Functional Programming and the Lambda Calculus

Stephen A. Edwards

Columbia University

Fall 2008

Functional vs. Imperative

Imperative programming concerned with “how.”

Functional programming concerned with “what.”

Based on the mathematics of the lambda calculus (Church as opposed to Turing).

“Programming without variables”

It is elegant and a difficult setting in which to create subtle bugs.

It’s a cult: once you catch the functional bug, you never escape.

Referential transparency

The main (good) property of functional programming is *referential transparency*.

Every expression denotes a single value.

The value cannot be changed by evaluating an expression or by sharing it between different parts of the program.

No references to global data; there *is* no global data.

There are no *side-effects*, unlike in *referentially opaque* languages.

The Lambda Calculus

Fancy name for rules about how to represent and evaluate expressions with unnamed functions.

Theoretical underpinning of functional languages. Side-effect free.

Very different from the Turing model of a store with evolving state.

O'Caml:

```
fun x -> 2 * x
```

The Lambda Calculus:

$$\boxed{\lambda x\,.\,*\;2\;x}$$

English:

The function of x that returns the product of two and x

Grammar of Lambda Expressions

$$
\begin{array}{lcl}
\mathit{expr} & \rightarrow & \mathit{constant} \\
 & | & \mathit{variable\text{-}name} \\
 & | & \mathit{expr}\ \mathit{expr} \\
 & | & (\mathit{expr}) \\
 & | & \lambda\ \mathit{variable\text{-}name}\,.\ \mathit{expr}
\end{array}
$$

Constants are numbers; variable names are identifiers and operators.

Somebody asked, “does a language needs to have a large syntax to be powerful?”

Bound and Unbound Variables

In $\lambda x \,.\, * \; 2 \; x$, x is a *bound variable*. Think of it as a formal parameter to a function.

“$* \; 2 \; x$” is the *body*.

The body can be any valid lambda expression, including another unnnamed function.

$$\boxed{\lambda x \,.\, \lambda y \,.\, * \; (+ \; x \; y) \; 2}$$

“The function of x that returns the function of y that returns the product of the sum of x and y and 2.”

Currying

$$\boxed{\lambda x\,.\,\lambda y\,.\,*\;(+\;x\;y)\;2}$$

is equivalent to the O'Caml

```
fun x -> fun y -> (x + y) * 2
```

All lambda calculus functions have a single argument.

As in O'Caml, multiple-argument functions can be built through such "currying."

Currying is named after Haskell Brooks Curry (1900–1982), who contributed to the theory of functional programming. The Haskell functional language is named after him.

Calling Lambda Functions

To invoke a Lambda function, we place it in parentheses before its argument.

Thus, calling $\lambda x \,.\, * \; 2 \; x$ with 4 is written

$\boxed{(\lambda x \,.\, * \; 2 \; x) \; 4}$

This means 8.

Curried functions need more parentheses:

$\boxed{(\lambda x \,.\, (\lambda y \,.\, * \; (+ \; x \; y) \; 2) \; 4) \; 5}$

This binds 4 to y, 5 to x, and means 18.

Evaluating Lambda Expressions

Pure lambda calculus has no built-in functions; we'll be impure.

To evaluate (+ (∗ 5 6) (∗ 8 3)), we can't start with + because it only operates on numbers.

There are two *reducible expressions*: (∗ 5 6) and (∗ 8 3). We can reduce either one first. For example:

```
(+ (∗ 5 6) (∗ 8 3))
(+ 30 (∗ 8 3))
(+ 30 24)
54
```

Looks like deriving a sentence from a grammar.

Evaluating Lambda Expressions

We need a reduction rule to handle λs:

$$(\lambda x\,.\,*\;2\;x)\;4$$
$$(*\;2\;4)$$
$$8$$

This is called β-reduction.

The formal parameter may be used several times:

$$(\lambda x\,.\,+\;x\;x)\;4$$
$$(+\;4\;4)$$
$$8$$

Beta-reduction

May have to be repeated:

$((\lambda x\,.\,(\lambda y\,.-x\ y))\ 5)\ 4$
$(\lambda y\,.-5\ y)\ 4$
$(-\ 5\ 4)$
1

Functions may be arguments:

$(\lambda f\,.\,f\ 3)(\lambda x\,.+x\ 1)$
$(\lambda x\,.+x\ 1)3$
$(+\ 3\ 1)$
4

More Beta-reduction

Repeated names can be tricky:

$$
\begin{array}{l}
(\lambda x\,.\,(\lambda x\,.\,+\,(-\;x\;1))\;x\;3)\;9 \\
(\lambda x\,.\,+\,(-\;x\;1))\;9\;3 \\
+\;(-\;9\;1)\;3 \\
+\;8\;3 \\
11
\end{array}
$$

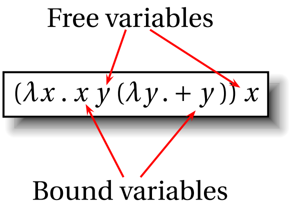
In the first line, the inner x belongs to the inner λ, the outer x belongs to the outer one.

Free and Bound Variables

In an expression, each appearance of a variable is either "free" (unconnected to a λ) or bound (an argument of a λ).

β-reduction of $(\lambda x\,.\,E)\ y$ replaces every x that *occurs free in* E with y.

Free or bound is a function of the position of each variable and its context.



Alpha conversion

One way to confuse yourself less is to do α-conversion.

This is renaming a λ argument and its bound variables.

Formal parameters are only names: they are correct if they are consistent.

$$\boxed{\lambda x\,.\,(\lambda x\,.\,x)\ (+\ 1\ x) \longleftrightarrow_{\alpha} \lambda x\,.\,(\lambda y\,.\,y)\ (+\ 1\ x)}$$

Alpha Conversion

An easier way to attack the earlier example:

$$(\lambda x\,.\,(\lambda x\,.\,+\,(-\;x\;1))\;x\;3)\;9$$
$$(\lambda x\,.\,(\lambda y\,.\,+\,(-\;y\;1))\;x\;3)\;9$$
$$(\lambda y\,.\,+\,(-\;y\;1))\;9\;3$$
$$+\,(-\;9\;1)\;3$$
$$+\;8\;3$$
$$11$$

Reduction Order

The order in which you reduce things can matter.

$$\boxed{(\lambda x\,.\,\lambda y\,.\,y)\;(\,(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\,)}$$

We could choose to reduce one of two things, either

$$\boxed{(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)}$$

or the whole thing

$$\boxed{(\lambda x\,.\,\lambda y\,.\,y)\;(\,(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\,)}$$

Reduction Order

Reducing $(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)$ effectively does nothing because $(\lambda z\,.\,z\;z)$ is the function that calls its first argument on its first argument. The expression reduces to itself:

$$\boxed{(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)}$$

So always reducing it does not terminate.

However, reducing the outermost function does terminate because it ignores its (nasty) argument:

$$\boxed{\begin{array}{l}(\lambda x\,.\,\lambda y\,.\,y)\;(\,(\lambda z\,.\,z\;z)\;(\lambda z\,.\,z\;z)\,)\\ \lambda y\,.\,y\end{array}}$$

Reduction Order

The *redex* is a sub-expression that can be reduced.

The *leftmost* redex is the one whose λ is to the left of all other redexes. You can guess which is the *rightmost*.

The *outermost* redex is not contained in any other.

The *innermost* redex does not contain any other.

For $\boxed{(\lambda x\,.\,\lambda y\,.\,y)\,(\,(\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)\,),}$

$\boxed{(\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)}$ is the leftmost innermost and

$\boxed{(\lambda x\,.\,\lambda y\,.\,y)\,(\,(\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)\,)}$ is the leftmost outermost.

Applicative vs. Normal Order

Applicative order reduction: Always reduce the leftmost innermost redex.

Normative order reduction: Always reduce the leftmost outermost redex.

For

$$\boxed{(\lambda x\,.\,\lambda y\,.\,y)\,(\,(\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)\,)}$$

applicative order reduction never terminated; normative order did.

Applicative vs. Normal Order

Applicative:
reduce leftmost innermost

“evaluate arguments before the function itself”

eager evaluation, call-by-value, usually more efficient

Normative:
reduce leftmost outermost

“evaluate the function before its arguments”

lazy evaluation, call-by-name, more costly to implement, accepts a larger class of programs

Normal Form

A lambda expression that cannot be reduced further is in *normal form.*

Thus,

$$\boxed{\lambda y\,.\,y}$$

is the normal form of

$$\boxed{(\lambda x\,.\,\lambda y\,.\,y)\,(\,(\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)\,)}$$

Normal Form

Not everything has a normal form. E.g.,

$\boxed{(\lambda z \,.\, z\, z)\, (\lambda z \,.\, z\, z)}$

can only be reduced to itself, so it never produces an non-reducible expression.

“Infinite loop.”

The Church-Rosser Theorems

If $E_1 \leftrightarrow E_2$ *(are interconvertable), then there exists an* E *such that* $E_1 \rightarrow E$ *and* $E_2 \rightarrow E$.

“Reduction in any way can eventually produce the same result.”

If $E_1 \rightarrow E_2$*, and* E_2 *is is normal form, then there is a normal-order reduction of* E_1 *to* E_2.

“Normal-order reduction will always produce a normal form, if one exists.”

Church-Rosser

Amazing result:

- Any way you choose to evaluate a lambda expression can produce the same result, i.e., it is confluent.
- “Running” a lambda calculus “program” gives a deterministic result if it terminates.
- Each program means exactly one thing: its normal form.
- Normal order reduction is the most general.

Alonzo Church

1903–1995

Professor at Princeton (1929–1967)
and UCLA (1967–1990)

Invented the Lambda Calculus

Had a few successful graduate students, including

- Stephen Kleene (Regular expressions)
- Michael O. Rabin† (Nondeterministic automata)
- Dana Scott† (Formal programming language semantics)
- Alan Turing (Turing machines)

† Turing award winners

Turing Machines vs. Lambda Calculus

In 1936,

- Alan Turing invented the Turing machine
- Alonzo Church invented the lambda calculus

In 1937, Turing proved that the two models were equivalent, i.e., that they define the same class of computable functions.

Modern processors are just overblown Turing machines.

Functional languages are just the lambda calculus with a more palatable syntax.