

# The $\lambda$ -calculus

John David Stone

February 16, 2009

The  $\lambda$ -calculus has three aspects. It is a notation for describing the construction and application of functions, an axiom system for proving the equivalence of functions, and a programming language that can be used to express computational algorithms that can be executed by applying functions. We'll examine these aspects in order.

## Specifying the syntax

The abstract syntax of the  $\lambda$ -calculus is admirably concise and simple: An expression is either an identifier, a  $\lambda$ -expression comprising a parameter and a body, or an application comprising an operator expression and one operand expression.

In the literature on the  $\lambda$ -calculus, several different concrete syntaxes are employed. Friedman and Wand (p. 9) provide one that makes the  $\lambda$ -calculus look like a very limited version of Scheme. This is reassuringly familiar, but it strikes some authors as a little cumbersome. The notation that Alonzo Church originally used when he came up with the notation back in the 1930s is more concise, but in practice there is some variation in the conventions about when you have to use parentheses and when you can leave them out, and in addition it uses a Greek letter, which isn't in the ASCII character set.

Let's dispose of the Greek letter first. Church chose ' $\lambda$ ' in the first place because he was adapting an older notation in which one formed an abstraction by writing a circumflex over the parameter, thus:  $\hat{x}$ . Church's typist had a machine that would do Greek letters, but made heavy weather of circumflexes. He chose  $\lambda$  because it looked kind of like a circumflex, lowered to the typescript baseline. Since our technology makes circumflexes easier

than Greek letters, let's reverse his decision and begin our  $\lambda$ -expressions with circumflexes.

The simplest way to regularize the use of parentheses is to adopt the fixed convention that *every* application be enclosed in parentheses. This gives us the following set of BNF rules:

```
<expression> ::= <identifier>
                | ^ <identifier> . <expression>
                | ( <expression> <expression> )
```

Let's specify that an identifier must begin with a letter and may contain letters, digits, hyphens, and question marks. We'll borrow the usual conventions about whitespace and comments from the languages in *Essentials of programming languages*.

## The universe of the $\lambda$ -calculus

In the *pure*  $\lambda$ -calculus, the initial environment is empty, so although you have single identifiers as expressions, they aren't independently meaningful. The usual convention is to suppose that identifiers that are not bound in any other way are bound to *themselves*, so that, for instance, a free-standing identifier `x` just signifies itself—as a symbol, if you like.

The simplest expressions that have genuine meanings in the pure  $\lambda$ -calculus, though, are  $\lambda$ -expressions in which the body matches the parameter, such as `^x.x`. All such expressions denote the identity function, which returns without change whatever value it is given as its argument. It doesn't make any difference what identifier is used, as long as the parameter is the same as the body; `^x.x` and `^given.given` are exactly the same function.

A slightly more complicated  $\lambda$ -expression is `^x.^y.x`, which denotes the constant-function builder: Give it any value `x`, and it will return a function that takes any value `y` and returns `x`—a constant function. (Again, the particular identifiers that appear in this expression don't matter; the expression `^given.^ignored.given` has exactly the same value.)

Functions like these—unary functions that take unary functions as arguments and return unary functions as values—are the only data type in the pure  $\lambda$ -calculus. Since everything belongs to the same type, there is no possibility of committing a type error; we can apply any unary function to any unary function whatsoever.

## Substitution

To explain the axioms and inference rules that we'll use to determine when two expressions of the  $\lambda$ -calculus denote the same function, we first need a notion of *substitution*. This is a purely syntactic operation in which we go through an expression, looking for free occurrences of an identifier; every time we find one, we remove it and put in its place a new expression—the same new expression every time.

Let's write " $[N/x]M$ " to denote the result of substituting an expression  $N$  for every free occurrence of an identifier  $x$  in an expression  $M$ . Here's a formal specification of this concept of substitution:

- For any expression  $N$  and any identifier  $x$ ,  $[N/x]x$  is  $N$ .
- For any expression  $N$  and any identifiers  $x$  and  $y$ , where  $y$  is not the same identifier as  $x$ ,  $[N/x]y$  is  $y$ .
- For any expressions  $N$ ,  $M_1$ , and  $M_2$  and any identifier  $x$ ,  $[N/x](M_1 M_2)$  is  $([N/x]M_1 [N/x]M_2)$ .
- For any expressions  $N$  and  $M$  and any identifier  $x$ ,  $[N/x]\hat{x}.M$  is  $\hat{x}.M$ .
- For any expressions  $N$  and  $M$  and any identifiers  $x$  and  $y$ , where  $y$  is not the same identifier as  $x$  and  $y$  occurs free in  $N$  and  $x$  occurs free in  $M$ ,  $[N/x]\hat{y}.M$  is  $\hat{z}.[N/x][z/y]M$ , where  $z$  is any identifier that does not occur free in either  $M$  or  $N$ .
- For any expressions  $N$  and  $M$  and any identifiers  $x$  and  $y$ , where  $y$  is not the same identifier as  $x$  and either  $y$  does not occur free in  $N$  or  $x$  does not occur free in  $M$ ,  $[N/x]\hat{y}.M$  is  $\hat{y}.[N/x]M$ .

In our interpreter, the substitution operation takes this form:

```
;; substitute : Expression * Identifier * Expression -> Expression
(define substitute
  (lambda (new old exp)
    (let (replacer ((subexp exp))
          (cases expression subexp
            (variable (id)
              (if (eqv? id old) new subexp))
```

```

(abstraction (parameter body)
  (cond ((equiv? parameter old) subexp)
        ((and (occurs-free? parameter new)
              (occurs-free? old subexp))
         (let ((new-id (gensym)))
           (abstraction
            new-id
            (replacer
             (substitute (variable new-id) parameter body))))))
        (else
         (abstraction parameter (replacer body))))))
(application (operator operand)
  (application (replacer operator) (replacer operand))))))

```

The `gensym` procedure delivers a new identifier each time it is invoked, so we invoke it when we need one that does not occur free in any of the expressions we're dealing with.

## Axioms for the $\lambda$ calculus

Here are the axioms and inference rules for the pure  $\lambda$  calculus (or at least the version of it that we'll be studying—there are variations).

- ( $\alpha$ ) For any identifier  $x$ , any expression  $M$ , and any identifier  $y$  that does not occur free in  $M$ ,  $\hat{x}.M = \hat{y}.[y/x]M$ .
- ( $\beta$ ) For any identifier  $x$  and any expressions  $M$  and  $N$ ,  $(\hat{x}.M N) = [N/x]M$ .
- ( $\rho$ ) For any expression  $M$ ,  $M = M$ .
- ( $\mu$ ) For any expressions  $M$ ,  $M'$ , and  $N$ , if  $M = M'$ , then  $(N M) = (N M')$ .
- ( $\nu$ ) For any expressions  $M$ ,  $M'$ , and  $N$ , if  $M = M'$ , then  $(M N) = (M' N)$ .
- ( $\xi$ ) For any expressions  $M$  and  $M'$  and any identifier  $x$ , if  $M = M'$ , then  $\hat{x}.M = \hat{x}.M'$ .
- ( $\sigma$ ) For any expressions  $M$  and  $N$ , if  $M = N$ , then  $N = M$ .

- $(\tau)$  For any expressions  $M$ ,  $N$ , and  $P$ , if  $M = N$  and  $N = P$ , then  $M = P$ .

Most of the axioms and rules on this list sound obvious and even trivial.  $(\rho)$ ,  $(\sigma)$ , and  $(\tau)$  say that equality of functions is reflexive, symmetric, and transitive, and  $(\mu)$ ,  $(\nu)$ , and  $(\xi)$  say that equals can be substituted for equals inside any composite expression. The axiom  $(\alpha)$  registers the point that it makes no difference what identifier you use as the parameter; if you systematically replace every bound occurrence of the parameter with a different identifier and then make that identifier the parameter, you get the same function over again. The only exceptions are identifiers that occur free in the body to begin with—those identifiers have to remain free to preserve the identity of the denoted function.

Thus most of the interest in this axiom system lies in axiom  $(\beta)$ , which in effect tells you what happens when you apply a function described by a  $\lambda$ -expression to an argument: The argument takes the place of the parameter in the body of the  $\lambda$ -expression.

To give a little of the flavor of this system, let's use it to prove that  $(\hat{x}.\hat{y}.x \hat{x}.x) = \hat{x}.\hat{y}.y$ :

We begin with proof with a special case of axiom  $(\beta)$ :

$$(\hat{x}.\hat{y}.x \hat{x}.x) = \hat{y}.\hat{x}.x. \quad (1)$$

Now, by axiom  $(\alpha)$ ,

$$\hat{y}.\hat{x}.x = \hat{z}.\hat{x}.x, \quad (2)$$

and so, by rule  $(\tau)$ ,

$$(\hat{x}.\hat{y}.x \hat{x}.x) = \hat{z}.\hat{x}.x. \quad (3)$$

Again, by  $(\alpha)$ ,

$$\hat{x}.x = \hat{y}.y, \quad (4)$$

and hence, by  $(\xi)$ ,

$$\hat{z}.\hat{x}.x = \hat{z}.\hat{y}.y, \quad (5)$$

so that, by  $(\tau)$  applied to equations (3) and (5),

$$(\hat{x}.\hat{y}.x \hat{x}.x) = \hat{z}.\hat{y}.y. \quad (6)$$

One more instance of  $(\alpha)$ ,

$$\hat{z}.\hat{y}.y = \hat{x}.\hat{y}.y, \quad (7)$$

and one more application of rule  $(\tau)$  yields the theorem:

$$(\hat{x}.\hat{y}.x \hat{x}.x) = \hat{x}.\hat{y}.y. \quad (8)$$

## $\beta$ -reduction

The use of the  $(\beta)$  axiom to move from the left-hand side of the equation to the right is somewhat similar to the process of evaluating a function call. This transition, in which an expression of the form  $(\hat{x}.M N)$  is replaced by  $[N/x]M$ , is called  $\beta$ -reduction, and it is the  $\lambda$ -calculus representation of a *step in a computation*. When we look at the  $\lambda$ -calculus as a programming language, a “program” is simply an expression, and we “execute” the program by performing  $\beta$ -reductions on its subexpressions. An expression in which no subexpression is  $\beta$ -reducible is called a *normal form*; when a computation reaches a normal form, we have to stop. But we know that the expression that we reach at the end of a computation denotes the same function as the original program did; it’s easy to construct the proof by linking up all the stages in the computation by appropriate instances of axiom  $(\beta)$  and rules  $(\mu)$ ,  $(\nu)$ ,  $(\xi)$ , and  $(\tau)$ .

We’ll regard the normal form of a program as its result or output. In other words, the point of the computation is simply to transform the program to normal form by performing  $\beta$ -reductions on it; the normal form is our “answer.”

## Reduction orders

It is possible for an expression of the  $\lambda$ -calculus to contain two or more  $\beta$ -*redexes*—subexpressions to which  $\beta$ -reduction could be applied. Here’s one that has three:

$$((\hat{x}.\hat{y}.x \hat{z}.z) \hat{w}.(w x)) (\hat{x}.\hat{y}.x \hat{z}.(z z))$$

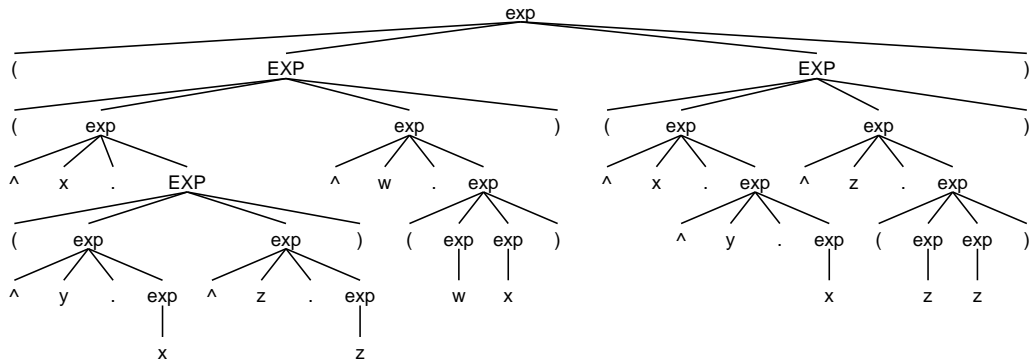


Figure 1: Concrete syntax tree for  $((\lambda x.(\lambda y.x \lambda z.z) \lambda w.(w x)) (\lambda x.\lambda y.x \lambda z.(z z)))$

Figure 1 shows the syntax tree for this expression. The nodes at which  $\beta$ -reduction is possible are the ones for which the label EXP appears in capital letters.

The subexpression  $(\lambda y.x \lambda z.z)$  is a  $\beta$ -redex, so we could start by reducing it (which would give simply  $x$ ). Also, the subexpression  $(\lambda x.(\lambda y.x \lambda z.z) \lambda w.(w x))$  is a  $\beta$ -redex; we could reduce it first, to get  $(\lambda y.\lambda w.(w x) \lambda z.z)$ . Finally, over at the right,  $(\lambda x.\lambda y.x \lambda z.(z z))$  is a  $\beta$ -redex; we could just as easily begin by reducing it to  $\lambda y.\lambda z.(z z)$ . So where should we start?

## Applicative order

One approach is suggested by the analogy between  $\beta$ -reduction, considered as a computational step, and the evaluation of a procedure call, with the operand as the “argument” that takes the place of the parameter in the body of the operator. In programming languages, we commonly evaluate both the operator and the operand in a procedure call before starting to evaluate the call itself. Suppose, then, that we have our interpreter traverse the program’s syntax tree, processing the nodes in *postorder*, so that the operator and operand in any application have to be completely evaluated before we try to reduce the application itself? This strategy is called *applicative-order reduction*.

It sometimes happens that the result of a  $\beta$ -reduction is another  $\beta$ -redex; for instance, in the example above, we reduced  $(\lambda x.(\lambda y.x \lambda z.z) \lambda w.(w x))$

```

;; eval-expression : Expression -> Expression
(define eval-expression
  (lambda (exp)
    (cases expression exp
      (variable (id) exp)
      (abstraction (parameter body) exp)
      (application (operator operand)
        (let ((reduced-rator (eval-expression operator))
              (reduced-rand (eval-expression operand)))
          (cases expression reduced-rator
            (variable (id)
              (application reduced-rator reduced-rand))
            (abstraction (id body)
              (eval-expression (substitute reduced-rand id body)))
            (application (operator operand)
              (application reduced-rator reduced-rand))))))))))

```

Figure 2: Applicative-order reduction

to  $(\hat{y}.\hat{w}.(w\ x)\ \hat{z}.z)$ , which is a  $\beta$ -redex. We could  $\beta$ -reduce again to get  $\hat{w}.(w\ x)$ . (Since  $y$  does not occur at all in  $\hat{w}.(w\ x)$ , “substituting”  $\hat{z}.z$  for it has no effect.) This expression is in normal form, so we’ve reached a stopping point for this particular series of reductions. In general, though, it may require any number of steps to reach a normal form (if there is one). When we start to evaluate an subexpression in the applicative-order strategy, we’ll keep working away at it until we reach a normal form, no matter how many steps it takes.

There are two variants of this strategy. If we think of  $\lambda$ -expressions as being just like `proc`-expressions in PROC, we would not try to evaluate any part of the body of a  $\lambda$ -expression until it is needed for a  $\beta$ -reduction. In traversing the syntax tree, we explore the subtrees of `application` nodes, but skip those of `abstraction` nodes. Figure 2 shows an expression evaluator that implements this strategy.

Since our “evaluation” operations are entirely syntactic, we don’t need to maintain an environment in which identifiers are bound to values, so `eval-expression` needs only one argument. A lone identifier evaluates to itself, and we also keep abstractions intact, not trying to evaluate their bodies. The applications are the interesting cases. We first evaluate the operator

$$\begin{aligned}
& ((\lambda x. (\lambda y. x \ z. z) \ \lambda w. (w \ x)) (\lambda x. \lambda y. x \ \lambda z. (z \ z))) \\
\implies & ((\lambda y. \lambda w. (w \ x) \ \lambda z. z) (\lambda x. \lambda y. x \ \lambda z. (z \ z))) \\
\implies & (\lambda w. (w \ x) (\lambda x. \lambda y. x \ \lambda z. (z \ z))) \\
\implies & (\lambda w. (w \ x) \ \lambda y. \lambda z. (z \ z)) \\
\implies & (\lambda y. \lambda z. (z \ z) \ x) \\
\implies & \lambda z. (z \ z)
\end{aligned}$$

Figure 3: Applicative-order computation of our sample program

and operand, then check whether the resulting value of the operator is an abstraction; if it is, we can do a  $\beta$ -reduction. We call **eval-expression** recursively in case the first  $\beta$ -reduction makes another  $\beta$ -reduction possible.

If the operator's value is not an abstraction, we construct a new application from the value of the operator and the value of the operand and return it. Since it is not a  $\beta$ -redex, there is no way to simplify it at this level.

If we apply this evaluation strategy to our sample expression, the  $\beta$ -redex that is selected for first reduction is  $(\lambda x. (\lambda y. x \ \lambda z. z) \ \lambda w. (w \ x))$ . (You might think that  $(\lambda y. x \ \lambda z. z)$  would be first, but notice that it's inside the body of a  $\lambda$ -expression, so it's skipped during this initial traversal.) The steps in the full applicative-order evaluation are shown in Figure 3.

This strategy turns out to be unsatisfactory from at least one point of view: The reduced expression that **eval-expression** returns is not always in normal form! For instance, consider the program  $(\lambda f. \lambda a. (f \ a) \ \lambda s. (s \ s))$ . The interpreter shown above performs one  $\beta$ -reduction and gets  $\lambda a. (\lambda s. (s \ s) \ a)$ , instead of proceeding all the way to the normal form  $\lambda a. (a \ a)$ . The problem is precisely that we skip over the bodies of  $\lambda$ -expressions; a  $\beta$ -redex that shows up inside the body of a  $\lambda$ -expression doesn't get reduced away.

## Partial evaluation

So we might do better to adopt the slightly different strategy of traversing every node of the syntax tree, performing  $\beta$ -reductions wherever possible, even inside the bodies of procedures. In a programming language, this corresponds to *partial evaluation*—a kind of optimization in which any computations in the body of a procedure definition that can be performed without knowing the arguments are done when the procedure value, the closure, is

```

;; eval-expression : Expression -> Expression
(define eval-expression
  (lambda (exp)
    (cases expression exp
      (variable (id) exp)
      (abstraction (parameter body)
        (abstraction parameter (eval-expression body)))
      (application (operator operand)
        (let ((reduced-rator (eval-expression operator))
              (reduced-rand (eval-expression operand)))
          (cases expression reduced-rator
            (variable (id)
              (application reduced-rator reduced-rand))
            (abstraction (parameter body)
              (eval-expression
                (substitute reduced-rand parameter body)))
            (application (operator operand)
              (application reduced-rator reduced-rand))))))))))

```

Figure 4: Applicative-order reduction with partial evaluation

formed. Partial evaluation can greatly speed up a procedure that is invoked many times, because the “pre-calculations” are done only once, at definition, rather than as part of each invocation.

Figure 4 shows a “partial evaluation” interpreter for the pure  $\lambda$ -calculus. The only change is that we now evaluate the body of a **abstraction** when we encounter it in the syntax tree.

Under this strategy, we really are doing a complete post-order traversal of the syntax tree, and  $\beta$ -redexes nearer the leaves always take precedence over the ones higher up on the branches of the tree. The evaluation of our sample expression under this strategy is shown in Figure 5.

Unfortunately, this approach also has a flaw: It sometimes fails to terminate, never reaching a normal form, even though the expression has a normal form that one can obtain by applying  $\beta$ -reductions in a different order. Here’s a simple example of such an expression:

$$((\hat{x}.\hat{y}.y (\hat{s}.(s s) \hat{s}.(s s))) \hat{z}.z)$$

The middle part of this expression,  $(\hat{s}.(s s) \hat{s}.(s s))$ , is a dangerous

$$\begin{aligned}
& ((\hat{x}.(\hat{y}.x \hat{z}.z) \hat{w}.(w x)) (\hat{x}.\hat{y}.x \hat{z}.(z z))) \\
\implies & ((\hat{x}.x \hat{w}.(w x)) (\hat{x}.\hat{y}.x \hat{z}.(z z))) \\
\implies & (\hat{w}.(w x) (\hat{x}.\hat{y}.x \hat{z}.(z z))) \\
\implies & (\hat{w}.(w x) \hat{y}.\hat{z}.(z z)) \\
\implies & (\hat{y}.\hat{z}.(z z) x) \\
\implies & \hat{z}.(z z)
\end{aligned}$$

Figure 5: Partial-evaluator computation of our sample expression

non-terminating expression in which the self-application function is applied to itself. This is one of two  $\beta$ -redexes in the expression; the other is  $(\hat{x}.\hat{y}.y (\hat{s}.(s s) \hat{s}.(s s)))$ , which  $\beta$ -reduces to  $\hat{y}.y$ ; in this case we never have to evaluate the non-terminating subexpression at all—it just drops out. To reach the normal form for this expression,  $\hat{z}.z$ , we have to choose the correct  $\beta$ -redex at the first step. But the rule for applicative order says that we have to evaluate the operand in an application before we can start the “procedure call” itself.

## Normal order

Is there an algorithm that we can follow to derive a normal form from any expression that has one? The answer is yes; it’s called *normal-order reduction*.

In a way, this strategy is just the opposite of application-order reduction: The  $\beta$ -redex that we select for reduction first must be *maximal*, in the sense that it is not contained in any other  $\beta$ -redex. (If you think about the syntax tree, a maximal  $\beta$ -redex is rooted at a node that has no  $\beta$ -redex node above it on the same branch.) Sometimes, as in our initial example, there are two or more maximal  $\beta$ -redexes. In such a case, the normal-order strategy is to choose whichever one of these is farthest to the left in the expression.

Implementing the normal-order interpreter is a little more difficult. In some cases, reducing a subexpression at a lower level of the syntax tree can cause a  $\beta$ -redex to appear at a higher level; this means that we can’t just keep  $\beta$ -reducing a subexpression until it is in normal form, because the subexpression may cease to be maximal before it reaches normal form, and in that case we have to move up in the tree in order to work on the new

leftmost maximal  $\beta$ -redex.

Figure 6 shows the implementation. The `reduce-leftmost-maximal` procedure performs at most one  $\beta$ -reduction, on the leftmost maximal  $\beta$ -redex; it returns `#f` if the expression that it is given has no  $\beta$ -redexes in it. The `eval-expression` procedure simply invokes `reduce-leftmost-maximal` over and over again until all of the  $\beta$ -redexes are gone.

The evaluation of our sample expression in normal order is shown in Figure 7.

In practice, applicative-order reduction tends to be more efficient than normal-order reduction, since the cost of evaluating duplicates of the same operand when it is substituted in at two or more points in the body of a  $\lambda$ -expression generally outweighs the benefit of not having to evaluate an operand at all when the identifier for which it is substituted does not occur free in that body. However, with normal-order reduction, you get the weak guarantee that a normal form will be reached if there is one. (This *standardization theorem* is due to Haskell Curry.)

## The $\lambda$ -calculus universe

The pure  $\lambda$ -calculus directly supports only one data type. The notation seems to imply a peculiarly self-contained universe, consisting of unary functions that take unary functions as arguments and return unary functions as values.

Some of these functions are familiar, or at least identifiable:  $\hat{x}.x$ , the identity function; the constant-function builder  $\hat{x}.\hat{y}.x$ , which takes any unary function  $x$  and returns a unary function that returns  $x$  no matter what it is given; the function  $\hat{x}.\hat{y}.y$  that returns the identity function no matter what it is given. Others are a little more exotic, such as the self-application function  $\hat{x}.(x\ x)$ , which takes any unary function  $x$  and returns the result of applying it to itself, and the fixed-point finder  $\hat{f}.( \hat{x}.(f\ (x\ x))\ \hat{x}.(f\ (x\ x)))$ , which we'll discuss at length a little later on.

To sustain our interest, however, the  $\lambda$ -calculus needs some closer connection with functions involving more familiar kinds of data: Booleans, natural numbers, and such like. One way to achieve this is to add, by simple fiat, names for such data and the functions that operate on them. This approach yields *applied  $\lambda$ -calculus*.

That's the slacker's way, though. The clear duty of every *real* programmer is to get in there and *build* the missing data values and operations out of the

```

;; reduce-leftmost-maximal : Expression -> SchemeVal
(define reduce-leftmost-maximal
  (lambda (exp)
    (cases expression exp
      (variable (id) #f)
      (abstraction (parameter body)
        (let ((reduced (reduce-leftmost-maximal body)))
          (if reduced
              (abstraction parameter reduced)
              #f)))
      (application (operator operand)
        (cases expression operator
          (variable (id)
            (let ((reduced-rand (reduce-leftmost-maximal operand)))
              (if reduced-rand
                  (application operator reduced-rand)
                  #f)))
          (abstraction (parameter body)
            (substitute operand parameter body))
          (application (inner-rator inner-rand)
            (let ((reduced-rator (reduce-leftmost-maximal operator)))
              (if reduced-rator
                  (application reduced-rator operand)
                  (let ((reduced-rand (reduce-leftmost-maximal operand)))
                    (if reduced-rand
                        (application operator reduced-rand)
                        #f))))))))))

;; eval-expression : Expression -> Expression
(define eval-expression
  (lambda (exp)
    (let ((reduced (reduce-leftmost-maximal exp)))
      (if reduced
          (eval-expression reduced)
          exp)))

```

Figure 6: Normal-order reduction

$$\begin{aligned}
& ((\hat{x}.(\hat{y}.x \hat{z}.z) \hat{w}.(w x)) (\hat{x}.\hat{y}.x \hat{z}.(z z))) \\
\implies & ((\hat{y}.\hat{w}.(w x) \hat{z}.z) (\hat{x}.\hat{y}.x \hat{z}.(z z))) \\
\implies & (\hat{w}.(w x) (\hat{x}.\hat{y}.x \hat{z}.(z z))) \\
\implies & ((\hat{x}.\hat{y}.x \hat{z}.(z z)) x) \\
\implies & (\hat{y}.\hat{z}.(z z) x) \\
\implies & \hat{z}.(z z)
\end{aligned}$$

Figure 7: Normal-order computation of our sample expression

materials at hand—or, more precisely, to build models of them within the  $\lambda$ -calculus universe. We shall treat these more familiar kinds of data as abstract data types and look for ways to implement them, using only unary functions as primitives.

## Polyadic functions

We can begin by modeling functions of two or more arguments within the  $\lambda$ -calculus universe of unary functions. The interface is a simple one: We need to be able to construct such functions, on the one hand, and to apply them, on the other.

The simplest implementation was discovered by Moses Schönfinkel in 1924. To represent a function with  $n$  parameters  $x_1, x_2, \dots, x_n$  and body  $M$ , we use the unary function defined by the nested  $\lambda$ -expression  $\hat{x}_1.\hat{x}_2.\dots.\hat{x}_n.M$ .

To apply this function to arguments  $N_1, N_2, \dots, N_n$ , we use correspondingly nested applications:

$$(\dots ((\hat{x}_1.\hat{x}_2.\dots.\hat{x}_n.M N_1) N_2) \dots N_n).$$

For instance, the function that takes four arguments and returns the result of applying the third of them to the first is constructed as

$$\hat{x}_1.\hat{x}_2.\hat{x}_3.\hat{x}_4.(x_3 x_1),$$

and to apply this to four copies of the identity function, we could write

$$(((\hat{x}_1.\hat{x}_2.\hat{x}_3.\hat{x}_4.(x_3 x_1) \hat{y}.y) \hat{y}.y) \hat{y}.y) \hat{y}.y)$$

We can now recognize the constant-function former  $\hat{x}.\hat{y}.x$  and the constant-identity function  $\hat{x}.\hat{y}.y$  as the implementations of the *binary projection functions*: Each of them takes two arguments,  $x$  and  $y$ ; the constant-function former returns  $x$  and the constant-identity function returns  $y$ . In general, we can write the projection function that returns the  $j$ th of  $n$  arguments as  $\hat{x}_1.\hat{x}_2.\dots.\hat{x}_n.x_j$ .

## Booleans

For a Boolean data type, we need two distinct values to play the roles of **true** and **false** and a mechanism for building conditionals that respond differently to those values. The binary projection functions are plausible candidates for **true** and **false**, since they are easily distinguished by the effects they have when applied and have a useful kind of symmetry.

The function that distinguishes them should take three arguments, to play the roles of condition, consequent, and alternative in a conditional expression. We expect the condition to be either **true** or **false**, and we want **true** to select the consequent and **false** the alternative. Easily done: We can just apply the value of the condition successively to the consequent and the alternative, thus:

```
((condition consequent) alternative)
```

If *condition* is **true**, it will select *consequent*, and if it is **false**, it will select *alternative*, as required. So we officially adopt the following definitions:

```

true  ≡   $\hat{x}.\hat{y}.x$ 
false ≡   $\hat{x}.\hat{y}.y$ 
if    ≡   $\hat{condition}.\hat{consequent}.\hat{alternative}.$ 
      ((condition consequent) alternative)

```

Implementing **if** as a function means that we cannot use applicative-order evaluation on applications of it if we expect it to work as a control structure. From the programmer's point of view, the purpose of an **if**-expression is to choose between evaluating the consequent and evaluating the alternative, bypassing the evaluation of the one not selected. An applicative-order interpreter would always evaluate both. Since we've chosen to use a

normal-order interpreter anyway, we don't have to make an exception to our general strategy for expressions containing `if`.

Other Boolean operators can now be defined in terms of `true`, `false`, and `if`:

```
and ≡ ^left.^right.(((if left) right) false)
not ≡ ^negand.(((if negand) false) true)
or  ≡ ^left.^right.(((if left) true) right)
```

## Natural numbers

Natural numbers have a richer structure and a more complicated interface. To begin with, we need a representation for the natural number zero and a way of forming the successor of any given natural number. In Alonzo Church's original development of the  $\lambda$ -calculus, he took the primary application of natural numbers to be counting the number of times some function  $f$  was applied iteratively to some initial value  $x$ . So he proposed the following system of numeration:

```
zero  ≡ ^f.^x.x
one   ≡ ^f.^x.(f x)
two   ≡ ^f.^x.(f (f x))
three ≡ ^f.^x.(f (f (f x)))
...   ...
```

The functions in this series are called the *Church numerals*.

The definition of `successor` that fits the Church numerals takes any Church numeral  $n$  as an argument and returns a function that applies  $f$  to  $x$  one more time than  $n$  itself does:

```
successor ≡ ^n.^f.^x.(f ((n f) x))
```

To test whether a given natural number is zero, we use the function

```
zero? ≡ ^n.((n ^x.false) true)
```

Applying this function to `zero` causes the constant-`false` function to be applied to `true` zero times (*i.e.*, not at all), so that the result is `true`. Applying it to any other Church numeral  $n$  causes the constant-`false` function to be applied  $n$  times in succession, so that the result is `false`.

One of the nice features of the Church numerals is that the definitions of most of the basic arithmetic functions are simple and direct:

```
plus ≡ ^augend.^addend.^f.^x.((augend f) ((addend f) x))
times ≡ ^multiplicand.^multiplier.^f.^x.
      ((multiplicand (multiplier f)) x)
expt ≡ ^base.^exponent.^f.^x.(((exponent base) f) x)
```

## The predecessor function

Some ingenious hacking by a logician named Paul Bernays leads to an appropriate definition of the `predecessor` function for Church numerals. It's easiest to explain if we define it in stages.

We'll begin with what I'll call *choices*; these are functions that respond differently to `zero` and to non-zero Church numerals, just as the functions we construct with `if` respond differently to `true` and `false`. The idea is to store the response to non-zeroes in a constant function that is to be applied some number of times to the response for `zero`. If we apply this constant function zero times, we get the response for `zero`; if we apply it any positive number of times, we get the response for non-zeroes.

```
choice ≡ ^yes.^no.^condition.((condition ^y.no) yes)
```

To compute predecessors, it turns out that we'll need to generate a whole series of choices in which the alternatives are successive Church numerals—`((choice one) zero)`, `((choice two) one)`, `((choice three) two)`, and so on. So we need a higher-order function that takes any choice in this series and produces the next one. I'll call this function `tweak`; here is its definition:

`tweak`  $\equiv$   $\hat{c}.$ ((choice (successor (c zero))) (c zero))

In other words: Given a choice  $c$ , `tweak` builds a new choice, in which the response to `zero` is the successor of  $c$ 's response to `zero`, and the response to any other Church numeral is the same as  $c$ 's response to `zero`.

To compute the predecessor of a positive integer  $n$ , given its Church numeral, the `predecessor` function first applies `tweak`  $n$  times to an initial choice that always returns `zero`, namely ((choice zero) zero). The result of all the tweaking is a choice in which the consequent is the Church numeral for  $n$  and the alternative is the Church numeral for  $n - 1$ . Finally, this choice is applied to `one` so that the alternative is extracted.

`predecessor`  $\equiv$   $\hat{n}.$ ((n tweak) ((choice zero) zero)) one)

## Binding and scope

In the  $\lambda$ -calculus, one can get local bindings, effectively like those in a `let`-expression, by making the bound variables into parameters in abstractions, thus:

```
( $\hat{\text{successor}}$ .
  ( $\hat{\text{zero}}$ .
    (successor (successor (successor zero)))
     $\hat{f}.$  $\hat{x}.$ x)
   $\hat{n}.$  $\hat{f}.$  $\hat{x}.$ (f ((n f) x)))
```

This says, in effect, “Let `successor` be  $\hat{n}.$  $\hat{f}.$  $\hat{x}.$ (f ((n f) x)) and let `zero` be  $\hat{f}.$  $\hat{x}.$ x in (successor (successor (successor zero))).” The process of normal-order  $\beta$ -reduction ensures that the computation begins with the replacement of the locally bound identifiers with their expansions, before the evaluation of the body begins.

One limitation of using  $\beta$ -reduction as a model for calling a procedure is that it doesn't seem to provide any mechanism for binding an identifier to a *recursive* function—one whose body includes a call to the very same function. Indeed, it's not obvious at first glance that the universe of the

$\lambda$ -calculus includes any functions that would have to be recursively defined. Even the odd-looking self-application function  $\lambda s.(s\ s)$  doesn't have anything recursive about it.

Nonetheless, there are recursively defined functions in the universe of the  $\lambda$ -calculus, and we can give algorithms for computing them; we just need a different, slightly trickier approach.

## Fixed points

This approach begins with the notion of a *fixed point*. A fixed point  $p$  of a given function  $f$  is an argument for which the following equation holds:  $(f\ p) = p$ . This notion may be familiar from the theory of functions of real numbers; for instance, 0 and 1 are fixed points of the square function in the domain of reals, and  $1/2$  is a fixed point of the function  $f(x) = 1 - x$ . In the domain of reals, a function may have any number of fixed points, including none ( $f(x) = x + 1$ ) or infinitely many ( $f(x) = |x|$ ).

Let's look at the fixed points of some of our good friends from the  $\lambda$ -calculus universe. Of course, by definition, absolutely everything is a fixed point of the identity function  $\lambda x.x$ . The constant-identity-returner  $\lambda x.\lambda y.y$  has the identity function as a fixed point—it returns identity if you give it identity (or anything else). Indeed, for any function  $f$ ,  $f$  is a fixed point of the constant function  $\lambda x.f$  that always returns  $f$ —again, that's true by definition.

One of the remarkable features of the  $\lambda$ -calculus, it turns out, is that *every* function in it has at least one fixed point—sometimes infinitely many, like  $\lambda x.x$ , but always at least one. For instance, the constant-function-former  $\lambda x.\lambda y.x$  has a fixed point, which is a function that, given any argument whatever, returns *itself* as value. If we try to write a  $\lambda$ -expression for such a function, we quickly run up against the same problem that recursion poses: We want the *body* of our  $\lambda$ -expression to be the *whole* of the very  $\lambda$ -expression that we're writing! It happens that the fixed point of the constant-function-former doesn't have a normal form, so we're going to have to approach the problem of computing it somewhat indirectly.

The fact that every function has a fixed point in the  $\lambda$ -calculus might seem to conflict with the fact that we can model natural numbers in the  $\lambda$ -calculus. How can, say, the **successor** function possibly have a fixed point? The answer is that there are many inhabitants of the  $\lambda$ -calculus universe that

won't have any role to play in our model of the natural numbers, and any fixed point of the successor function is one of those.

## The fixed-point finder

An even more remarkable feature of the  $\lambda$ -calculus is that it provides a *fixed-point finder*—a function that takes any function as argument and returns a fixed point for that function. Indeed, there are infinitely many fixed-point finders. Here is a very simple one, known in the literature as  $Y$ :

$$\hat{f}.(\hat{x}.(f (x x)) \hat{x}.(f (x x)))$$

$Y$  does not have a normal form, unfortunately, so there isn't really any simpler or better way to express it. You have to get a sense of what it does from examples. Here is the computation that ensues when we run the expression in which  $Y$  is applied to our constant-identity function, using normal-order evaluation:

$$\begin{aligned} & (\hat{f}.(\hat{x}.(f (x x)) \hat{x}.(f (x x))) \hat{x}.\hat{y}.y) \\ \implies & (\hat{x}.( \hat{x}.\hat{y}.y (x x)) \hat{x}.( \hat{x}.\hat{y}.y (x x))) \\ \implies & (\hat{x}.\hat{y}.y (\hat{x}.( \hat{x}.\hat{y}.y (x x)) \hat{x}.( \hat{x}.\hat{y}.y (x x)))) \\ \implies & \hat{y}.y \end{aligned}$$

The effect of the first  $\beta$ -reduction is to put the function whose fixed point we are seeking in for  $f$ . Notice the structure of the second step—it is an instance of self-application, and the function that is being self-applied is one that takes any argument  $x$  and applies our constant-identity function to the result of applying  $x$  to itself. In other words, the function that is being self-applied is one that involves both the function whose fixed point we're seeking and an algorithm for regenerating itself if necessary.

In this case, the regeneration *isn't* necessary, because under normal-order evaluation we don't even have to look at the argument to the constant-identity function in order to figure out what to return. We make the transition from the third to the fourth step of the derivation without ever evaluating the operand.

In finding the fixed point of some other function, however, we might need to regenerate the self-application. Here's the beginning of a normal-order evaluation of the expression for computing the fixed point of the constant-forming function  $\hat{x}.\hat{y}.x$ :

```

    (λf.(λx.(f (x x)) λx.(f (x x))) λx.λy.x)
==> (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x)))
==> (λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
==> λy.(λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x)))
==> λy.(λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
==> λy.λy.(λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
==> λy.λy.λy.(λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
==> λy.λy.λy.λy.(λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
==> λy.λy.λy.λy.λy.(λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
==> λy.λy.λy.λy.λy.λy.(λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
==> λy.λy.λy.λy.λy.λy.λy.(λx.λy.x (λx.(λx.λy.x (x x)) λx.(λx.λy.x (x x))))
...

```

Since we're using  $\beta$ -reduction, all of these expressions denote the same function, and by comparing the second and fourth lines you can see that it is indeed a function that takes any argument  $y$  and returns itself. The following lines make it clear that no matter how many times you apply it and no matter what you apply it to, it always returns itself.

To see what's happening in the general case, let's apply the fixed-point finder to a free variable  $F$ :

```

    (λf.(λx.(f (x x)) λx.(f (x x))) F)
==> (λx.(F (x x)) λx.(F (x x)))
==> (F (λx.(F (x x)) λx.(F (x x))))
==> (F (F (λx.(F (x x)) λx.(F (x x)))))
==> (F (F (F (λx.(F (x x)) λx.(F (x x)))))
==> (F (F (F (F (λx.(F (x x)) λx.(F (x x)))))
==> (F (F (F (F (F (λx.(F (x x)) λx.(F (x x)))))
==> (F (F (F (F (F (F (λx.(F (x x)) λx.(F (x x)))))

```

==> (F (F (F (F (F (F (F (F (^x.(F (x x)) ^x.(F (x x))))))))))  
 ...

The computation first builds a self-application, then embeds it inside an application with  $F$  as its operand, then embeds that inside an application with  $F$  as its operand, and so on forever. In cases where the fixed point has a normal form, there will be some point along the way at which it is possible to take advantage of the structure of the operator so as eliminate the endless self-application and terminate the computation; otherwise, it can go on forever. But the application of  $Y$  to a function does always give you a fixed point; it's just that our interpreter can't succeed in reducing that fixed point to a normal form.

When a function has more than one fixed point, it can happen that some of the fixed points have normal forms and others don't. In such cases,  $Y$  may perversely choose one of the ones that doesn't. For instance, it computes the fixed point of  $\lambda x.x$  as  $(\lambda x.(x x) \lambda x.(x x))$ , the result of applying the self-application operator to itself. It is true that this function is a fixed point of identity—since everything is a fixed point of identity!—but it's not the most obvious or convenient one.

## Solving functional equations

With the fixed-point finder, we can automatically construct a solution to any *functional equation* of the form

$$(f \ x) = \dots f \dots x \dots$$

where the right-hand side is any  $\lambda$ -calculus expression, even one containing arbitrary numbers of occurrences of  $f$  and  $x$ . For instance, suppose we want a function that, when applied to any argument  $x$ , gives the result of applying  $x$  to it twice in succession:

$$(f \ x) = (x \ (x \ f))$$

The solution to this equation is found by abstracting  $f$  and  $x$  on the right-hand side and then applying the fixed-point finder:

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

Again, we can confirm that this is a solution by running the expression in which this function is applied to a free variable `X`:

```

((^f.(^x.(f (x x)) ^x.(f (x x))) ^f.^x.(x (x f))) X)
==> ((^x.(^f.^x.(x (x f)) (x x)) ^x.(^f.^x.(x (x f)) (x x))) X)
==> ((^f.^x.(x (x f)) (^x.(^f.^x.(x (x f)) (x x))
      ^x.(^f.^x.(x (x f)) (x x)))) X)
==> (^x.(x (x (^x.(^f.^x.(x (x f)) (x x))
      ^x.(^f.^x.(x (x f)) (x x)))) X)
==> (X (X (^x.(^f.^x.(x (x f)) (x x)) ^x.(^f.^x.(x (x f)) (x x))))
==> (X (X (^f.^x.(x (x f)) (^x.(^f.^x.(x (x f)) (x x))
      ^x.(^f.^x.(x (x f)) (x x))))
==> (X (X ^x.(x (x (^x.(^f.^x.(x (x f)) (x x))
      ^x.(^f.^x.(x (x f)) (x x))))))

```

Compare the fourth and seventh steps to confirm that the desired equation holds.

When we want to create a recursive procedure, then, all we need to do is write the appropriate right-hand side for a functional equation, abstract out the function name and parameter, and apply the fixed-point finder to the result.

## Recursion over natural numbers

With the fixed-point finder for defining recursive procedures, we can start programming in the domain of natural numbers in much the usual way. For example, we can code up a more or less recognizable factorial function:

```

factorial ≡ (Y ^f.^n.(((if (zero? n)) one)
                    ((times n) (f (predecessor n))))))

```

We can use similar techniques to build data structures. For instance, the only operations we need for Scheme-like pairs are the constructor `cons` and the selectors `car` and `cdr`. Here's one possible implementation:

```

cons ≡ ^left.^right.^selector.((selector left) right)
car  ≡ ^pair.(pair true)
cdr  ≡ ^pair.(pair false)

```

We must now choose some value for the empty list that we can distinguish from all pairs. The `null?` predicate can simply invoke its argument, giving it a “selector” value for which any pair will return `false`—say, `^x.^y.false`. Here’s the definition:

$$\text{null?} \equiv \lambda z.(z \ \lambda x.\lambda y.\text{false})$$

Whatever we choose as the representation of the empty list must satisfy this predicate. We can arrange this by having the empty list apply the selector to two dummy arguments, yielding `false`, and then inverting the result:

$$\text{null} \equiv \lambda x.(\text{not } ((x \ \lambda y.y) \ \lambda y.y))$$

Again, this is enough of a base to start coding—using list recursion, for instance:

$$\text{length} \equiv (\text{Y } \lambda \text{len}.\lambda \text{ls}.\left(\left(\text{if } (\text{null? } \text{ls}) \text{ zero}\right)\right. \\ \left.\left(\text{successor } (\text{len } (\text{cdr } \text{ls}))\right)\right))$$

I am indebted to Lindsey Kuper for detecting and correcting an error in an earlier version of this handout.

This work is licensed under the Creative Commons Attribution–ShareAlike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/us/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.