Using Our Interpreter

- Here's the abstract syntax of our implemented interpreter:

Names	x	
Numbers	i	
Terms	E	$::= \; x \mid i \mid \lambda x.E \mid E\,E'$

- We extend the language by a simple `let` construct to bind names to expressions:

$$let\ x\ =\ E\ in\ E' \quad \overset{\text{def}}{\equiv} \quad (\lambda x.E')E$$

Encoding Algebraic Datatypes

- We can encode any algebraic datatype in our simple language using *Church Encoding*
- Let's model lists: data List a = Nil | Cons a (List a)
- A list `xs` can be accessed using pattern matching:

```
case xs of
    Nil            ⇒ E1
|   Cons (x, xs)   ⇒ E2
```

- The constructors `Nil` and `Cons` are encoded as functions that take the two expressions E_1 and E_2 of a case–expression and return the corresponding one:

```
let nil  = λa.λb.a in
let cons = λx.λxs.λa.λb.b x xs in ...
```

- Here's a definition of the `isEmpty` function:

```
let isEmpty = λxs.xs true (λy.λys.false) in ...
```


Church Encoding

For datatype D with constructors $K_1, \ldots, K_m$ of arity $n_1, \ldots, n_m$, a constructor

$$K_i\ x_1 \ldots x_{n_i}$$

is encoded as

$$K_i \;=\; \lambda e_1 \ldots \lambda e_m.\lambda x_1 \ldots \lambda x_{n_i}.e_i\ x_1 \ldots x_{n_i}$$

A case–expression for accessing this datatype

$$\begin{array}{ll} \textbf{case}\ E\ \texttt{of} & \\ & K_1\ x_1\ \ldots\ x_{n_1} = E_1 \\ \mid & \ldots \\ \mid & K_m\ x_1\ \ldots\ x_{n_m} = E_m \end{array}$$

is encoded as

$$E\ (\lambda x_1 \ldots \lambda x_{n_1}.E_1)\ \ldots\ (\lambda x_1 \ldots \lambda x_{n_m}.E_m)$$

Recursive Functions

- Our `let` construct doesn't allow recursion; the following definition of a `length` function for lists would be illegal:

  `let length =` λ`xs.xs 0 (`λ`y.`λ`ys.1 + (length ys)) in` E
- According to our definition of `let`, this would be:

  `(`λ`length.`E`) (`λ`xs.xs 0 (`λ`y.`λ`ys.1 + (length ys)))`
- Since `length` is not bound in the argument of this application, we have to find another way of expressing recursion

Fixed Points

If we have a recursive definition

`let` f `=` E `in ...`

where E refers to f, we can interpret this as a solution F to the equation

$$\begin{array}{rrcl} & f & = & E \\ \Longleftrightarrow & F & = & [F/f]E \\ \Longleftrightarrow & F & = & (\lambda f.E)\ F \\ \Longleftrightarrow & F & \text{is} & \text{a fixed point of } \lambda f.E \end{array}$$

Proposition: The solutions of $f = E$ are exactly the fixed points of $\lambda f.E$

Fixed Point Operator Y

There exists a fixed point operator Y

$$Y \stackrel{\text{def}}{\equiv} \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$$

that computes a fixed point of a given function:

$$Y f \quad = \quad f\ (Y f)$$

Let's write function `length` using this operator:

`let length =` Y `(`λ`l.`λ`xs.xs 0 (`λ`y.`λ`ys.1 + (l ys))) in ...`

We apply `length` to the one element list `[1]` to see how the Y operator works:

`length (cons 1 nil)`

$$\begin{array}{ll}
 & Y\,\underbrace{(\lambda l.\lambda xs.xs\ 0\ (\lambda y.\lambda ys.1\ +\ (l\ ys)))}_{f}\ (cons\ 1\ nil) \\
\rightarrow & \underbrace{(\lambda x.f(x\ x))\ (\lambda x.f(x\ x))}_{g}\ (cons\ 1\ nil) \\
\rightarrow & f\ g\ (cons\ 1\ nil) \\
\rightarrow & (\lambda xs.xs\ 0\ (\lambda y.\lambda ys.1\ +\ (g\ ys)))\ (\lambda a.\lambda b.b\ 1\ nil) \\
\rightarrow & (\lambda a.\lambda b.b\ 1\ nil)\ 0\ (\lambda y.\lambda ys.1\ +\ (g\ ys)) \\
\rightarrow & (\lambda b.b\ 1\ nil)\ (\lambda y.\lambda ys.1\ +\ (g\ ys)) \\
\rightarrow & (\lambda y.\lambda ys.1\ +\ (g\ ys))\ 1\ nil \\
\rightarrow & (\lambda ys.1\ +\ (g\ ys))\ nil \\
\rightarrow & 1\ +\ (g\ nil) \\
\rightarrow & 1\ +\ (f\ g\ (\lambda a.\lambda b.a)) \\
\rightarrow & \ldots \\
\rightarrow & 1 + 0
\end{array}$$

The Applicative-Order Fixed Point Operator

- The Y combinator only works for call-by-name
- But we can modify it for using it with a call-by-value lambda calculus:
  - An expression E that evaluates to a function can be re-written in the following way: $\lambda z.E\ z$
  - Within the definition of Y, f takes a function and returns one, so we can apply our rule:

$$f\ (x\ x) \quad \rightarrow \quad f(\lambda z.x\ x\ z)$$

- Our new Y operator is called applicative-order fixed point operator:

$$Y \quad \stackrel{\text{def}}{\equiv} \quad \lambda f.(\lambda x.f\ (\lambda z.x\ x\ z))\ (\lambda x.f\ (\lambda z.x\ x\ z))$$

- Let's see how it works, compared to the previous version of Y

$$
\begin{array}{ll}
 & Y\,\underbrace{(\lambda l.\lambda xs.xs\ 0\ (\lambda y.\lambda ys.1\ +\ (l\ ys)))}_{f}\ (cons\ 1\ nil) \\
\rightarrow & (\lambda x.f(\lambda z.x\ x\ z))\ \underbrace{(\lambda x.f(\lambda z.x\ x\ z))}_{h}\ (cons\ 1\ nil) \\
\rightarrow & f\ \underbrace{(\lambda z.h\ h\ z)}_{g}\ (cons\ 1\ nil) \\
\rightarrow & (\lambda xs.xs\ 0\ (\lambda y.\lambda ys.1\ +\ (g\ ys)))\ (\lambda a.\lambda b.b\ 1\ nil) \\
\rightarrow & (\lambda a.\lambda b.b\ 1\ nil)\ 0\ (\lambda y.\lambda ys.1\ +\ (g\ ys)) \\
\rightarrow & (\lambda b.b\ 1\ nil)\ (\lambda y.\lambda ys.1\ +\ (g\ ys)) \\
\rightarrow & (\lambda y.\lambda ys.1\ +\ (g\ ys))\ 1\ nil \\
\rightarrow & (\lambda ys.1\ +\ (g\ ys))\ nil \\
\rightarrow & 1\ +\ (g\ nil) \\
\rightarrow & 1\ +\ (h\ h\ (\lambda a.\lambda b.a)) \\
\rightarrow & \ldots
\end{array}
$$

Bibliography

If you want to understand Y completely, then you should read

*A Lecture on the **Why** of **Y***, by Matthias Felleisen

In this paper Y is derived step by step. You can get a copy from our homepage at:

`http://lampwww.epfl.ch/courses/fondements00/`