Referential Transparency

In functional languages, as in mathematics, there is no notion of a variable being modified.

- No assignment statement. The equation

\[ x = x + 1 \]

has no solution in mathematics.

The lack of assignment ("side-effect") leads to the notion of referential transparency

- "equals can be replaced by equals"

If we say

\[
\text{let } x = f(a) \\
\text{in } \ldots x + x \ldots
\]

then we can be sure that the meaning of \( x + x \) is the same as

\[ f(a) + f(a). \]

Assuming there is no intervening declaration of a new \( x \).

Pragmatically, this is beneficial for understanding and debugging code. We simply need to look at the declaration of a variable to understand its behavior.
This is attractive for philosophical reasons,

• functions are values, thus should be treated like any other value

and for pragmatic reasons.

• gives an additional mechanism of abstraction.

```haskell
fun quadrature(f, x, end, interval) = 
    if x = end then 0
    else ((f(left) + f(x+interval))/2) * interval +
        quadrature(f, x+interval, end, interval)
```

In languages without higher-order functions (or generics), you would have to write a different quadrature routine for each function.
Higher Order Functions

One of the elegant features of the Lambda calculus is that functions (lambda abstractions) are values. This leads to the notion of higher order functions

• functions that manipulate other functions

Functions in functional languages (as in the lambda calculus) are first class objects, they can be

• passed as parameters to other functions,
• returned as results of function calls, and
• stored in aggregates.
Church showed that the lambda calculus is a consistent mathematical system.

- Scott and Strachey (and others) gave a mathematical semantics to the lambda calculus, showing that lambda abstractions do indeed denote values in domains of functions.

- Non-trivial result, since self-application cannot be described representing functions the traditional way as sets.

Modern functional languages are essentially the lambda calculus (in some cases, a typed version) with nicer syntax!

- Thus, the simplicity, consistency, Church-Rosser theorems, etc. all come along for free!
But, hey!, the Y combinator was defined recursively!

• No, Y is just

\[(\lambda h. ((\lambda x. (h \ (x \ x))) \ (\lambda x. (h \ (x \ x)))) \)]

• To see this, for any expression e,

\[Y \ e \ = \ (\lambda h. ((\lambda x. (h \ (x \ x))) \ (\lambda x. (h \ (x \ x)))) \ e \]
\[\Rightarrow ((\lambda x. (e \ (x \ x))) \ (\lambda x. (e \ (x \ x)))) \]
\[\Rightarrow (e \ ((\lambda x. (e \ (x \ x))) \ (\lambda x. (e \ (x \ x)))) \) \Leftrightarrow e \ (Y \ e)\]
Why is this useful? Because now fac can be written as

\[ Y (\lambda \text{fac. } \lambda x. (if (= x 0) 1 (* x (\text{fac} (- x 1)))) ) \]

• To see that this has the desired behavior, let

\[ F = \lambda \text{fac. } \lambda x. (if (= x 0) 1 (* x (\text{fac} (- x 1)))) \]

• Notice that

\[
(Y F) 3 \Rightarrow^* Y (\lambda \text{fac. } \lambda x. (if (= x 0) 1 (* x (\text{fac} (- x 1)))) ) 3 \\
\Rightarrow^* (\lambda \text{fac. } \lambda x. (if (= x 0) 1 (* x (\text{fac} (- x 1)))) ) (Y F) 3 \\
\Rightarrow^* (\lambda x. (if (= x 0) 1 (* x ((Y F) (- x 1)))) ) 3 \\
\Rightarrow^* (* 3 ((Y F) 2)) \\
\Rightarrow^* \ldots \\
\]

• In general, if you want to write a recursive function of the form

\[ f = \lambda x. e \]

where \( f \) occurs free in \( e \), write it in the lambda calculus as

\[ Y (\lambda f. \lambda x. \text{body}) \]
Recursion in the lambda calculus

It appears impossible to define recursion functions, since the functions aren’t named.

• Can’t write

\[ \text{fac} = \lambda x. (\text{if } (\text{=} x 0) 1 (* x (\text{fac} (- x 1)))) \]

• So what can we do?

First, some terminology:

• The fixpoint of a function \( f \) is the value \( e \) such that

\[ f e = e \]

• For recursion in the lambda calculus, one can use the fixpoint combinator \( Y \), defined as

\[ Y f = f (Y f) \]

• For any function \( f \), \((Y f)\) computes \( f \)’s fixpoint.
But, here is the first data point:

**Church Rosser Theorem II**

If $e_1 \Rightarrow^* e_2$ and $e_2$ is in normal form, then there exists a normal-order reduction from $e_1$ to $e_2$.

This says that if any reduction sequence terminates, then normal order reduction will.

- normal order reduction is the most likely to terminate!
Common Evaluation Orders

• **Applicative order evaluation**: reduce the leftmost innermost redex first.
  
  • intuitively, evaluate the arguments first
  
  • used by most programming languages, including “strict” functional languages

• **Normal Order evaluation**: reduce the leftmost outermost redex first.

  • intuitively, evaluate the body of the function first and the arguments when necessary.

  • used by “non-strict” functional languages

Which is better? Well... stay tuned!
Can two terminating reductions give different answers?

**Church-Rosser Theorem I**
If $e_1 \Leftrightarrow e_2$ then there exists an $e_3$ such that $e_1 \Rightarrow e_3$ and $e_2 \Rightarrow e_3$

**Corollary**
No lambda expression can be converted to two distinct normal forms.

- So, all terminating reduction sequences give the same answer
**Does the order in which redexes are chosen matter?**

Sure!

Consider

\[
(\lambda y. \, 3) \, ((\lambda x. \, (x \, x)) \, (\lambda x. \, (x \, x)))
\]

Reducing the outer redex first gives us

\[3\]

Reducing the inner redex first gives us

\[
(\lambda y. \, 3) \, ((\lambda x. \, (x \, x)) \, (\lambda x. \, (x \, x))) \Rightarrow_\beta (\lambda y. \, 3) \, ((\lambda x. \, (x \, x)) \, (\lambda x. \, (x \, x))) \\
\Rightarrow_\beta \ldots
\]

The reduction of the argument never terminates, but its value isn’t needed.

- In this case, one reduction order terminated and the other didn’t.
**Reduction Order**

- An expression may contain several reducible expressions, called *redexes*. For example,

\[
((\lambda x. (+ x x)) (+ 3 2))
\]

- In general, there may be many redexes to choose from.
We model computation as the process of taking an expression and reducing it as far as possible, to a *normal form*

- An expression that cannot be reduced further

Not all expressions can be reduced to a normal form.

\[(\lambda x. (x \ x)) \ (\lambda x. (x \ x))\]

has no normal form:

\[(\lambda x. (x \ x)) \ (\lambda x. (x \ x)) \ \Rightarrow_B (\lambda x. (x \ x)) \ (\lambda x. (x \ x))\]

\[\Rightarrow_B ...\]
We write

\[ e_1 \leftrightarrow^* e_2 \]

if \( e_1 \) and \( e_2 \) can be converted to one another by zero or more applications of the conversion rules (i.e. the reflexive transitive closure).

Although conversion is both ways (\( \leftrightarrow \) above) we are mainly interested in \( \beta-, \delta-, \) and \( \eta\)-reduction, in which the conversion is only \( \Rightarrow \).

- \( \beta\)-Reduction

\[
(\lambda x. e) \ M \Rightarrow_\beta e[M/x]
\]

- \( \eta\)-Reduction

\[
\lambda x. (e \ x) \Rightarrow_\eta e \quad \text{where } x \not\in \text{fv}(e)
\]

Similarly

\[ e_1 \Rightarrow^* e_2 \]

denotes the reduction of \( e_1 \) to \( e_2 \) by zero or more applications of the reduction rules.
Conversions between Lambda Expressions

• \( \alpha \)-conversion (renaming of bound variables)

\[
\lambda x. e \Leftrightarrow_{\alpha} \lambda y. e[y/x] \quad \text{where } y \notin \text{fv}(e)
\]

• \( \beta \)-conversion (application)

\[
(\lambda x. e) M \Leftrightarrow_{\beta} e[M/x]
\]

• \( \eta \)-conversion

\[
\lambda x. (e \, x) \Leftrightarrow_{\eta} e \quad \text{where } x \notin \text{fv}(e)
\]

For the pre-defined operators, there are conversions, called \( \delta \)-conversions, between an application of the operator and the result. For example,

\[
(\, + \, 1 \, 2) \Leftrightarrow_{\delta} 3
\]

\[
(\text{if true } e1 \, e2) \Leftrightarrow_{\delta} e1
\]

\[
(\text{if false } e1 \, e2) \Leftrightarrow_{\delta} e2
\]
Computation is modeled by conversions using *textual substitution* on lambda expressions.

**Free variables and substitution**

- Intuitively, the *free variables* in an expression are the “non-local” variables.

- The free variables of an expression are defined as follows:

  \[
  \begin{align*}
  \text{fv}(x) & = \{x\} \\
  \text{fv}(e_1 \ e_2) & = \text{fv}(e_1) \cup \text{fv}(e_2) \\
  \text{fv}(\lambda x.e) & = \text{fv}(e) - \{x\}
  \end{align*}
  \]

The notation \( e[M/x] \) denotes the result replacing all free occurrences of the variable \( x \) with the expression \( M \) in \( e \).

- One has to be careful, though, to avoid name conflicts.

  \[
  \begin{align*}
  x[M/x] & = M \\
  y[M/x] & = y \quad \text{where } y \text{ is a variable, } y \neq x \\
  (e_1 \ e_2) [M/x] & = (e_1[M/x]) \ (e_2[M/x]) \\
  (\lambda x.e) [M/x] & = \lambda x.e \\
  (\lambda y.e) [M/x] & = \lambda y.(e[M/x]) \text{ where } y \neq x, y \notin \text{fv}(M) \\
  (\lambda y.e) [M/x] & = (\lambda z.e[z/y]) [M/x] \text{ otherwise, where } z \neq y, z \neq x, z \notin (\text{fv}(e) \cup \text{fv}(M))
  \end{align*}
  \]
The Lambda Calculus

We’ll only be talking about the *untyped* lambda calculus augmented with constants - there are many others versions.

- Just a set of rules describing what constitutes a legal expression and conversions between expressions.

**Lambda Expressions**

\[
e ::= \quad c \quad \text{constant (including operators +, -, if, etc.)} \\
\quad \mid x \quad \text{variable} \\
\quad \mid e_1 e_2 \quad \text{application} \\
\quad \mid \lambda x.e \quad \text{lambda abstraction (models functions)}
\]

Application is left associative, so

\[(e_1 e_2 e_3)\]

is equivalent to

\[((e_1 e_2) e_3)\]

Examples:

\[(\lambda x. + x x)\]
\[(\lambda x.x x) (\lambda y. y y)\]
\[+(((\lambda x. + x 3) 4) 5)\]