (id 1, id true)
```

Does SML implement System F?

Is the polymorphism of SML equivalent to the polymorphism of System F?

Is $'a \rightarrow 'a$ always really $\forall a.a \rightarrow a$?

Consider:

```
fun hmm (id : 'a -> 'a) = (id 1, id true)
```

Type error! In SML, big lambdas can only be present at *declarations*, not arbitrarily inside expressions.

Our function here is equivalent to:

$$hmm = \Lambda(a : \text{Type})\lambda(id : a \rightarrow a)(id\ 1, id\ true)$$

Which is *not* the same as:

$$hmm = \lambda(id : \forall a.a \rightarrow a)(id[int]\ 1, id[bool]\ true)$$

Why? Because type inference for System F is undecidable!

For all? Exists

If we can express "for all" as a type, can we express "there exists" as a type?

For all? Exists

If we can express "for all" as a type, can we express "there exists" as a type?

$\forall t.t \rightarrow t$ means "for any type t , if you give me a t , I'll give you a t "

$\exists t.t \rightarrow t$ means "there is some *specific* type t , and if you give me a t , I'll give you a t "

For all? Exists

If we can express "for all" as a type, can we express "there exists" as a type?

$\forall t.t \rightarrow t$ means "for any type t , if you give me a t , I'll give you a t "

$\exists t.t \rightarrow t$ means "there is some *specific* type t , and if you give me a t , I'll give you a t "

Where have you seen the idea of specific, yet unknown type?

For all? Exists

If we can express "for all" as a type, can we express "there exists" as a type?

$\forall t.t \rightarrow t$ means "for any type t , if you give me a t , I'll give you a t "

$\exists t.t \rightarrow t$ means "there is some *specific* type t , and if you give me a t , I'll give you a t "

Where have you seen the idea of specific, yet unknown type?

Modules!

Existentialism

```
signature S =  
  sig  
    type t  
    val x : t  
    val f : t -> t  
  end
```

is basically equivalent to:

$$\exists t. \{x : t, f : t \rightarrow t\}$$

or even more simply:

$$\exists t. t \times (t \rightarrow t)$$

Da Rules

$$\frac{\Delta, t \vdash \tau \text{ type}}{\Delta \vdash \exists t. \tau \text{ type}}$$

$$\frac{\Delta; \Gamma \vdash e : [\rho/t]\tau \quad \Delta \vdash \rho \text{ type}}{\Delta; \Gamma \vdash \text{struct type } t = \rho \text{ in } e : \exists t. \tau}$$

$$\frac{\Delta; \Gamma \vdash M : \exists t. \tau \quad \Delta, t; \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau' \text{ type}}{\Delta; \Gamma \vdash \text{open } M \text{ as } t, x \text{ in } e : \tau'}$$

Practical Uses

Stack =

Practical Uses

Stack =

$\exists t. \{ \text{empty} : t, \text{push} : \text{int} \rightarrow t \rightarrow t, \text{pop} : t \rightarrow (\text{int} \times t) \text{ option} \}$

EvenStack =

Practical Uses

Stack =

$\exists t. \{ \text{empty} : t, \text{push} : \text{int} \rightarrow t \rightarrow t, \text{pop} : t \rightarrow (\text{int} \times t) \text{ option} \}$

EvenStack =

$\exists t. \{ \text{empty} : t, \text{push} : \text{int} \rightarrow \text{int} \rightarrow t \rightarrow t, \text{pop} : t \rightarrow (\text{int} \times t) \text{ option} \}$

Practical Uses

ListStack : *Stack* =

Practical Uses

ListStack : Stack = struct type t = int list in

{empty = Nil,

push = Cons,

pop = λ(s : int list) case s of Nil ⇒ None | Cons(x, xs) ⇒ Some(x, xs)}

Practical Uses

$mkEvenStack : Stack \rightarrow EvenStack =$

Practical Uses

$mkEvenStack : Stack \rightarrow EvenStack =$

$\lambda(S : Stack)open\ S\ as\ t, s\ in$

$struct\ type\ t' = t\ in$

$\{empty = s.empty,$

$push = \lambda(x : int)\lambda(y : int)s.push\ y \circ s.push\ x,$

$pop = s.pop\}$

Function Types? No: Type Functions

In SML I can write

```
type 'a f = 'a * 'a
```

What is `f`? Does it have a type?

Function Types? No: Type Functions

In SML I can write

```
type 'a f = 'a * 'a
```

What is `f`? Does it have a type?

In simply typed lambda calculus, we can write functions from *terms* to *terms*: $\lambda(x : \text{Nat})x$

Function Types? No: Type Functions

In SML I can write

```
type 'a f = 'a * 'a
```

What is `f`? Does it have a type?

In simply typed lambda calculus, we can write functions from *terms* to *terms*: $\lambda(x : \text{Nat})x$

In System F we can write functions from *types* to *terms*:

$\Lambda(A : \text{Type})\lambda(x : A)x$

Function Types? No: Type Functions

In SML I can write

```
type 'a f = 'a * 'a
```

What is f ? Does it have a type?

In simply typed lambda calculus, we can write functions from *terms* to *terms*: $\lambda(x : Nat)x$

In System F we can write functions from *types* to *terms*:

$\Lambda(A : Type)\lambda(x : A)x$

f is a function from a *type* to a *type*. $f : Type \rightarrow Type$.

Function Types? No: Type Functions

In SML I can write

```
type 'a f = 'a * 'a
```

What is f ? Does it have a type?

In simply typed lambda calculus, we can write functions from *terms* to *terms*: $\lambda(x : \text{Nat})x$

In System F we can write functions from *types* to *terms*:

$$\Lambda(A : \text{Type})\lambda(x : A)x$$

f is a function from a *type* to a *type*. $f : \text{Type} \rightarrow \text{Type}$.

In SML we're limited to $\text{Type} \rightarrow \text{Type}$, but we could go further.

In System F_ω , we can write functions like:

$$\lambda(F : \text{Type} \rightarrow \text{Type})\lambda(A : \text{Type})(A \times A) \rightarrow F A$$

A Curious Observation

Do \wedge and \forall seem conceptually similar to any language features we've already seen?

A Curious Observation

Do Λ and \forall seem conceptually similar to any language features we've already seen?

Λ functions much like λ , but instead of taking a *term*, it takes a *type*.

\forall and \rightarrow seem related in the same sort of way.

$$\forall t. \tau \equiv (t : \text{Type}) \rightarrow \tau$$

$$\Lambda(t)e \equiv \lambda(t : \text{Type})e$$

A Curious Observation

Do Λ and \forall seem conceptually similar to any language features we've already seen?

Λ functions much like λ , but instead of taking a *term*, it takes a *type*.

\forall and \rightarrow seem related in the same sort of way.

$$\forall t. \tau \equiv (t : \text{Type}) \rightarrow \tau \qquad \Lambda(t)e \equiv \lambda(t : \text{Type})e$$

Do *struct type* $t = \rho$ in e and \exists remind you of anything?

A Curious Observation

Do Λ and \forall seem conceptually similar to any language features we've already seen?

Λ functions much like λ , but instead of taking a *term*, it takes a *type*.

\forall and \rightarrow seem related in the same sort of way.

$$\forall t. \tau \equiv (t : \text{Type}) \rightarrow \tau$$

$$\Lambda(t)e \equiv \lambda(t : \text{Type})e$$

Do *struct type* $t = \rho$ in e and \exists remind you of anything?

Our module expressions are really just *tuples* of a *type*, and a term that uses that type!

A Curious Observation

Do Λ and \forall seem conceptually similar to any language features we've already seen?

Λ functions much like λ , but instead of taking a *term*, it takes a *type*.

\forall and \rightarrow seem related in the same sort of way.

$$\forall t. \tau \equiv (t : \text{Type}) \rightarrow \tau$$

$$\Lambda(t)e \equiv \lambda(t : \text{Type})e$$

Do *struct type* $t = \rho$ in e and \exists remind you of anything?

Our module expressions are really just *tuples* of a *type*, and a term that uses that type!

$$\exists t. \tau \equiv (t : \text{Type}) \times \tau$$

$$\text{struct type } t = \rho \text{ in } e \equiv \langle \rho, e \rangle$$

This is how we'd express these concepts in a language where we can treat *types* like *terms*!

We don't need no type constructors (except \forall and \rightarrow)

Can we encode $A \times B$ in System F?

We don't need no type constructors (except \forall and \rightarrow)

Can we encode $A \times B$ in System F? Yes! But How?

We don't need no type constructors (except \forall and \rightarrow)

Can we encode $A \times B$ in System F? Yes! But How?

What can you do with a value of type $A \times B$?

We don't need no type constructors (except \forall and \rightarrow)

Can we encode $A \times B$ in System F? Yes! But How?

What can you do with a value of type $A \times B$?

Well, if we have a function that requires a value of type A and a value of type B , then we can provide it arguments.

We don't need no type constructors (except \forall and \rightarrow)

Can we encode $A \times B$ in System F? Yes! But How?

What can you do with a value of type $A \times B$?

Well, if we have a function that requires a value of type A and a value of type B , then we can provide it arguments.

$$A \times B = \forall R. (A \rightarrow B \rightarrow R) \rightarrow R$$

$$\text{pair} : \forall A B. A \rightarrow B \rightarrow \forall R. (A \rightarrow B \rightarrow R) \rightarrow R =$$

$$\Lambda(A B) \lambda(x : A) \lambda(y : B) \Lambda(R) \lambda(f : A \rightarrow B \rightarrow R) f \ x \ y$$

$$\text{fst} : \forall A B. (\forall R. (A \rightarrow B \rightarrow R) \rightarrow R) \rightarrow A =$$

$$\Lambda(A B) \lambda(p : \forall R. A \rightarrow B \rightarrow R) p[A](\lambda(x : A) \lambda(y : B) x)$$

$$\text{snd} : \forall A B. (\forall R. (A \rightarrow B \rightarrow R) \rightarrow R) \rightarrow B =$$

$$\Lambda(A B) \lambda(p : \forall R. A \rightarrow B \rightarrow R) p[B](\lambda(x : A) \lambda(y : B) y)$$

Sum Types?

What can we do with a value of type $A + B$?

Sum Types?

What can we do with a value of type $A + B$?

If we can a function that takes an A and a function that takes a B , we can definitely provide an argument to *one* of them.

Sum Types?

What can we do with a value of type $A + B$?

If we can a function that takes an A and a function that takes a B , we can definitely provide an argument to *one* of them.

$$A + B = \forall R. (A \rightarrow R) \rightarrow (B \rightarrow R) \rightarrow R$$

$$\text{left} : \forall A B. A \rightarrow \forall R. (A \rightarrow R) \rightarrow (B \rightarrow R) \rightarrow R =$$

$$\Lambda(A B)\lambda(x : A)\Lambda(R)\lambda(\text{left} : A \rightarrow R)\lambda(\text{right} : B \rightarrow R)\text{left } x$$

$$\text{right} : \forall A B. B \rightarrow \forall R. (A \rightarrow R) \rightarrow (B \rightarrow R) \rightarrow R =$$

$$\Lambda(A B)\lambda(x : B)\Lambda(R)\lambda(\text{left} : A \rightarrow R)\lambda(\text{right} : B \rightarrow R)\text{right } x$$

Sum Types?

What can we do with a value of type $A + B$?

If we can a function that takes an A and a function that takes a B , we can definitely provide an argument to *one* of them.

$$A + B = \forall R. (A \rightarrow R) \rightarrow (B \rightarrow R) \rightarrow R$$

$$\text{left} : \forall A B. A \rightarrow \forall R. (A \rightarrow R) \rightarrow (B \rightarrow R) \rightarrow R =$$

$$\Lambda(A B)\lambda(x : A)\Lambda(R)\lambda(\text{left} : A \rightarrow R)\lambda(\text{right} : B \rightarrow R)\text{left } x$$

$$\text{right} : \forall A B. B \rightarrow \forall R. (A \rightarrow R) \rightarrow (B \rightarrow R) \rightarrow R =$$

$$\Lambda(A B)\lambda(x : B)\Lambda(R)\lambda(\text{left} : A \rightarrow R)\lambda(\text{right} : B \rightarrow R)\text{right } x$$

What about case?

An encoded value of type $A + B$ *is already* a case!