CSc 520

Principles of Programming Languages

23: Lambda Calculus — Pure

Christian Collberg

collberg@cs.arizona.edu

Department of Computer Science

University of Arizona

Pure vs. Impure Lambda Calculus

- The version of lambda calculus we have looked at so far has been impure —it has contained constants such as $\langle 1, 2, 3, \cdots \rangle$ and add, sqr, etc.
- Church's Thesis says that lambda calculus can define every computable function.
- If we're going to believe Church's Thesis we are going to have to define the natural numbers, arithmetic operations, booleans, pairs, conditional expressions and recursion, directly in the calculus.
- A lambda calculus so defined contains no constants, and is said to be pure.

Church's Numerals

- We can encode a natural number as the number of times a function parameter is applied:

$$0 \equiv (\lambda f.(\lambda x.x))$$

$$1 \equiv (\lambda f.(\lambda x.(f \;\; x)))$$

$$2 \equiv (\lambda f.(\lambda x.(f \;\; (f \;\; x))))$$

$$3 \equiv (\lambda f.(\lambda x.(f \;\; (f \;\; (f \;\; x)))))$$

- We can now define arithmetic operations:

$$\text{succ} \equiv (\lambda n.(\lambda f.(\lambda x.(f \;\; ((n \;\; f) \;\; x)))))$$

$$\text{add} \equiv (\lambda m.(\lambda n.(\lambda f.(\lambda x.((m \;\; f) \;\; ((n \;\; f) \;\; x))))))$$

Church's Numerals — succ

- succ's first argument is n, the number to be incremented. succ just adds one more application of the f function (its second argument). The third argument (x) is the "base case", that is, zero.

$$2 \equiv (\lambda g.(\lambda y.(g \;\; (g \;\; y))))$$

$$\text{succ} \equiv (\lambda n.(\lambda f.(\lambda x.(f \;\; ((n \;\; f) \;\; x)))))$$

$$(\text{succ} \;\; 2) \Rightarrow$$

$$((\lambda n.(\lambda f.(\lambda x.(f \;\; ((n \;\; f) \;\; x))))) \;\; (\lambda g.(\lambda y.(g \;\; (g \;\; y)))))$$

$$\vdots$$

Church's Numerals — succ...

succ $\equiv (\lambda n.(\lambda f.(\lambda x.(f \;\; ((n \;\; f) \;\; x)))))$
$2 \equiv (\lambda g.(\lambda y.(g \;\; (g \;\; y))))$
$3 \equiv (\lambda f.(\lambda x.(f \;\; (f \;\; (f \;\; x)))))$

(succ $2) \Rightarrow$

$((\lambda n.(\lambda f.(\lambda x.(f \;\; ((n \;\; f) \;\; x))))) \;\; (\lambda g.(\lambda y.(g \;\; (g \;\; y))))) \Rightarrow_\beta$

$(\lambda f.(\lambda x.(f \;\; (((\lambda g.(\lambda y.(g \;\; (g \;\; y)))) \;\; f) \;\; x)))) \Rightarrow_\beta$

$(\lambda f.(\lambda x.(f \;\; ((\lambda y.(f \;\; (f \;\; y))) \;\; x)))) \Rightarrow_\beta$

$(\lambda f.(\lambda x.(f \;\; (f \;\; (f \;\; x))))) \equiv 3$

Church's Numerals — add

- add takes two numbers n and m as arguments.
- $(m \; f)$ simply plugs in f as the function used to represent numbers in the expression for m, $(n \; f)$ does the same for the second number.
- (add $f(f(x))\; g(g(g(x))))$ (representing $2+3$) constructs a new function $h(h(h(h(h(x)))))$ (representing 5).

$2 \equiv (\lambda g.(\lambda y.(g \;\; (g \;\; y))))$
$3 \equiv (\lambda h.(\lambda z.(h \;\; (h \;\; (h \;\; z)))))$
add $\equiv (\lambda m.(\lambda n.(\lambda f.(\lambda x.((m \;\; f) \;\; ((n \;\; f) \;\; x))))))$

(add $2 \;\; 3) \Rightarrow$
$(((\lambda m.(\lambda n.(\lambda f.(\lambda x.((m \;\; f) \;\; ((n \;\; f) \;\; x)))))) \;\; (\lambda g.(\lambda y.(g \;\; (g \;\; y))))) \;\; 3$

Church's Numerals — add...

$2 \equiv (\lambda g.(\lambda y.(g \;\; (g \;\; y))))$
$3 \equiv (\lambda h.(\lambda z.(h \;\; (h \;\; (h \;\; z)))))$
add $\equiv (\lambda m.(\lambda n.(\lambda f.(\lambda x.((m \;\; f) \;\; ((n \;\; f) \;\; x))))))$

(add $2 \;\; 3) \Rightarrow$
$(((\lambda m.(\lambda n.(\lambda f.(\lambda x.((m \;\; f) \;\; ((n \;\; f) \;\; x)))))) \;\; (\lambda g.(\lambda y.(g \;\; (g \;\; y))))) \;\; 3) \Rightarrow_\beta$
$((\lambda n.(\lambda f.(\lambda x.((\lambda y.(f \;\; (f \;\; y))) \;\; ((n \;\; f) \;\; x))))) \;\; 3) \Rightarrow_\beta$
$((\lambda n.(\lambda f.(\lambda x.((\lambda y.(f \;\; (f \;\; y))) \;\; ((n \;\; f) \;\; x))))) \;\; (\lambda h.(\lambda z.(h \;\; (h \;\; (h \;\; z)))))) \Rightarrow_\beta$
$(\lambda f.(\lambda x.((\lambda y.(f \;\; (f \;\; y))) \;\; (((\lambda h.(\lambda z.(h \;\; (h \;\; (h \;\; z))))) \;\; f) \;\; x)))) \Rightarrow_\beta$
$(\lambda f.(\lambda x.((\lambda y.(f \;\; (f \;\; y))) \;\; ((\lambda z.(f \;\; (f \;\; (f \;\; z)))) \;\; x)))) \Rightarrow_\beta$
$(\lambda f.(\lambda x.((\lambda y.(f \;\; (f \;\; y))) \;\; (f \;\; (f \;\; (f \;\; x)))))) \Rightarrow_\beta$
$(\lambda f.(\lambda x.(f \;\; (f \;\; (f \;\; (f \;\; (f \;\; x))))))) = 5$

Church's Numerals — mult

- Multiplying $m * n$ is like adding m copies of n together:

add $\equiv (\lambda m.(\lambda n.(\lambda f.(\lambda x.((m \;\; f) \;\; ((n \;\; f) \;\; x))))))$

mult $\equiv (\lambda m.(\lambda n.(m \;\; (($plus $n) \;\;$ zero$))))$

Pairs

- Just like in Scheme, we can define pairs —these allow us to construct data structures such as lists and trees.
- The definition of Pair below is similar to a dotted pair notation (or `cons`) in Scheme.
- Head and Tail correspond to `car` and `cdr`, Nil is a special constant.

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))$$

$$\text{Head} \equiv (\lambda g.(g\ \ (\lambda a.(\lambda b.a))))$$

$$\text{Tail} \equiv (\lambda g.(g\ \ (\lambda a.(\lambda b.b))))$$

$$\text{Nil} \equiv (\lambda x.(\lambda a.(\lambda b.a)))$$

Pairs...

- We can construct a pair (p, q) (or $(p.q)$ in Scheme notation) like this:

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))$$

$$((\text{Pair}\ \ p)\ \ q) =$$

$$(((\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))\ \ p)\ \ q) \Rightarrow_\beta$$

$$((\lambda b.(\lambda f.((f\ \ p)\ \ b)))\ \ q) \Rightarrow_\beta$$

$$(\lambda f.((f\ \ p)\ \ q))$$

Pairs...

- We can construct the list $[2]$ like this:

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))$$
$$\text{Nil} \equiv (\lambda x.(\lambda a.(\lambda b.a)))$$

$$((\text{Pair}\ \ 2)\ \ \text{Nil}) =$$

$$(((\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))\ \ 2)\ \ \text{Nil}) \Rightarrow_\beta$$

$$((\lambda b.(\lambda f.((f\ \ 2)\ \ b)))\ \ \text{Nil}) \Rightarrow_\beta$$

$$(\lambda f.((f\ \ 2)\ \ \text{Nil})) = (\lambda f.((f\ \ 2)\ \ (\lambda x.(\lambda a.(\lambda b.a)))))$$

- We can go even further and substitute in the definition of 2.

Pairs...

- We can construct the list $[1, 2]$ like this:

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))$$
$$\text{Nil} \equiv (\lambda x.(\lambda a.(\lambda b.a)))$$

$$((\text{Pair}\ \ 1)\ \ ((\text{Pair}\ \ 2)\ \ \text{Nil})) =$$

$$((\text{Pair}\ \ 1)\ \ (\lambda f.((f\ \ 2)\ \ \text{Nil}))) =$$

$$(((\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))\ \ 1)\ \ (\lambda g.((g\ \ 2)\ \ \text{Nil}))) \Rightarrow_\beta$$

$$((\lambda b.(\lambda f.((f\ \ 1)\ \ b)))\ \ (\lambda g.((g\ \ 2)\ \ \text{Nil}))) \Rightarrow_\beta$$

$$(\lambda f.((f\ \ 1)\ \ (\lambda g.((g\ \ 2)\ \ \text{Nil}))))$$

Pairs...

- We can verify that Head works as specified:

$$\text{Pair} \equiv (\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))$$
$$\text{Head} \equiv (\lambda g.(g\ \ (\lambda a.(\lambda b.a))))$$

$$((\text{Pair}\ \ p)\ \ q) = (((\lambda a.(\lambda b.(\lambda f.((f\ \ a)\ \ b))))\ \ p)\ \ q) \Rightarrow_\beta$$

$$((\lambda b.(\lambda f.((f\ \ p)\ \ b)))\ \ q) \Rightarrow_\beta (\lambda f.((f\ \ p)\ \ q))$$

$$(\text{Head}\ \ ((\text{Pair}\ \ p)\ \ q)) = (\text{Head}\ \ (\lambda f.((f\ \ p)\ \ q))) =$$

$$((\lambda g.(g\ \ (\lambda a.(\lambda b.a))))\ \ (\lambda f.((f\ \ p)\ \ q))) \Rightarrow_\beta$$

$$((\lambda f.((f\ \ p)\ \ q))\ \ (\lambda a.(\lambda b.a))) \Rightarrow_\beta (((\lambda a.(\lambda b.a))\ \ p)\ \ q) \Rightarrow_\beta^* p$$

Church's Booleans

- We define two constants for true and false, and a function if for selection:

$$\text{true} \equiv (\lambda t.(\lambda f.t))$$

$$\text{false} \equiv (\lambda t.(\lambda f.f))$$

$$\text{if} \equiv (\lambda l.(\lambda m.(\lambda n.((l\ \ m)\ \ n))))$$

- We can now write programs with control flow!

Church's Booleans...

- We can verify that if works as expected:

$$\text{true} \equiv (\lambda t.(\lambda f.t))$$
$$\text{if} \equiv (\lambda l.(\lambda m.(\lambda n.((l\ \ m)\ \ n))))$$

$$(((\text{if}\ \ \text{true})\ \ v)\ \ w) = ((((\lambda l.(\lambda m.(\lambda n.((l\ \ m)\ \ n))))\ \ \text{true})\ \ v)\ \ w) \Rightarrow_\beta$$

$$(((\lambda m.(\lambda n.((\text{true}\ \ m)\ \ n)))\ \ v)\ \ w) =$$

$$(((\lambda m.(\lambda n.(((\lambda t.(\lambda f.t))\ \ m)\ \ n)))\ \ v)\ \ w) \Rightarrow_\beta$$

$$(((\lambda m.(\lambda n.((\lambda f.m)\ \ n)))\ \ v)\ \ w) \Rightarrow_\beta$$

$$(((\lambda m.(\lambda n.m))\ \ v)\ \ w) \Rightarrow_\beta ((\lambda n.v)\ \ w) \Rightarrow_\beta v$$

Church's Booleans...

- and can be defined like this:

$$\text{false} \equiv (\lambda t.(\lambda f.f))$$

$$\text{if} \equiv (\lambda l.(\lambda m.(\lambda n.((l\ \ m)\ \ n))))$$

$$\text{and} \equiv (\lambda a.(\lambda b.(((\text{if}\ \ a)\ \ b)\ \ \text{false}))) =$$

$$(\lambda a.(\lambda b.((((\lambda l.(\lambda m.(\lambda n.((l\ \ m)\ \ n))))\ \ a)\ \ b)\ \ \text{false}))) \Rightarrow_\beta^*$$

$$(\lambda a.(\lambda b.((a\ \ b)\ \ \text{false})))$$

Church's Booleans...

- iszero can be defined like this:

$$\text{false} \equiv (\lambda t.(\lambda f.f))$$

$$\text{if} \equiv (\lambda l.(\lambda m.(\lambda n.((l \;\; m) \;\; n))))$$

$$\text{iszero} \equiv (\lambda m.((m \;\; (\lambda x.\text{false})) \;\; \text{true}))$$

Recursion

Recursive Functions

- If lambda calculus is going to allow us to compute any function, we need for it to handle recursion.
- Example:

$$\text{fact} \equiv (\lambda n.\text{if } (\text{zero } n) \; 1 \; (\text{mult } n \; (\text{fact } (\text{pred } n))))$$

- Unfortunately, the name fact appears in the expression itself. Remember that we defined $\equiv$ -operator as *macro-expansion*, and recursive macros make no sense.
- Recursion is defined in normal programming languages, but not in lambda calculus.

Fixed Points

- A fixed point is a value x in the domain of a function that is the same in the range $f(x)$.
- In other words, a fixed point of a function is a value left fixed by that function; for example, 0 and 1 are fixed points of the squaring function.
- Formally, a value x is a fixed point of a function f if

$$f(x) = x$$

Fixed Points — Examples

- Every value in the domain of the identity function is a fixed point: $((\lambda x.x)$)
- $\texttt{factorial}(1) = 1$
- $\texttt{fibonacci}(0) = 0$
- $\texttt{fibonacci}(1) = 1$
- $\texttt{square}(0) = 0$
- $\texttt{square}(1) = 1$
- $\frac{de^x}{dx} = e^x$

Fixed Points — Examples...

f	fixed point
$(\lambda x.6)$	6
$(\lambda x.6 - x)$	3
$(\lambda x.x^2 + x - 4)$	2,-2
$(\lambda x.x)$	every value
$(\lambda x.x + 1)$	no value

- I.e., a fixed point is where you get back whatever you put in!

Fixed Point Combinators

- A combinator is a lambda-expression with no free variables.
- A fixed point combinator is a function Y which, given another function f, computes a fixed point of f, so that

$$f(\mathsf{Y}(f)) = \mathsf{Y}(f)$$

  for all functions f.
- Let's look at the fact function again:

$$\text{fact} \equiv (\lambda n.) if\ (\text{zero } n)\ 1\ (\text{mult } n\ (\text{fact } (\text{pred } n)))$$

Fixed Point Combinators...

- Let's turn

$$\text{fact} \equiv (\lambda n.\text{if } (\text{zero } n)\ 1\ (\text{mult } n\ (\text{fact } (\text{pred } n))))$$

  into a higher-order function, by replacing the call to fact with a function f

$$\text{ffact} \equiv (\lambda f.(\lambda n.\text{if } (\text{zero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n)))))$$

- Now, pass fact to ffact as a parameter, and do a β-reduction:

$$(\text{ffact fact}) \Rightarrow_\beta (\lambda n.\text{if } (\text{zero } n)\ 1\ (\text{mult } n\ (\text{fact } (\text{pred } n))))$$

Fixed Point Combinators. . .

- But, the right-hand side of

$$(\text{ffact fact}) \Rightarrow_\beta (\lambda n.\text{if } (\text{zero } n)\ 1\ (\text{mult } n\ (\text{fact } (\text{pred } n))))$$

is just the body of fact

$$\text{fact} \equiv (\lambda n.\text{if } (\text{zero } n)\ 1\ (\text{mult } n\ (\text{fact } (\text{pred } n))))$$

so we can write the identity:

$$(\text{ffact fact}) = \text{fact}$$

- Thus, fact is a fixed point for ffact.

Fixed Point Combinators. . .

- In lambda calculus, the fixed point combinator Y is defined as

$$\mathsf{Y} \equiv (\lambda h.((\lambda x.(h\ (x\ x)))\ (\lambda x.(h\ (x\ x)))))$$

- Let's see what happens when we apply that to an expression E:

$$\begin{aligned}(\mathsf{Y}\ \ E) =&\\ &((\lambda h.((\lambda x.(h\ \ (x\ \ x)))\ \ (\lambda x.(h\ \ (x\ \ x)))))\ \ E) \Rightarrow_\beta\\ &((\lambda x.(E\ \ (x\ \ x)))\ \ (\lambda x.(E\ \ (x\ \ x)))) \Rightarrow_\beta\\ &(E\ \ ((\lambda x.(E\ \ (x\ \ x)))\ \ (\lambda x.(E\ \ (x\ \ x))))) =\\ &(E\ \ (\mathsf{Y}\ \ E))\end{aligned}$$

Fixed Point Combinators. . .

- So, we saw that

$$(\mathsf{Y}\ E) \Rightarrow^*_\beta (E\ (\mathsf{Y}\ E))$$

- In other words,

$$E(\mathsf{Y}E) = \mathsf{Y}E$$

or for any expression E, $\mathsf{Y}E$ is a fixed point for E.

Fixed Point Combinators — Example

- Let's get back to our definition of fact:

$$\text{fact} \equiv (\lambda n.\text{if } (\text{zero } n)\ 1\ (\text{mult } n\ (\text{fact } (\text{pred } n))))$$

and the beta abstracted version ffact (we'll call it F for brevity)

$$\mathsf{F} \equiv (\lambda f.(\lambda n.\text{if } (\text{zero } n)\ 1\ (\text{mult } n\ (f\ (\text{pred } n)))))$$

- And, so we can define

$$\text{fact} \equiv (\mathsf{Y}\ \mathsf{F})$$

- Let's try to evaluate

$$(\text{fact } 3)$$

ixed Point Combinators — Example...

$\mathsf{F} \equiv (\lambda f.(\lambda n.\mathsf{if}\ \ (\mathsf{zero}\ \ n)\ \ 1\ \ (\mathsf{mult}\ \ n\ \ (f\ \ (\mathsf{pred}\ \ n)))))$
$\mathsf{fact} \equiv (\mathsf{Y}\ \ \mathsf{F})$
$\mathsf{Y} \equiv (\lambda h.((\lambda x.(h\ \ (x\ \ x)))\ \ (\lambda x.(h\ \ (x\ \ x)))))$

$(\mathsf{fact}\ \ 3) = ((\mathsf{Y}\ \ \mathsf{F})\ \ 3) =$

$(((\lambda h.((\lambda x.(h\ \ (x\ \ x)))\ \ (\lambda x.(h\ \ (x\ \ x)))))\ \ \mathsf{F})\ \ 3) \Rightarrow_\beta$

$(((\lambda x.(F\ \ (x\ \ x)))\ \ (\lambda x.(F\ \ (x\ \ x))))\ \ 3) =$

$((K\ \ K)\ \ 3) = \ldots$

- Where we've used the abbreviation

$$\mathsf{K} \equiv (\lambda x.(F\ (x\ x)))$$

Fixed Point Combinators — Example...

$\mathsf{F} \equiv (\lambda f.(\lambda n.\mathsf{if}\ \ (\mathsf{zero}\ \ n)\ \ 1\ \ (\mathsf{mult}\ \ n\ \ (f\ \ (\mathsf{pred}\ \ n)))))$
$\mathsf{K} \equiv (\lambda x.(F\ \ (x\ \ x)))$

$((K\ \ K)\ \ 3) =$

$(((\lambda x.(F\ \ (x\ \ x)))\ \ K)\ \ 3) \Rightarrow_\beta$

$((F\ \ (K\ \ K))\ \ 3) =$

$(((\lambda f.(\lambda n.\mathsf{if}\ \ (\mathsf{zero}\ \ n)\ \ 1\ \ (\mathsf{mult}\ \ n\ \ (f\ \ (\mathsf{pred}\ \ n)))))\ \ (K\ \ K))\ \ 3) \Rightarrow_\beta$

$((\lambda n.\mathsf{if}\ \ (\mathsf{zero}\ \ n)\ \ 1\ \ (\mathsf{mult}\ \ n\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ n))))\ \ 3) \Rightarrow_\beta$

$\mathsf{if}\ \ (\mathsf{zero}\ \ 3)\ \ 1\ \ (\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3))) \Rightarrow_\beta \ldots$

ixed Point Combinators — Example...

$\mathsf{F} \equiv (\lambda f.(\lambda n.\mathsf{if}\ \ (\mathsf{zero}\ \ n)\ \ 1\ \ (\mathsf{mult}\ \ n\ \ (f\ \ (\mathsf{pred}\ \ n)))))$
$\mathsf{K} \equiv (\lambda x.(F\ \ (x\ \ x)))$
$\mathsf{if} \equiv (\lambda l.(\lambda m.(\lambda n.((l\ \ m)\ \ n))))$
$\mathsf{false} \equiv (\lambda t.(\lambda f.f))$

$((K\ \ K)\ \ 3) \Rightarrow_\beta^* \mathsf{if}\ \ (\mathsf{zero}\ \ 3)\ \ 1\ \ (\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3))) \Rightarrow_\beta$

$((\lambda l.(\lambda m.(\lambda n.((l\ \ m)\ \ n))))\ \ (\mathsf{zero}\ \ 3)\ \ 1\ \ (\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3)))) \Rightarrow_\beta^*$

$(\mathsf{zero}\ \ 3)\ \ 1\ \ (\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3) \Rightarrow_\delta$

$\mathsf{false}\ \ 1\ \ (\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3) =$

$((\lambda t.(\lambda f.f))\ \ 1\ \ (\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3)) \Rightarrow_\beta \ldots$

Fixed Point Combinators — Example...

$\mathsf{F} \equiv (\lambda f.(\lambda n.\mathsf{if}\ \ (\mathsf{zero}\ \ n)\ \ 1\ \ (\mathsf{mult}\ \ n\ \ (f\ \ (\mathsf{pred}\ \ n)))))$
$\mathsf{K} \equiv (\lambda x.(F\ \ (x\ \ x)))$

$((K\ \ K)\ \ 3) \Rightarrow_\beta^* ((\lambda t.(\lambda f.f))\ \ 1\ \ (\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3)) \Rightarrow_\beta$

$\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ (\mathsf{pred}\ \ 3)) \Rightarrow_\delta$

$\mathsf{mult}\ \ 3\ \ ((K\ \ K)\ \ 2) =$

$\mathsf{mult}\ \ 3\ \ (((\lambda x.(F\ \ (x\ \ x)))\ \ K)\ \ 2) \Rightarrow_\beta$

$\mathsf{mult}\ \ 3\ \ ((F\ \ (K\ \ K))\ \ 2) =$

$\mathsf{mult}\ \ 3\ \ (((\lambda f.(\lambda n.\mathsf{if}\ \ (\mathsf{zero}\ \ n)\ \ 1\ \ (\mathsf{mult}\ \ n\ \ (f\ \ (\mathsf{pred}\ \ n)))))\ \ (K\ \ K))\ \ 2) \Rightarrow_\beta^* 6$

Fixed Point Functions in Haskell

```
fix :: (a -> a) -> a
fix f = f (fix f)

fact :: Integer->Integer
fact = fix (factfn)

factfn :: Num a => (a -> a) -> a -> a
factfn f n = if n==0 then 1 else n*f(n-1)

> fact 3
6
```


Fixed Point Functions in Haskell. . .

```
fix f = f (fix f)
fact = fix (factfn)
factfn f n = if n==0 then 1 else n*f(n-1)

fact 3 ⇒
   (\ f n -> if n==0 then 1
         else n*f(n-1)) (fix f) 3 ⇒
   (if 3==0 then 1 else 3*(fix f)(3-1)) ⇒
   (3*((\f n -> if n==0 then 1
         else n*f(n-1))(fix f)(3-1)) ⇒
               ··· ⇒
   (3*(2*(1*1))) ⇒ 6
```


Readings and References

- Read pp. 156–157, in *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz, http://www.cs.uiowa.edu/˜slonnegr/plf/Book.
- Read pp. 618–621, in Scott.

Acknowledgments

- Much of the material in this lecture on Lambda Calculus is taken from the book *Syntax and Semantics of Programming Languages*, by Ken Slonneger and Barry Kurtz, http://www.cs.uiowa.edu/˜slonnegr/plf/Book.