

Existential Types

Avik Chaudhuri

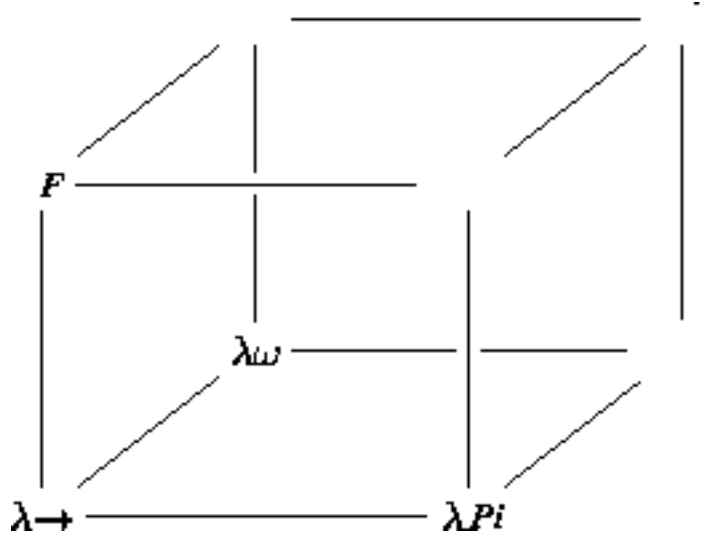
May 12, 2005

1 System F Theory

1.1 Lambda Cube

The different fundamental changes we've made to the simple languages can be described as progression along different axes. Movement on each axis allows more kinds of parameters than the simple lambda calculus which is the origin.

Language	New Parameters	
λ_{\rightarrow}	$\lambda x.t$	$Terms \rightarrow Terms$
<i>System F</i>	$\Lambda X.t$	$Types \rightarrow Terms$
λ_{ω}	$\Pi X, T$	$Types \rightarrow Types$
λ_{Π}	$\Pi x.T$	$Terms \rightarrow Types$



The remaining four languages were left unnamed.

1.2 Type Reconstruction Theory

System F does not possess a Type Reconstruction Theorem (9.5.2) as in the simply-typed lambda calculus. The theorem is as follows:

Theorem 1. Type Reconstruction Theorem for System F

Given a closed term m in untyped lambda calculus it is undecidable whether there is some well-typed term t in System F such that $\text{erase}(t) = m$.

where the $\text{erase}(t)$ is a map from terms in System F to untyped lambda calculus which strips terms of their type.

Note: The theory for System F covers that of System F with existentials.

2 System F with Existential Types

2.1 Syntax

$$t ::= \dot{\quad}$$

$\{^*T, t\}$ as T	(packing)
$\text{let}\{X, x\} = t$ in t	(unpacking)

$$T ::= \dot{\quad}$$

$\exists X.T$

$$v ::= \dot{\quad}$$

$\{^*T, v\}$ as T

2.2 Examples

The main example here is the Counter as in the book. Both the Abstract Data Type (ADT) and Object oriented version of Counter were implemented in class.

2.3 ADT

The philosophy in this implementation is that there should be a layer of interface over the concrete data types. In the Counter the underlying data type is a nat.

Instead of incrementing the Nat directly an accessor function `inc` will change the value of the counter.

$$\text{CounterADT} = \{\text{Hidden}\} \text{ as } \exists \text{Counter}.\{get : \text{Counter} \rightarrow \text{Nat},$$

$$\quad \quad \quad inc : \text{Counter} \rightarrow \text{Counter},$$

$$\quad \quad \quad new : \text{Counter}\}$$

Where, $\{\text{Hidden}\} = \{^*\text{Nat}, \{new = 0,$
 $\quad \quad \quad inc : \lambda x : \text{Nat}.(x + 1),$
 $\quad \quad \quad get : \lambda x : \text{Nat}.x\}\}$

$\{\text{Hidden}\}$ could also be implemented with a bool as the underlying data type. Note: ADTs are packed and the unpacked immediately.

$\text{let}\{\text{Counter}, c\} = \text{CounterADT}$ in t
 t can use `c.get`, `c.inc`, `c.new`.
 $c.get(c.new) + 1 \rightarrow 1$
 $c.new + 1 \rightarrow \text{error}$

2.4 Objects

The Counter can be implemented as an object as well.

$$c = \{ *Nat, \\ \{state = 5, \\ methods = \{get = \lambda x : Nat.x, \\ inc = \lambda x : Nat.succ(x)\}\} \} \text{ as CounterO}$$

Where $\text{CounterO} = \{\exists X, \{state : X, methods : \{get : X \rightarrow Nat, inc : X \rightarrow X\}\}\}$

Objects "carry" their own methods along with them. For example the *sendget* and *incget* functions on p.373 of Pierce¹.

2.5 ADT v. Objects

The advantage of Objects is that because the accessor methods are carried along with it, a single program can handle different implementations of the type CounterO . This is impossible and will lead to a typing error under an ADT implementation. For example after unpacking two differing implementations of CounterADT :

$$\text{let } \{Counter, C\} = \text{CounterADT}_1 \text{ in } t_1 \text{ let } \{Counter, C\} = \text{CounterADT}_2 \text{ in } t_2$$

the types of the counters in t_1 and t_2 are:

$$\{new : Counter_1, get : Counter_1 \rightarrow Nat, inc : Counter_1 \rightarrow Counter_1\}$$
$$\{new : Counter_2, get : Counter_2 \rightarrow Nat, inc : Counter_2 \rightarrow Counter_2\}$$

respectively. Clearly programs can't treat different implementations of CounterADT uniformly without a type error.

The advantage of ADT style is that they are susceptible to strong binary operations. More specifically because of their reliance on the underlying data types, strong binary operations such as equal are more easily written to manipulate two counters and determine equality. Other operations such as the union of two set types are impossible to write with a Counter written in object style while it is possible with one written in the ADT style.

3 Connection to Howard-Curry Isomorphism

What would an existential type $\exists X.T$ be in the Howard-Curry Isomorphism? In a way the packing construct $\{*T, t\}$ as T is a proof that the type T is inhabited with $*T$ as the witness and t as the proof of its type.

The \exists construct has the type-checker lose information about the type of a term and thus forces the programmer to ignore the underlying data type and manipulate the object based on the meta-type X in the unpacking construct $\text{let}\{X, x\} = t \text{ in } t$. For that matter why is the value of the underlying data type, X given to the programmer? Avik believes that because the scope of the type can be used later outside of the let construct to be used later in the code.

4 Encoding existentials in universals

In normal logic $\exists X.T = \neg(\forall X.\neg T)$ holds. So if we could translate the individual symbol of negation to our type system we would be set. One way of looking out this is that $\neg p = p \Rightarrow \text{false}$. As false

¹Benjamin C. Pierce *Types and Programming Languages* MIT Press, Cambridge, MA, USA, 2002

in our language is represented by \perp this translates to $\neg p = p \Rightarrow \perp$. Another way of looking at false is that it is the same as $\forall Y.Y$. From these adaptations we come up with a translation for first equation:

$$\begin{aligned} \exists X.T &= \neg(\forall X.\neg T) \\ &= (\forall X.\neg T) \Rightarrow \forall Y.Y \\ &= (\forall X.T \Rightarrow \forall Y.Y) \Rightarrow \forall Y.Y \\ &= \forall Y.(\forall X.T \Rightarrow Y) \Rightarrow \forall Y.Y \\ &= \forall Y.((\forall X.T \Rightarrow Y) \Rightarrow Y) \end{aligned}$$

Lesson: Logic can be used to build languages. What are the benefits?

$$\begin{aligned} \{ *S, t \} \text{ as } \exists X.T &= \Lambda Y.\lambda f : (\forall X.T \rightarrow Y).(\dots) \\ &\quad (\text{as an } \exists \text{ type begins with a type parameterization and a function of type } (\forall X.T \rightarrow Y).) \\ &= \Lambda Y.\lambda f : (\forall X.T \rightarrow Y).f[S] \\ &\quad (\text{as we want } Y \text{ to be our final value we shall apply } f \text{ to something. We need a type for } f.) \\ &= \Lambda Y.\lambda f : (\forall X.T \rightarrow Y).f[S]t \\ &\quad (\text{we need to apply something of type } t \dots) \end{aligned}$$

Using similar reasoning

$$\text{let } \{ X, x \} = t_1 \text{ in } t_2 = t_1[T_2](\Lambda X.\lambda x : T_{11}.t_2)$$

5 Examples

Recall how List X was implemented in the book.

List X = $\forall R.(X \rightarrow R \rightarrow R) \rightarrow X \rightarrow R$

nil[X]: List X

cons[X]: $X \rightarrow ListX \rightarrow ListX$

nil = $\Lambda X. \Lambda R. \lambda c : (X \rightarrow R \rightarrow R). \lambda n : R. n$

cons = $\Lambda X. \lambda h : X. \lambda t : ListX. \Lambda R. \lambda c : (X \rightarrow R \rightarrow R). \lambda n : R. c[h] (t[R] c n)$

For example the list [1; 2 ; 3] will look like so: $f(1, f(2, f(3, init)))$.