The Lambda Calculus

The theoretical foundations of funuctional programming include:

- The lambda calculus,
- Domain theory,
- Category theory, and
- Type theory.

We will briefly consider aspects of these theoretical foundations as well as more practical aspects of functional programming. Our aim is to study the theory to the extent that it helps us with the practical aspects of functional programming.

The lambda calculus is a notation and associated Turing complete model of computation.

Syntax

The abstract syntax of the lambda calculus is:

$$\begin{array}{lcll} <exp> & ::= & <var> & \\ & | & <exp><exp> & \text{Application} \\ & | & \lambda <var> . <exp> & \text{Abstraction} \\ <var> & ::= & \text{Some countably infinite set} & \end{array}$$

In the concrete syntax we allow parentheses. In addition, to reduce the number of parentheses, we use the following conventions:

- Application is left associative, and
- Abstractions extend as far to the right as possible.

Sometimes the lambda calculus it defined to include constants as well as variables, with the addition of the productions:

$$\begin{array}{lcl} <exp> & ::= & <const> \ldots \\ <const> & ::= & \ldots \end{array}$$

For example, the constants might include integers and basic integer operations.

We will see, in section 3.7 of Bird, that natural numbers can be represented by functions, so the addition of constants is not necessary. With just the basic lambda calculus we can express any Turing computable function from natural numbers to natural numbers.

The lambda notation lets us define functions without giving them names.

For example, f where $f(x) = g(h(x))$ can be written as λ`x.g (h x)`.

We can still name things when we want to.

For example, let $x = y$ in $f(x)$ can be written as `(`λ`x.f x) y`.

Later on, we will informally mix our notation at times.

Examples

```
(λ a.λ b.a) c
(λ a.λ b.a) c d
(λ a.λ b.b) c
(λ a.λ b.b) c d
(λ a.λ b.a) b
(λ a.λ b.a) b d
(λ x.x x)(λ x.x x)
(λ x.x x x)(λ x.x x x)
λ h.(λ x.h (x x))(λ x.h (x x))
(λ h.(λ x.h (x x))(λ x.h (x x))) f
```


Reduction

The process of "evaluating" a lambda expression is called *reduction*.

In the following, we will use v, w and x to denote arbitrary variables and E, E', E_1, and E_2 to denote arbitrary expressions.

Free Variables

Informally, a variable is *free* if it is not within the scope of a variable appearing after a λ (that is, it is "undefined").

Formally, we define $FV(E)$, the set of free variables in a lambda expression E, by

$$\begin{aligned} FV(v) &= \{v\} \\ FV(E_1 E_2) &= FV(E_1) \cup FV(E_2) \\ FV(\lambda v . E) &= FV(E) - \{v\} \end{aligned}$$

Substitution

E	$E[E'/v]$	Restriction	Comment
v	E'		
w	w	$w \neq v$	
$E_1 E_2$	$(E_1[E'/v])\,(E_2[E'/v])$		
$\lambda v\,.\,E_1$	$\lambda v\,.\,E_1$		
$\lambda w\,.\,E_1$	$\lambda w\,.(E_1[E'/v])$	$w \neq v \wedge$	
		$(v \notin FV(E_1) \vee$	No Substitution
		$w \notin FV(E'))$	No Name Conflict
$\lambda w\,.\,E_1$	$\lambda x\,.((E_1[x/w])\,[E'/v])$	$w \neq v \wedge$	
		$v \in FV(E_1) \wedge$	Substitution
		$w \in FV(E') \wedge$	Name Conflict
		$x \notin FV(E_1 E')$	x is new

Reduction Rules

α-conversion

$$\lambda v\,.\,E \underset{\alpha}{\longleftrightarrow} \lambda x\,.\,E[x/v], \qquad \text{if } x \notin FV(E)$$

β-reduction

$$(\lambda v\,.\,E_1)E_2 \underset{\beta}{\longrightarrow} E_1[E_2/v]$$

η-reduction

$$\lambda v\,.(Ev) \underset{\eta}{\longrightarrow} E, \qquad \text{if } v \notin FV(E)$$

Reduction of subexpressions

$$\frac{E_1 \xrightarrow[\beta]{} E_2}{EE_1 \xrightarrow[\beta]{} EE_2}$$

$$\frac{E_1 \xrightarrow[\beta]{} E_2}{E_1E \xrightarrow[\beta]{} E_2E}$$

$$\frac{E_1 \xrightarrow[\beta]{} E_2}{\lambda v \mathbin{.} E_1 \xrightarrow[\beta]{} \lambda v \mathbin{.} E_2}$$

Similar rules apply to α-conversion and η-reduction.

An expression, E, is called a redex if it is of the form $(\lambda v \mathbin{.} E_1)E_2$ or $\lambda v \mathbin{.} (E_3 v)$, with $v \notin FV(E_3)$.

Hence, it is possible to apply at least one of the reduction rules to an expression iff the expression contains a redex.

An expression, E, is said to be in normal form if it does not contain a redux.

We note that if $\lambda v.E_1 \underset{\alpha}{\longleftrightarrow} \lambda x.E_2$ then, by definition of α-conversion, x cannot be free in E_1 and v cannot be free in E_2. It follows that $(E_1[x/v])[v/x]$ is exactly the same as $E_1$and hence $\lambda v.E_1 \underset{\alpha}{\longleftrightarrow} \lambda x.E_2$ iff $\lambda x.E_2 \underset{\alpha}{\longleftrightarrow} \lambda v.E_1$. That is, $\underset{\alpha}{\longleftrightarrow}$ is symmetric, so the reflexive, transitive closure of $\underset{\alpha}{\longleftrightarrow}$, which we will call $\Leftrightarrow$, is an equivalence relation.

Clearly, $E_1 \Leftrightarrow E_2$ means that the two expressions are identical except for the renaming of bound variables.

We will define

$$\longrightarrow \; = \; \underset{\alpha}{\longleftrightarrow} \cup \underset{\beta}{\longrightarrow} \cup \underset{\eta}{\longrightarrow}$$

and let $\xrightarrow{*}$ be the reflexive, transitive closure of $\longrightarrow$, and $\xleftrightarrow{*}$ be the symmetric, reflexive, transitive closure of $\longrightarrow$.

The relation $\xleftrightarrow{*}$ is an equivalence relation, but with larger equivalence classes than $\Leftrightarrow$. That is, whenever $E_1 \Leftrightarrow E_2$ then $E_1 \xleftrightarrow{*} E_2$, but not conversely.

If $n \geq 0$ and $E_0 \xrightarrow{*} E_n$, then a sequence of reduction steps $E_0 \longrightarrow E_1 \longrightarrow E_2 \longrightarrow \cdots \longrightarrow E_n$ is called a reduction . E_n may or may not be in normal form.

In general, an expression may contain a number of redexes, so there may be a number of different ways to go about reducing an expression.

If at each step during a reduction the leftmost outermost redex is chosen for reduction, then the reduction is called a normal order reduction .

If at each step during a reduction the leftmost innermost redex is chosen for reduction, then the reduction is called a applicative order reduction .

Theorem 1 (Church-Rosser Theorem I) If $E \xrightarrow{*} E_1$ and $E \xrightarrow{*} E_2$ then there exists an E' such that $E_1 \xrightarrow{*} E'$ and $E_2 \xrightarrow{*} E'$.

Proof This is an Area II course, so you'll just have to take my word for this.

Corollary 1 If $E \xrightarrow{*} E_1$ and $E \xrightarrow{*} E_2$ and E_1 and E_2 are both in normal form, then $E_1 \Leftrightarrow E_2$.

Theorem 2 (Church-Rosser Theorem II) If $E \xrightarrow{*} E_1$ and E_1 is in normal form, then there exists a normal order reduction of E to E_1,

A lazy functional programming language uses normal order reduction, but with an optimization that avoids reevaluation of subexpressions that are duplicated during the reduction.

Normal order reduction with this optimization is called lazy evaluation .

Normal order reduction without this optimization yields call-by-name parameter passing, and applicative order reduction is somewhat similar to call-by-value parameter passing.