Polymorphism: What's the deal with 'a?

Hype for Types

March 12, 2024

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 how 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 how 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 how 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 $\forall a.a \rightarrow a$ 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 $\forall a.a \to a$ 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!

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

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!

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

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

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

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

Question

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

Some F-ing Functions

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

Some F-ing Functions

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

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

Some F-ing 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 F-ing 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 $\forall a.a \to a$?
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!

What about exists?

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

What about 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`"

So $\exists t.t \to t$ should probably mean "there is some *specific* type `t`, and if you give me that `t`, I'll give you a `t`"

What about 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`"

So $\exists t.t \to t$ should probably mean "there is some *specific* type `t`, and if you give me that `t`, I'll give you a `t`"

Question

Does this sound similar to anything in SML?

Existentialism == Modules!

```
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:

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

Existentialism == Modules!

```
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:

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

Main Idea

We use **signatures** to represent **existential types**!

Existentialism == Modules!

Question

What is a value of type $\exists t.\tau$?

Existentialism == Modules!

Question

What is a value of type $\exists t.\tau$?

Answer: A module!

Existentialism == Modules!

Question

What is a value of type $\exists t.\tau$?

Answer: A module!

```
structure M : S =
  struct
    type t = int
    val x = 150
    val f = fn x => x + 1
  end
```

is a value of type $\exists t.\{x : t, f : t \to t\}$

Existentialism == Modules!

To unpack a structure, use the `open` keyword!

Existentialism == Modules!

To unpack a structure, use the `open` keyword!

`open M` gives me:

- a type `t`
- a value of type `t`
- a value of type `t -> t`

Existentialism == Modules!

To unpack a structure, use the `open` keyword!

`open M` gives me:

- a type `t`
- a value of type `t`
- a value of type `t -> t`

In other words, I have a type `t` and a value of type `t * (t -> t)`
(Remember the type of `M` was $\exists t.t \times (t \to t)$)

Existentialism == Modules!

To unpack a structure, use the `open` keyword!

`open M` gives me:

- a type `t`
- a value of type `t`
- a value of type `t -> t`

In other words, I have a type `t` and a value of type `t * (t -> t)`
(Remember the type of `M` was $\exists t.t \times (t \to t)$)

Main Idea

opening a value (module) of type $\exists t.\tau$ gives us a type t and a value of type τ

Typechecking Rules

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

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

Example: Stacks!

```
signature STACK =
  sig
    type t
    val empty : t
    val push : int -> t -> t
    val pop : t -> (int * t) option
  end

structure ListStack : STACK =
  struct
    type t = int list
    val empty = []
    fun push x xs = x :: xs
    fun pop [] = NONE
      | pop (x :: xs) = SOME (x, xs)
  end
```

Example: Stacks!

$$Stack =$$

Example: Stacks!

$$Stack =$$

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

$$ListStack : Stack =$$

Example: Stacks!

$$Stack =$$

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

$$ListStack : Stack =$$

$$struct\ \ type\ \ t = int\ \ list\ \ in$$

$$\{empty = Nil,$$

$$push = Cons,$$

$$pop = ...\}$$

What about functors?

```
signature STACK =
  sig
    type t
    val empty : t
    val push : int -> t -> t
    val pop : t -> (int * t) option
  end

functor MkDoubleStack (S : STACK) : STACK =
  struct
    type t = S.t
    val empty = S.empty
    fun push x s = S.push x (S.push x s)
    val pop = S.pop
  end
```

What about functors?

$$MkDoubleStack : Stack \to Stack =$$

What about functors?

$$MkDoubleStack : Stack \to Stack =$$

$$\lambda(S : Stack).$$

$$open\ S\ as\ t', s\ in$$

What about functors?

$$MkDoubleStack : Stack \rightarrow Stack =$$

$$\lambda(S : Stack).$$

$$open\ S\ as\ t', s\ in$$

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

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

$$push = \lambda(x : int).(s.push\ x)\ o\ (s.push\ x)$$

$$pop = s.pop\}$$

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

Question

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

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

Question

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

Answer: Yes! But How?

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

Question

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

Answer: 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 $\rightarrow$)

Question

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

Answer: Yes! But How?

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

If we have a function that requires a value of type A and a value of type B, we can give it arguments!

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

Question

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

Answer: Yes! But How?

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

If we have a function that requires a value of type A and a value of type B, we can give it arguments!

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

Product Types in System F

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

Product Types in System F

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

$$pair : \forall A\ B.A \to B \to A \times B =$$

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

Product Types in System F

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

$$pair : \forall A\ B.A \to B \to A \times B =$$

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

$$fst : \forall A\ B.A \times B \to A =$$

$$\Lambda(A\ B)\ \lambda(p : A \times B)\ p[A]\ (\lambda(x : A)\ \lambda(y : B)\ x)$$

Product Types in System F

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

$$pair : \forall A\ B.A \to B \to A \times B =$$

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

$$fst : \forall A\ B.A \times B \to A =$$

$$\Lambda(A\ B)\ \lambda(p : A \times B)\ p[A]\ (\lambda(x : A)\ \lambda(y : B)\ x)$$

$$snd : \forall A\ B.A \times B \to B =$$

$$\Lambda(A\ B)\ \lambda(p : A \times B)\ p[B]\ (\lambda(x : A)\ \lambda(y : B)\ y)$$

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.

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

Sum Types

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

Sum Types

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

$$\mathit{InjectLeft} : \forall A\ B.A \to A + B =$$

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

Sum Types

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

$$\mathit{InjectLeft} : \forall A\ B.A \to A + B =$$

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

$$\mathit{InjectRight} : \forall A\ B.B \to A + B =$$

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

Sum Types

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

$$InjectLeft : \forall A\ B.A \to A + B =$$

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

$$InjectRight : \forall A\ B.B \to A + B =$$

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

Question

What about case?

Sum Types

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

$$\mathit{InjectLeft} : \forall A\ B.A \to A + B =$$

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

$$\mathit{InjectRight} : \forall A\ B.B \to A + B =$$

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

Question

What about case?

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