Polymorphism: What's the deal with 'a?

Hype for Types

March 31, 2022

Polymorphism

Identity

Recall lambda abstraction from the Simply Typed Lambda Calculus

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

Identity

Recall lambda abstraction from the Simply Typed Lambda Calculus

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

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

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

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

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

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 -> 'a` as "for all `'a`, `'a -> 'a`"

The "for all" is *implicit*.

This is great for programming, but confusing to formalize.

Let's make it *explicit*!

`'a -> 'a` $\Longrightarrow \forall a.a \to a$

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 $\forall a. a \to a$ 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 $\forall a.a \to a$ 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*!
$\Lambda(a : \text{Type})\lambda(x : a)x : \forall a.a \to a$

Polymorphism

How do we construct a value of type $\forall a.a \to a$ 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*!
$\Lambda(a : \text{Type})\lambda(x : a)x : \forall a.a \to a$
How do we use this?

Polymorphism

How do we construct a value of type $\forall a.a \to a$ 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*!
$\Lambda(a : \text{Type})\lambda(x : a)x : \forall a.a \to a$
How do we use this?
$(\Lambda(a : \text{Type})\lambda(x : a)x)[Nat] \Longrightarrow \lambda(x : Nat)x$

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!

$$
\begin{array}{rcll}
e & ::= & x & \text{term variable} \\
 & \mid & \lambda(x : \tau)e & \text{term abstraction} \\
 & \mid & \Lambda(t : \text{Type})e & \text{type abstraction} \\
 & \mid & e_1\, e_2 & \text{term application} \\
 & \mid & e_1[\tau] & \text{type application}
\end{array}
$$

$$
\begin{array}{rcll}
\tau & ::= & t & \text{type variable} \\
 & \mid & \tau_1 \to \tau_2 & \text{function type} \\
 & \mid & \forall t.\tau & \text{polymorphic type}
\end{array}
$$

System F

And some inference rules!

System F

And some inference rules!

$$\frac{t \in \Delta}{\Delta \vdash t \ \ \mathit{type}} \qquad \frac{\Delta \vdash \tau_1 \ \ \mathit{type} \quad \Delta \vdash \tau_2 \ \ \mathit{type}}{\Delta \vdash \tau_1 \to \tau_2 \ \ \mathit{type}} \qquad \frac{\Delta, t \vdash \tau \ \ \mathit{type}}{\Delta \vdash \forall t.\tau \ \ \mathit{type}}$$

System F

And some inference rules!

$$\frac{t \in \Delta}{\Delta \vdash t \ \mathit{type}} \qquad \frac{\Delta \vdash \tau_1 \ \mathit{type} \quad \Delta \vdash \tau_2 \ \mathit{type}}{\Delta \vdash \tau_1 \to \tau_2 \ \mathit{type}} \qquad \frac{\Delta, t \vdash \tau \ \mathit{type}}{\Delta \vdash \forall t.\tau \ \mathit{type}}$$

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \qquad \frac{\Delta; \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \ \mathit{type}}{\Delta; \Gamma \vdash \lambda(x : \tau)e : \tau \to \tau'}$$

$$\frac{\Delta, t; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda(t : \mathrm{Type})e : \forall t.\tau} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash e_1 e_2 : \tau'}$$

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

System F

And some inference rules!

$$\frac{t \in \Delta}{\Delta \vdash t \;\; \mathit{type}} \qquad \frac{\Delta \vdash \tau_1 \;\; \mathit{type} \quad \Delta \vdash \tau_2 \;\; \mathit{type}}{\Delta \vdash \tau_1 \to \tau_2 \;\; \mathit{type}} \qquad \frac{\Delta, t \vdash \tau \;\; \mathit{type}}{\Delta \vdash \forall t.\tau \;\; \mathit{type}}$$

$$\frac{x : \tau \in \Gamma}{\Delta; \Gamma \vdash x : \tau} \qquad \frac{\Delta; \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \;\; \mathit{type}}{\Delta; \Gamma \vdash \lambda(x : \tau)e : \tau \to \tau'}$$

$$\frac{\Delta, t; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda(t : \mathrm{Type})e : \forall t.\tau} \qquad \frac{\Delta; \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta; \Gamma \vdash e_2 : \tau}{\Delta; \Gamma \vdash e_1 e_2 : \tau'}$$

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

Question

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

Some Fing Functions

$$swap : \forall a\ b\ c.(a \to b \to c) \to (b \to a \to c) =$$

Some Fing Functions

$$swap : \forall a\ b\ c.(a \to b \to c) \to (b \to a \to c) =$$

$$\Lambda(a\ b\ c : \mathrm{Type})\lambda(f : a \to b \to c)\lambda(x : b)\lambda(y : a)f\ \ y\ \ x$$

Some Fing Functions

$$swap : \forall a\ b\ c.(a \to b \to c) \to (b \to a \to c) =$$

$$\Lambda(a\ b\ c : \text{Type})\lambda(f : a \to b \to c)\lambda(x : b)\lambda(y : a)f\ \ y\ \ x$$

$$compose : \forall a\ b\ c.(a \to b) \to (b \to c) \to (a \to c) =$$

Some Fing Functions

$$swap : \forall a\ b\ c.(a \to b \to c) \to (b \to a \to c) =$$

$$\Lambda(a\ b\ c : \text{Type})\lambda(f : a \to b \to c)\lambda(x : b)\lambda(y : a)f\ y\ x$$

$$compose : \forall a\ b\ c.(a \to b) \to (b \to c) \to (a \to c) =$$

$$\Lambda(a\ b\ c : \text{Type})\lambda(f : a \to b)\lambda(g : b \to c)\lambda(x : a)g(f\ x)$$

Does SML implement System F?

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

Is `'a -> '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 -> ’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 -> 'a` always really $\forall a.a \to a$?
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:

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

Which is *not* the same as:

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

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?
$\forall t.t \to t$ means "for any type t, if you give me a t, I'll give you a t
$\exists t.t \to t$ 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:

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

or even more simply:

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

Da Rules

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

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

Practical Uses

$$\mathit{Stack} =$$

Practical Uses

$$Stack =$$

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

$$EvenStack =$$

Practical Uses

$$Stack =$$

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

$$EvenStack =$$

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

Practical Uses

$$ListStack : Stack =$$

Practical Uses

$$ListStack : Stack = struct\ \ type\ \ t = int\ \ list\ \ in$$

$$\{empty = Nil,$$

$$push = Cons,$$

$$pop = \lambda(s : int\ \ list) case\ \ s\ \ of\ \ Nil \Rightarrow None | Cons(x, xs) \Rightarrow Some(x, xs)\}$$

Practical Uses

$$mkEvenStack : Stack \to EvenStack =$$

Practical Uses

$$mkEvenStack : Stack \to EvenStack =$$

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

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

$$\{empty = s.empty,$$

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

$$pop = s.pop\}$$

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*: $\lambda(x : Nat)x$

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*: $\lambda(x : Nat)x$
In System F we can write functions from *types* to *terms*:
$\Lambda(A : Type)\lambda(x : A)x$

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`: $Type \to Type$.

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*: $\lambda(x : Nat)x$
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`: $Type \to Type$.
In SML we're limited to $Type \to Type$, but we could go further.
In System F_ω, we can write functions like:
$\lambda(F : Type \to Type)\lambda(A : Type)(A \times A) \to F\ A$

A Curious Observation

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

A Curious Observation

Do Λ and ∀ 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*.

∀ and → seem related in the same sort of way.

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

A Curious Observation

Do Λ and ∀ 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*.

∀ and → seem related in the same sort of way.

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

Do $\mathit{struct\ type}\ t = \rho\ \mathit{in}\ e$ and ∃ remind you of anything?

A Curious Observation

Do Λ and ∀ 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*.

∀ and → seem related in the same sort of way.

$$\forall t.\tau \equiv (t : Type) \to \tau \qquad \Lambda(t)e \equiv \lambda(t : Type)e$$

Do $struct\ type\ t = \rho\ in\ e$ and ∃ 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.

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

Do $\mathit{struct}\ \mathit{type}\ t = \rho\ \mathit{in}\ e$ and $\exists$ remind you of anything?
Our module expressions are really just *tuples* of a *type*, and a term that uses that type!

$$\exists t.\tau \equiv (t : \mathit{Type}) \times \tau \qquad\qquad \mathit{struct}\ \mathit{type}\ t = \rho\ \mathit{in}\ e \equiv \langle \rho, e \rangle$$

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 $\to$)

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

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

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

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

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 ∀ and →)

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.

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

$$pair : \forall A\ B.A \to B \to \forall R.(A \to B \to R) \to R =$$

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

$$fst : \forall A\ B.(\forall R.(A \to B \to R) \to R) \to A =$$

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

$$snd : \forall A\ B.(\forall R.(A \to B \to R) \to R) \to B =$$

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

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.

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

$$\mathit{left} : \forall A\ B.A \to \forall R.(A \to R) \to (B \to R) \to R =$$

$$\Lambda(A\ B)\lambda(x : A)\Lambda(R)\lambda(\mathit{left} : A \to R)\lambda(\mathit{right} : B \to R)\mathit{left}\ x$$

$$\mathit{right} : \forall A\ B.B \to \forall R.(A \to R) \to (B \to R) \to R =$$

$$\Lambda(A\ B)\lambda(x : A)\Lambda(R)\lambda(\mathit{left} : B \to R)\lambda(\mathit{right} : B \to R)\mathit{right}\ x$$

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.

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

$$\mathit{left} : \forall A\ B.A \to \forall R.(A \to R) \to (B \to R) \to R =$$

$$\Lambda(A\ B)\lambda(x : A)\Lambda(R)\lambda(\mathit{left} : A \to R)\lambda(\mathit{right} : B \to R)\mathit{left}\ x$$

$$\mathit{right} : \forall A\ B.B \to \forall R.(A \to R) \to (B \to R) \to R =$$

$$\Lambda(A\ B)\lambda(x : A)\Lambda(R)\lambda(\mathit{left} : B \to R)\lambda(\mathit{right} : B \to R)\mathit{right}\ x$$

What about case?

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