Polymorphism and Type Inference

Hype for Types

September 30, 2020

Polymorphism

Identity

Recall lambda abstraction from the Simply Typed Lambda Calculus

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

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
```

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*.
Let's bind it *explicitly*!
$\Lambda(a)\lambda(x : a)x : \forall a.a \to a$
How do we use this?
$(\Lambda(a)\lambda(x : a)x)[Nat] \Longrightarrow \lambda(x : Nat)x$

System F

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

$$
\begin{array}{rcll}
e & ::= & x & \text{term variable} \\
 & | & \lambda(x : \tau)e & \text{term abstraction} \\
 & | & \Lambda(t)e & \text{type abstraction} \\
 & | & e_1 e_2 & \text{term application} \\
 & | & e_1[\tau] & \text{type application} \\
 & & & \\
\tau & ::= & t & \text{type variable} \\
 & | & \tau_1 \to \tau_2 & \text{function type} \\
 & | & \forall t.\tau & \text{polymorphic type}
\end{array}
$$

System F

And some inference rules!

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

$$\frac{x : \tau \in \Gamma}{\Delta | \Gamma \vdash x : \tau} \qquad \frac{\Delta | \Gamma, x : \tau \vdash e : \tau' \quad \Delta \vdash \tau \ type}{\Delta | \Gamma \vdash \lambda(x : \tau)e : \tau \to \tau'} \qquad \frac{\Delta, t | \Gamma \vdash e : \tau}{\Delta | \Gamma \vdash \Lambda(t)e : \forall t.\tau}$$

$$\frac{\Delta | \Gamma \vdash e_1 : \tau \to \tau' \quad \Delta | \Gamma \vdash e_2 : \tau}{\Delta | \Gamma \vdash e_1 e_2 : \tau'} \qquad \frac{\Delta | \Gamma \vdash e : \forall t.\tau \quad \Delta \vdash \tau' \ type}{\Delta | \Gamma \vdash e[\tau'] : \tau[\tau'/t]}$$

Question

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

Some Fing Functions

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

$$\Lambda(a)\Lambda(b)\Lambda(c)\lambda(f : a \to b \to c)\lambda(x : b)\lambda(y : a)f\ y\ x$$

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

$$\Lambda(a)\Lambda(b)\Lambda(c)\lambda(f : a \to b)\lambda(g : b \to c)\lambda(x : a)g(f\ x)$$

Does SML implement System F?

Is the polymorphism of SML equivalent to the polymorphism of System F?
Is `'a -> 'a` always really $\forall a.a \to a$?
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:

$$hmm = \Lambda(a)\lambda(id : a \to a)(id\ 1, id\ true)$$

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!

Type Inference

Is it possible?

Warning

We're leaving full System F! Type inference there is *undecidable*.

Instead, we'll look at type inference for a more restricted language, similar to SML.

For All Foralls

Restriction

All $\forall$'s must be at the *front* of a type. For example:

$$\forall\alpha.\ \forall\beta.\ \alpha \times \beta \to \beta$$

We will not consider types such as:

$$\forall\alpha.\ \alpha \times (\forall\beta.\ \beta \to \beta) \to \alpha$$

This is how SML works!

```
fun fst (a : 'a, b : 'b) : 'a = a
```

Fun fact:

```
fun ('a,'b) fst (a : 'a, b : 'b) : 'a = a
```

But not:

```
fun 'a nope (a : 'a, f : 'b -> 'b) : 'a = f a
```

Type Compatibility

Are the following pairs of types "compatible" in SML?

- $\alpha \times$ **nat** and **bool** $\times$ **nat**?
- $\alpha \times$ **bool** and **bool** $\times$ **nat**?
- $\alpha \times$ **nat** and **bool** $\times \beta$?
- α **list** and **nat list**?
- α **list** and **nat option**?
- $\alpha \times \beta$ and $\beta \times \gamma$?
- $\alpha \times \alpha$ **list** and **nat** $\times$ **bool list**?
- α and **nat** $\times \alpha$?

Key Idea

Two types are "compatible" iff there's a *substitution* which makes them equal.

Substitution

Definition

A *substitution* is a function σ : variable $\to$ type.

Consider $\alpha \times \textbf{nat}$ and $\textbf{bool} \times \beta$. Given the substitution

$$\sigma(t) = \begin{cases} \textbf{bool} & t = \alpha \\ \textbf{nat} & t = \beta \\ t & \text{otherwise} \end{cases}$$

we have that $\sigma(\alpha \times \textbf{nat}) = \textbf{bool} \times \textbf{nat} = \sigma(\textbf{bool} \times \beta)$.

Therefore, we say that $\alpha \times \textbf{nat}$ and $\textbf{bool} \times \beta$ are *unifiable*.

Unification Algorithm

We describe a judgement $C \Rightarrow C'$ which attempts to remove one constraint, or fail (represented as $\bot$).

We start with one constraint when checking if τ_1 and τ_2 are unifiable: simply $\langle \tau_1, \tau_2 \rangle$.

$$\frac{}{C, \langle c(\tau_1, \ldots, \tau_n), c(\nu_1, \ldots, \nu_n) \rangle \Rightarrow C, \langle \tau_1, \nu_1 \rangle, \ldots, \langle \tau_n, \nu_n \rangle} \text{ (Decompose)}$$

$$\frac{c \neq d}{C, \langle c(\vec{\tau}), d(\vec{\nu}) \rangle \Rightarrow \bot} \text{ (Conflict)} \qquad \frac{}{C, \langle t, t \rangle \Rightarrow C} \text{ (Delete)}$$

$$\frac{t \notin \tau \quad \sigma = t \to \tau \quad t \in C}{C, \langle t, \tau \rangle \Rightarrow \sigma(C), \langle t, \tau \rangle} \text{ (Eliminate)}$$

$$\frac{}{C, \langle c(\vec{\tau}), t \rangle \Rightarrow C, \langle t, c(\vec{\tau}) \rangle} \text{ (Swap)} \qquad \frac{t \in \tau \quad t \neq \tau}{C, \langle t, \tau \rangle \Rightarrow \bot} \text{ (Occurs Check)}$$

Tracing Unification

```
val map    : ('a -> 'b) -> 'a list -> 'b list
val foldr  : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
val op o   : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c)

val result = map o foldr
```

Tracing Unification

```
('y -> 'z)                                    * ('x -> 'y)
(('a -> 'b) -> 'a list -> 'b list) *
  (('c * 'd -> 'd) -> 'd -> 'c list -> 'd)
```

Tracing Unification

```
'y -> 'z
('a -> 'b) -> 'a list -> 'b list

'x -> 'y
('c * 'd -> 'd) -> 'd -> 'c list -> 'd
```

Tracing Unification

```
'y
'a -> 'b

'z
'a list -> 'b list

'x -> 'y
('c * 'd -> 'd) -> 'd -> 'c list -> 'd
```

Tracing Unification

```
'y
'a -> 'b

'z
'a list -> 'b list

'x -> ('a -> 'b)
('c * 'd -> 'd) -> 'd -> 'c list -> 'd
```

Tracing Unification

```
'y
'a -> 'b

'z
'a list -> 'b list

'x
'c * 'd -> 'd

'd -> 'c list -> 'd
'a -> 'b
```

Tracing Unification

```
'y
'a -> 'b

'z
'a list -> 'b list

'x
'c * 'd -> 'd

'd
'a

'c list -> 'd
'b
```

Tracing Unification

```
'y
'a -> 'b

'z
'a list -> 'b list

'x
'c * 'a -> 'a

'd
'a

'c list -> 'a
'b
```

Tracing Unification

```
'y
'a -> 'b

'z
'a list -> 'b list

'x
'c * 'a -> 'a

'd
'a

'b
'c list -> 'a
```

Tracing Unification

```
'y
'a -> ('c list -> 'a)

'z
'a list -> ('c list -> 'a) list

'x
'c * 'a -> 'a

'd
'a

'b
'c list -> 'a
```

Tracing Unification

- `'b` is `'c list -> 'a`
- `'d` is `'a`
- `'x` is `'c * 'a -> 'a`
- `'y` is `'a -> 'c list -> 'a`
- `'z` is `'a list -> ('c list -> 'a) list`

Recall:

```
map o foldr : 'x -> 'z
```

Hence:

```
map o foldr :
  ('c * 'a -> 'a)
  -> ('a list -> ('c list -> 'a) list)
```

Success!

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?
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

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

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

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

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

$$\{empty = Nil,$$

$$push = Cons,$$

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

Practical Uses

$$mkEvenStack : Stack \to EvenStack =$$

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

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

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

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

$$pop = s.pop\}$$

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 *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!

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

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 $\rightarrow$)

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.

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

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

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

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

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

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

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

Sum Types?

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