Contents

Abstract

Acknowledgements

Introduction

Chapter 1

Lambda Calculus

The Lambda Calculus is an example of a formal system which consists a language of lambda terms and some auxiliary notions such as *free variables* and *subterms*, and the transformation theory. The core of the theory is the notion of substitution - the driving force behind function application.

1.1 λ-terms

The expressions in lambda calculus can be formalized into following:

Definition 1.1 (λ-TERMS) λ-terms can be defined by the following rules:

- a variable, x, itself is a lambda term
- if M is a lambda term, and x is a variable, then $(\lambda x.M)$ is a lambda term
- if M and N are both lambda terms, then *(MN)* is a lambda term

A lambda term is valid if and only if it can be obtained by repeated application of these rules. However, some parentheses can be omitted in certain forms. For example the leftmost, outermost brackets are usually omitted.

A lambda abstraction $\lambda x.M$ takes a single input x and returns a term M. Thus, it defines an **anonymous** function. For example $\lambda x.x^2$ is a lambda abstraction for the function $f(x) = x^2$ using the term x^2 for M, then we can say that $f = \lambda x.x^2$. The function is anonymous since we write $(x^2)2$ rather than $f3$.

An application (MN) applies the input term N to the function M. For example $(\lambda x.x^2)2$ is an application that applies 2 to the function $f(x) = x^2$, it return 2^2 which equals 4.

As mentioned above, leftmost, outermost brackets are usually omitted. Therefore, $M_1M_2(M_3M_4)$ stands for $((M_1 \cdot M_2)(M_3 \cdot M_4))$. Similarly, λ can be omitted in repeated abstractions, for example $\lambda x_1x_2.M$ stands for $\lambda x_1.(\lambda x_2.M)$.

Definition 1.2 (free and bound variables) The set of free and bound variables are defined inductively by the following function:

$$\begin{array}{lcllcl} FV(x) & = & \{x\} & BV(x) & = & \emptyset \\ FV(\lambda x.M) & = & FV(M)\backslash\{x\} & BV(\lambda x.M) & = & BV(M) \cup \{x\} \\ FV(MN) & = & FV(M) \cup FV(N) & BV(MN) & = & BV(M) \cup BV(N) \end{array}$$

For example, the lambda term $\lambda x.x$ has no free variables but a bound variable x. The function $\lambda x.x + y$ only has a single free variable y and a bound variable x. Notice that, the sets of bound and free variables are not necessarily disjoint; x occurs both bound and free in:

$$x(\lambda xy.x)$$

$$
\begin{array}{lrll}
(1) & x[x := N] & = N & \\
(2) & y[x := N] & = y, & if\ y \neq x \\
(3) & (\lambda y.M)[x := N] & = \lambda y.M, & if\ y = x \\
(4) & (\lambda y.M)[x := N] & = \lambda y.(M[x := N]), & if\ y \notin FV(N)\ \&\ y \neq x \\
(5) & (\lambda y.M)[x := N] & = \lambda z.(M[y := z])[x := N], & if\ y \in FV(N)\ \&\ y \neq x, z\ new \\
(6) & (M_1 M_2)[x := N] & = (M_1[x := N])(M_2[x := N]) &
\end{array}
$$

Definition 1.3 (α-equivalent) *M is α-equivalent to N, written $M \equiv_\alpha N$, if N results from M by a series of changes of bound variable.*

The notion of alpha-equivalent(or congruence) is a basic form of equivalence defined in lambda calculus. It captures the property that the particular choice of a bound variable in a lambda abstraction does not matter. For instance, $\lambda x.x$ and $\lambda y.y$ are α-congruent lambda terms which represent the same function. Notice that, the term-variable x and y are not α-equivalent terms since they are not bound in a lambda abstraction.

With the property of alpha equivalence, we can define the *α-conversion*:

$$\lambda x.M =_\alpha \lambda z.(M[z := x])$$

In some sense, the α-conversion is defined by the spirit of the (5) substitution rule. If we have a function $\lambda x.M$, then we can apply a substitution to this function: $(\lambda x.M)[z := x]$. According to the fifth substitution rule, it is unfolded to $(\lambda z.M[z := x])$. Using the α-conversion, we can always rename the bound variables of a term.

Definition 1.4 (Variable Convention[5]) *If M_1,...,M_n occur in a certain context then in these terms all bound variables are chosen to be different from free variables.*

The alpha-conversion is built based on the variable convention. Alpha convertion is used to allow bound variable names to be changed. For example, alpha-conversion of $\lambda x.x$ might get $\lambda y.y$. Terms that differ only by alpha-conversion are called alpha-equivalence(or alpha-congruence) as mentioned in the definition 1.3. By the assumption that free and bound variables are always different(variable convention), and that alpha-conversion will take place whenever a variable is both free and bound. The definition of term substitution becomes:

$$
\begin{array}{rll}
x[x := N] & = N & \\
y[x := N] & = y, & if\ y \neq x \\
(\lambda y.M)[x := N] & = \lambda y.(M[x := N]) & \\
(M_1 M_2)[x := N] & = (M_1[x := N])(M_2[x := N]) &
\end{array}
$$

By this definition of term substitution, the *variable capture* is avoided. Since that y is bound in $\lambda y.M$ of the context:

$$\lambda y.M[x := N]$$

y can only be bound in N according to variable convention rule, otherwise, if y is also free in N, it would be renamed by another variable.

Here is an example of its use:

$$
\begin{array}{ll}
 & (\lambda xyz.xzy)(\lambda xz.x) \\
= & \lambda yz.(\lambda xz.x)zy \\
= & \lambda yz.(\lambda xw.x)zy \ \ by\ the\ variable\ convention \\
= & \lambda yz.(\lambda w.z)y \\
= & \lambda yz.z
\end{array}
$$

1.2 β-reduction

There are various notions of reduction for λ-terms, but the principle one is β-reduction. β-reduction is the one-step reduction relation, written as $' \rightarrow'_\beta$.

Definition 1.5 *(β-reduction) For λ-terms M and N, we say that M β-reduces in one step to N, written as $M \rightarrow_\beta N$:*

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$

$$\frac{M \rightarrow_\beta N}{MZ \rightarrow_\beta NZ}$$

$$\frac{M \rightarrow_\beta N}{ZM \rightarrow_\beta ZM}$$

$$\frac{M \rightarrow_\beta N}{\lambda x.M \rightarrow_\beta \lambda x.N}$$

The reduction relation, written $' \twoheadrightarrow'_\beta$, is the reflexive, transitive closure of the one-step reduction relation. The one-step reduction relation allows a single step of reduction, while the reduction relation allows many steps. The reflexive transitive closure is defined as follows:

$$\frac{M \rightarrow_\beta N}{M \twoheadrightarrow_\beta N}$$

$$M \twoheadrightarrow_\beta M$$

$$\frac{M \twoheadrightarrow_\beta N \quad N \twoheadrightarrow_\beta Z}{M \twoheadrightarrow_\beta Z}$$

For the notion $'M \rightarrow_\beta N'$, it is read as 'M β-reduces to N'. The first rule defines that the reduction relation is reflexive to one-step reduction relation, while the third rule indicates the transitivity of reduction relation.
Finally, the β-equality, written as '$=_\beta$'. For λ-terms A and B, we say that $A =_\beta B$ if either $A \equiv B$ or there exists a sequence of reduction starting with A, ending with B. It is the equivalence relation generated by $\twoheadrightarrow_\beta$:

$$\frac{M \twoheadrightarrow_\beta N}{M =_\beta N}$$

$$\frac{M =_\beta N}{N =_\beta M}$$

$$\frac{M =_\beta N \quad N =_\beta Z}{M =_\beta Z}$$

For the notion $'M =_\beta N'$, we say 'M β-equivalent to N'.
Following is a simple example which applies β-reduction:

$$\begin{aligned} & (\lambda x.x^2)3 \\ = \; & (x^2)[x := 3] \\ = \; & 3^2 \\ = \; & 9 \end{aligned}$$

Definition 1.6 *(β-redex) A β-redex of a λ-term M is a subterm of M of the form $(\lambda x.P)Q$. A term M is called in β-normal form if it has no β-redex.*

A β-redex is essentially a candidate for an application of β-reduction. A term M has a β-normal form if there exists a term N such that N is in β-normal form and $M \twoheadrightarrow_\beta N$.

1.3 Head Normal Form

A λ-term in head normal form is generally in the form:

$$\lambda x_1 \ldots x_n.xM_1 \ldots M_m \qquad n, m \geqslant 0$$

In this case x is called the head variable. If $M \equiv \lambda x_1 \ldots x_n.(\lambda x.M_0)M_1 \ldots M_m$ where $n \geqslant 0$, $m \geqslant 1$ then the subterm $(\lambda x.M_0)M_1$ is called the head redex of M. Following are some examples of λ-terms in head normal form:

$$(1)\ xM \quad (2)\ \lambda x.x \quad (3)\ \lambda xy.x((\lambda z.z)y)$$

Normal order: Notice that, an expression in head normal form may contain redexes in argument positions whereas a normal form may not.

1.4 Reduction strategies

Recall that a term is said to be in β-normal form if it has no β-redexes, that is, subterms of the shape (λx.M)N. A term has a β-normal form if it can be reduced to a term in β in β-normal form. It is clear that if a term has a β-normal form, then we can get to the β-normal form by exhaustively reducing all β-redexes of the term, then reducing all β-redexes of all resulting terms, and so forth until we get the β-normal form. A **β-reduction strategy** is a function that selects, whenever a term has multiple β-redexes, which one should be reduced first. If a term is in β-normal form, then no β-reduction is to be done. Notice that, a β-reduction strategy may or may not have the property that adhering to the strategy will ensure that we(eventually) reach a β-normal form, if one exists.

In general, there are several different β-reductions possible for a given term. In this project, there are 5 different reduction strategies studied and implemented: normal order, call-by-name, call-by-value, head reduction, and applicative order. Figure X generally summarizes and compares the property of different strategies:

Reduction Strategy	Reduction Order	Reach β-normal form	Form Reached
Normal Order	leftmost outermost	Yes	normal form
Call-by-name	leftmost outermost	No	weak head normal form
Call-by-value	leftmost innermost	No	weak normal form
Head Reduction	inside lambda abstractions	No	head normal form
Applicative Order	leftmost innermost	Yes	normal form

Table 1.1: Reduction Strategies

Normal order: Lambda terms are modeled as Haskell constructed data, representing variable names by character:

```
type Name = Char

data Term = Var Name | Abs Name Term | App Term Term
            deriving (Show, Eq)
```

Operational Semantics

A sequence of computational steps of a valid program is the operational semantics for the programming language. The final step of a terminating sequence returns the resulting value of the

program. Operational semantics can be classified into two types: **structural operational semantics(SOS or small-step semantics)** and **natural semantics(NS or big-step semantics)**. The structural operational semantics describes each single step of a computation in a program, while the natural semantics describes the overall resulting value generated by the program. Following, we use both SOS and NS to define the behavior of each reduction strategy. The small-step semantics gives reduction rules of the strategy, when the big-step semantics describes how the final term is reached.

Recursion

Although a reduction strategy may or may not have the property that ensures it reaches a β-normal form, it would recursively contract the selected redex until no redex(decided by specific reduction strategy) can be contracted. All the following Haskell functions defined conrresponding to the reduction strategy model the small-step semantics. Therefore, in order to recursively reduce a λ-term, an auxiliary function `recur_eval :: Term -> IO()` is defined:

```
recur_eval :: Term -> IO()
recur_eval t | evalcbn t == t = putStrLn("")
             | otherwise = do
                         putStrLn("==> "++ tostring (evalcbn t))
                         recur_eval (evalcbn t)
```

The Haskell function above defines the recursive reduction by Call-by-Name strategy. It uses the small-step function defined in Section 1.4.2. The recursion will terminate if the input λ-term cannot be reduced anymore, otherwise, it would print-out the term after a single step reduction and continue reducing the resulting term.

1.4.1 Normal Order Reduction

The normal order reduction $e \xrightarrow{no} e'$ continually applies te rule for β-reduction on the redex in leftmost outermost position until no more β-reduction can be performed. At that point, the resulting term is in normal form. When reducing an application $(e_1 e_2)$, the function term e_1 must be reduced using call-by-name. Since the strategy is outermost, when e_1 is reduced to an abstraction $(\lambda x.e)$, then the redex $((\lambda x.e)e_2)$ must be reduced before redexes in e.

$$\llbracket x \rrbracket^{ls}_{no} = x$$

$$\frac{\llbracket e \rrbracket^{ls}_{no} = e'}{\llbracket (\lambda x.e) \rrbracket^{ls}_{no} = (\lambda x.e')}$$

$$\frac{\llbracket e_1 \rrbracket^{ls}_{cbn} = (\lambda x.e) \qquad \llbracket e[e_2/x] \rrbracket^{ls}_{no} = e'}{\llbracket (e_1 e_2) \rrbracket^{ls}_{no} = e'}$$

$$\frac{\llbracket e_1 \rrbracket^{ls}_{cbn} = e_1' \neq (\lambda x.e) \qquad \llbracket e_1' \rrbracket^{ls}_{no} = e_1'' \qquad \llbracket e_2 \rrbracket^{ls}_{no} = e_2'}{\llbracket (e_1 e_2) \rrbracket^{ls}_{no} = (e_1'' e_2')}$$

Normal Order: Big-step operational semantics

$$\frac{}{(\lambda x.M)N \to_\beta M[N/x]} \quad \frac{M \to_\beta N}{MP \to_\beta NP} \quad \frac{M \to_\beta N}{PM \to_\beta PN}(P \textit{ contains no redex}) \quad \frac{M \to_\beta N}{(\lambda x.M) \to_\beta (\lambda x.N)}$$

Normal Order: Small-step operational semantics

Normal order reduction is *normalizing*, since it reduces the λ-term until there is no redex, redex in abstractions is also contracted. So, the normal order reduction will terminate with the normal form as result.

The corresponding Haskell function `evalnormal :: Term -> Term` below implements the normal order reduction. Notice that, it uses the function evalcbn defined in 1.4.2:

```
evalnormal :: Term -> Term
evalnormal (Var x) = Var x
evalnormal (Abs x t) = (Abs x (evalnormal t))
evalnormal (App (Abs x t) v) = subs t (Var x, v)
evalnormal (App x t) | x == evalcbn x = (App x (evalnormal t))
                     | otherwise = (App (evalcbn x) t)
```

The first function clause above handles variables x. It is not a β-reduction step, since a variable is not a redex. It returns itself which indicates no reduction can be done. It also conforms the pattern matching mechanism in Haskell. Similarly, the second function clause handles abstractions $(\lambda x.e)$ and implements the fourth SOS rule. The third function handles applications $(e_1 e_2)$ when e_1 is an abstraction, then the substitution will take place. It applies an argument to the function and returns the resulting value. Finally, the fourth function handles applications $(e_1 e_2)$ when e_1 is not an abstraction. Since the normal reduction is leftmost outermost, it reduces the function e_1 first. If e_1 cannot be reduced(returns itself), then it goes to the argument e_2.

***Example* 1.4.1** Folllowing is the running example of normal order reduction in Haskell. We use "/" to represent the λ symbol:

```
    (/a.a)(/b.b)((/x.x)(/y.(/z.z)w))
==> (/b.b)((/x.x)(/y.(/z.z)w))
==> (/x.x)(/y.(/z.z)w)
==> /y.(/z.z)w
==> /y.w
Reduced to:/y.w
```

As we can see, it uses leftmost outermost strategy. The leftmost outermost redex $(\lambda a.a)(\lambda b.b)$ is first reduced. Then the whole reduced term is a redex, the argument $((\lambda x.x)(\lambda y.(\lambda z.z)w))$ substitutes the bound variables in the abstraction $(\lambda b.b)$. After that, the same substitution take place. Finally, it reaches into the redex $(\lambda z.z)w$ in the abstraction $\lambda y.(\lambda z.z)w$ and generates the resulting term $\lambda y.w$

1.4.2 Call-by-Name Reduction

The Call-by-Name reduction $e \xrightarrow{cbn} e'$ reduces the leftmost outermost redex not insde a lambda abstraction first. That is, the arguments to a function are not evaluated before the function is called. The redex in e of the asbtraction $(\lambda x.e)$ will never be reduced.

$$[\![x]\!]^{ls}_{cbn} = x$$

$$[\![(\lambda x.e)]\!]^{ls}_{cbn} = (\lambda x.e)$$

$$\frac{[\![e_1]\!]^{ls}_{cbn} = (\lambda x.e) \qquad [\![e[e_2/x]]\!]^{ls}_{cbn} = e'}{[\![(e_1 e_2)]\!]^{ls}_{cbn} = e'}$$

$$\frac{[\![e_1]\!]^{ls}_{cbn} = e_1' \neq (\lambda x.e)}{[\![(e_1 e_2)]\!]^{ls}_{cbn} = (e_1' e_2)}$$

Call-by-Name: Big-step operational semantics

$$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]} \qquad \frac{M \rightarrow_\beta N}{MP \rightarrow_\beta NP}$$

Call-by-Name: Small-step operational semantics

The Call-by-Name reduction generates term in weak head normal form. A lambda expression is in weak head normal form if it is a head normal form or any lambda abstraction.

The corresponding Haskell function `evalcbn :: Term -> Term` below implements the Call-by-Name reduction:

```
evalcbn :: Term -> Term
evalcbn (Var x) = Var x
evalcbn (Abs x t) = (Abs x t)
evalcbn (App (Abs x t) y) = subs t (Var x, y)
evalcbn (App x y) | x == evalcbn x = (App x y)
                  | otherwise = (App (evalcbn x) y)
```

The first two function clauses handle the variables and abstractions, it returns itself since they cannot be reduced under Call-by-Name reduction. The third function implements the first SOS rule which performs substitution. The last function implements the second SOS rule: if the function e_1 of the application (e_1e_2) cannot be reduced, then the argument e_2 will never be reduced.

***Example* 1.4.2** The same example in 1.4.1 run in Haskell by the Call-by-Name strategy is as follows:

```
    (/a.a)(/b.b)((/x.x)(/y.(/z.z)w))
==> (/b.b)((/x.x)(/y.(/z.z)w))
==> (/x.x)(/y.(/z.z)w)
==> /y.(/z.z)w
Reduced to:/y.(/z.z)w
```

It is easy to see that the first three reduction steps are the same as normal order reduction. In this example, it stops at $\lambda y.(\lambda z.z)w$. Since the only difference between normal order and call-by-name is the redex inside abstractions will never be reduced in call-by-name.

1.4.3 Call-by-Value Reduction

Call-by-Value reduction is the most common reduction strategy. In Call-by-Value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function. It first reduces the leftmost innermost redex not inside an abstraction. It never reduces a redex when the argument is not a *value*. It differs from Call-by-Name only by reducing the argument e_2 of the application (e_1e_2) before the substitution take place.

$$\llbracket x \rrbracket^{ls}_{cbv} = x$$

$$\llbracket (\lambda x.e) \rrbracket^{ls}_{cbn} = (\lambda x.e)$$

$$\frac{\llbracket e_1 \rrbracket^{ls}_{cbv} = (\lambda x.e) \qquad \llbracket e_2 \rrbracket^{ls}_{cbv} = e_2' \qquad \llbracket e[e_2'/x] \rrbracket^{ls}_{cbv} = e'}{\llbracket (e_1e_2) \rrbracket^{ls}_{cbv} = e'}$$

$$\frac{\llbracket e_1 \rrbracket^{ls}_{cbv} = e_1' \neq (\lambda x.e) \qquad \llbracket e_2 \rrbracket^{ls}_{cbv} = e_2'}{\llbracket (e_1e_2) \rrbracket^{ls}_{cbv} = (e_1'e_2')}$$

Call-by-Value: Big-step operational semantics

$$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]} \quad \frac{M \rightarrow_\beta N}{MP \rightarrow_\beta NP}(M \textit{ not an abstraction}) \quad \frac{M \rightarrow_\beta N}{VM \rightarrow_\beta VN}(M \textit{ not an abstraction})$$

Call-by-Value: Small-step operational semantics

Call-by-Value generates terms in weak head normal form only. The implementation of the rules by an Haskell function `evalcbv :: Term -> Term` is as follows:

```
evalcbv :: Term -> Term
evalcbv (Var x) = Var x
evalcbv (Abs x t) = (Abs x t)
evalcbv (App (Abs x t) y) = subs t (Var x, y)
evalcbv (App x y) | x == evalcbv x = (App x (evalcbv y))
                  | otherwise = (App (evalcbv x) y)
```

The functions are similar to Call-by-Name, the only difference is the argument is reduced before the substitution take place as indicated in the function clause 3.

***Example* 1.4.3** The sample example run in Haskell by the Call-by-Value strategy is as follows:

```
    (/a.a)(/b.b)((/x.x)(/y.(/z.z)w))
==> (/b.b)((/x.x)(/y.(/z.z)w))
==> (/b.b)(/y.(/z.z)w)
==> /y.(/z.z)w
Reduced to:/y.(/z.z)w
```

As we can see, the reduction procedure is different from previous. It differs from previous two strategies at the second reduction step. Since the call-by-value reduction always reduces the argument before the substitution take place, it reduces the redex in the argument $((\lambda x.x)(\lambda y.(\lambda z.z)w))$ before it replaces the bound variables in the function $(\lambda b.b)$.

1.4.4 Head Reduction

The head reduction performs reductions inside lambda abstractions, but only in head position. Notice that, the *head reduction* strategy introduced in this project is the same as defined by Barendregt [1], however it differs from Sessoft's [7] *head spine reduction*. In the leftmost head reduction, only head redexes are reduced. A redex $((\lambda x.e_0)e_1)$ is a *head redex* if it is proceded to the left only by lambda abstractions of non-redexes, as in $\lambda x_1 ... \lambda x_n.(\lambda x.e_0)e_1 ... e_m$ when $n \geqslant 0$ *and* $m \geqslant 1$.

$$\llbracket x \rrbracket^{ls}_{hr} = x$$

$$\frac{\llbracket e \rrbracket^{ls}_{hr} = e'}{\llbracket (\lambda x.e) \rrbracket^{ls}_{hr} = (\lambda x.e')}$$

$$\frac{\llbracket e_1 \rrbracket^{ls}_{cbn} = (\lambda x.e) \qquad \llbracket e[e_2/x] \rrbracket^{ls}_{hr} = e'}{\llbracket (e_1 e_2) \rrbracket^{ls}_{hr} = e'}$$

$$\frac{\llbracket e_1 \rrbracket^{ls}_{cbn} = e_1' \neq (\lambda x.e)}{\llbracket (e_1 e_2) \rrbracket^{ls}_{hr} = (e_1' e_2)}$$

Head Reduction: Big-step operational semantics

$$\frac{}{(\lambda x.M)N \to_\beta M[N/x]} \qquad \frac{M \to_\beta N}{MP \to_\beta NP} \qquad \frac{M \to_\beta N}{(\lambda x.M) \to_\beta (\lambda x.N)}$$

Head Reduction: Small-step operational semantics

The head reduction strategy generates terms in head normal form. Recall that, a term is in head normal form if it has the form: $\lambda x_1 \ldots x_n.xM_1 \ldots M_m \quad n, m \geqslant 0$

```
headreduction :: Term -> Term
headreduction (Var x) = Var x
headreduction (Abs x t) = (Abs x (headreduction t))
headreduction (App (Abs x t) y) = subs t (Var x, y)
headreduction (App x y) | x == evalcbn x = (App x y)
                        | otherwise = (App (evalcbn x) y)
```

The first 3 function clauses are similar as before, it transfers the semantic rules straght forward into Haskell functions. The last function clause uses Call-by-Name function defined in 1.4.2, because it has to avoid premature reduction of inner redexes.

***Example* 1.4.4** The sample example run by Head Reduction is as follows:

```
    (/a.a)(/b.b)((/x.x)(/y.(/z.z)w))
==> (/b.b)((/x.x)(/y.(/z.z)w))
==> (/x.x)(/y.(/z.z)w)
==> /y.(/z.z)w
==> /y.w
Reduced to:/y.w
```

The reduction procedure is the same as normal order reduction. Notice that, in the third reduction step, the redex $(\lambda z.z)w$ is a head redex in the abstraction $\lambda y.(\lambda z.z)w$.

1.4.5 Applicative Order Reduction

Applicative order reduction $e \xrightarrow{ao} e'$ reduces the leftmost innermost redex first. A function's arguments are always reduced before the function itself. Applicative order always attempts to apply functions to normal forms. It differs from Call-by-Value only by reducing also under abstractions:

$$\llbracket x \rrbracket_{ao}^{ls} = x$$

$$\frac{\llbracket e \rrbracket_{ao}^{ls} = e'}{\llbracket (\lambda x.e) \rrbracket_{ao}^{ls} = (\lambda x.e')}$$

$$\frac{\llbracket e_1 \rrbracket_{ao}^{ls} = (\lambda x.e) \qquad \llbracket e_2 \rrbracket_{ao}^{ls} = e_2' \qquad \llbracket e[e_2'/x] \rrbracket_{ao}^{ls} = e'}{\llbracket (e_1 e_2) \rrbracket_{ao}^{ls} = e'}$$

$$\frac{\llbracket e_1 \rrbracket_{ao}^{ls} = e_1' \neq (\lambda x.e) \qquad \llbracket e_2 \rrbracket_{ao}^{ls} = e_2'}{\llbracket (e_1 e_2) \rrbracket_{ao}^{ls} = (e_1' e_2')}$$

Applicative Order: Big-step operational semantics

$$\frac{}{(\lambda x.M)N \rightarrow_\beta M[N/x]} (M,\ N \textit{ in normal form}) \qquad \frac{M \rightarrow_\beta N}{MP \rightarrow_\beta NP} \qquad \frac{M \rightarrow_\beta N}{PM \rightarrow_\beta PN} (P \textit{ contains no redex})$$

$$\frac{M \rightarrow_\beta N}{(\lambda x.M) \rightarrow_\beta (\lambda x.N)}$$

Applicative Order: Small-step operational semantics

Applicative orde reduction always generates terms in normal form. Since it also reduce the redexes in abstractions. The applicative order reduction is not normalizing, functions applied to non-normalizing arguments are non-normalizing.

```
apporder :: Term -> Term
apporder (Var x) = Var x
apporder (Abs x t) = (Abs x (apporder t))
apporder (App (Abs x t) y) = subs t (Var x, y)
apporder (App x y) | x == apporder x = (App x (apporder y))
                   | otherwise = (App (apporder x) y)
```

The function clauses are simliar to the Call-by-Value introduced in 1.4.3. The only difference is the redexex inside abstractions are also reduced.

***Example* 1.4.5** The sample example run by Applicative Order reduction in Haskell is as follows:

```
    (/a.a)(/b.b)((/x.x)(/y.(/z.z)w))
==> (/b.b)((/x.x)(/y.(/z.z)w))
==> (/b.b)((/x.x)(/y.w))
==> (/b.b)(/y.w)
==> /y.w
Reduced to:/y.w
```

Since applicative order reduction reduces the leftmost innermost redex first, it firstly reduces the redex `(/a.a)(/b.b)`. It always reduces the arguments before the function, so it reduces the argument `((/x.x)(/y.(/z.z)w))` until it contains no redex(in normal form). When the argument is in the normal form `(/y.w)`, the substitution takes place and finally it reaches the normalized term `(/y.w)`.

1.5 Comparisons between Different Reduction Strategies

1.6 λ-calculus with Explicit Substitution and Garbage Collection

We currently have introduced the untyped λ-calculus with implicit substitution. When we perform β-reduction on an application $(\lambda x.xz)y$ whose function is an abstraciton, the substitution is done in 1 single step: that is, all unbound occurrences of variable x are substituted by the argument y. However, as a term becomes complex, it gives less details about how the substitution is done. In this section, a new calculi with explicit substitution built based on ordinary λ-calculus is introduced. It generates substitutions and treats them as a λxgc-term. A set of explicit substitution rules are defined to reduce a term into normal form.

1.6.1 The explicit substitution calculi

In Bloo et al. [2], a new λ-calculus with explicit substitution and garbage collection is introduced and studied. It retains the use of traditional veriable names, specifying terms modulo renaming and includes reduction rules for *explicit garbage collection*. It is a conservative extension of the ordinary λ-calculus from several properties. The λxgc-calculus with syntactic reductions for explicit substitution and garbage collection is represented.
The set of λxgc-terms Λx, ranged over by $MNPQ$, is formally defined by:

Definition 1.7 (λxgc-terms). λxgc-terms are the the class satisfing the following grammar:

$$M ::= \ x \mid \lambda x.M \mid MN \mid M\langle x := N\rangle$$

where the letters $xyzvw$ range over an infinite set of variables. The λxgc-calculus follows the parenthese omission of original λ-calculus and write λxy.M for $\lambda x.(\lambda y.M)$ and $MN(PQ)$ for $(MN)(PQ)$; explicit substitution is given highest precedence so $\lambda x.MNP\langle x := Q\rangle$ is $\lambda x.((MN)(P\langle x := N\rangle)$.

The λxgc-terms include as a subset the ordinary λ-terms Λ; a λxgc-terms is 'pure' if it is also a λ-term, *i.e.*, if it has no subterms of the form $M\langle x := N\rangle$.

Definition 1.8 (free and bound variables of λ-xgc-terms) The set of free and bound variables are defined inductively by the following function:

$FV_{xgc}(x)$	$=$	$\{x\}$	$BV_{xgc}(x)$	$=$	$\emptyset$
$FV_{xgc}(\lambda x.M)$	$=$	$FV_{xgc}(M)\backslash\{x\}$	$BV_{xgc}(\lambda x.M)$	$=$	$BV(M) \cup \{x\}$
$FV_{xgc}(MN)$	$=$	$FV_{xgc}(M) \cup FV_{xgc}(N)$	$BV_{xgc}(MN)$	$=$	$BV_{xgc}(M) \cup BV_{xgc}(N)$
$FV_{xgc}(M\langle x := N\rangle)$	$=$	$(FV_{xgc}(M)\backslash\{x\}) \cup FV_{xgc}(N)$	$BV_{xgc}(M\langle x := N\rangle)$	$=$	$BV_{xgc}(M) \cup x \cup BV_{xgc}(N)$

The last free and bound variable set definition of substitution is new. All the free occurrences of x in M is actually bounded by x in the substitution $\langle x := N\rangle$.

Definition 1.9 (λxgc-term concepts).

- **(α-equivalent)** Two terms are α-equivalent, written $M \equiv N$, is as for λ-calculus plus (the new) $M\langle x := N\rangle \equiv P\langle y := Q\rangle$ if $N \equiv Q$ and $M[x := z] \equiv P[y := z]$ for $z \notin FV(M)$
- **(Garbage)** The substitution $\langle x := N\rangle$ in $M\langle x := N\rangle$ is called *garbage* if $x \notin FV(M)$

Definition 1.10 ('raw' reduction step of xgc)

- **Substitution generation**, $\underset{b}{\rightarrow}$, is

$$(\lambda x.M)N \rightarrow M\langle x := N\rangle \qquad \text{(b)}$$

- **Explicit Substitution**, $\underset{x}{\rightarrow}$ is

$$\begin{aligned}
x\langle x := N\rangle &\rightarrow N && && \text{(xv)}\\
x\langle y := N\rangle &\rightarrow x && if\ x \neq y && \text{(xvgc)}\\
(\lambda x.M)\langle y := N\rangle &\rightarrow \lambda z.M[x := z]\langle y := N\rangle && if\ x \in FV(N)\ \&\ x \neq y, z\ new && \text{(xaba)}\\
(\lambda x.M)\langle y := N\rangle &\rightarrow \lambda x.M\langle y := N\rangle && if\ x \notin FV(N)\ \&\ x \neq y && \text{(xab)}\\
(M_1M_2)\langle y := N\rangle &\rightarrow M_2\langle y := N\rangle M_2\langle y := N\rangle && && \text{(xap)}
\end{aligned}$$

- **Garbage Collection**, $\overrightarrow{gc}$, is

$$M\langle x := N\rangle \rightarrow M \qquad if\ x \notin FV(M) \qquad \text{(gc)}$$

- $\underset{xgc}{\longrightarrow}$ is the union of $\underset{x}{\rightarrow}$ and $\underset{gc}{\longrightarrow}$.
- $\underset{bxgc}{\longrightarrow}$ is the union of $\underset{b}{\rightarrow}$ and $\underset{xgc}{\longrightarrow}$.

***Example* 1.6.1** The sample run of normal order reduction with explicit substitution and garbage collection is as following('\' as λ):

```
(1)          (\x.(\y.x)x)(\z.q)          whole term is a redex
(2)      ==> ((\y.x)x)<x:=\z.q>          xgc-reduction
(3)      ==> ((\y.x)<x:=\z.q>)(x<x:=\z.q>)
(4)      ==> (\y.x<x:=\z.q>)(x<x:=\z.q>)
(5)      ==> (x<x:=\z.q>)<y:=x<x:=\z.q>>  garbage collection
(6)      ==> x<x:=\z.q>
(7)      ==> \z.q
```

Recall that, the normal order reduction reduces the leftmost outermost redex first. At the beginning, the whole term `(\x.(\y.x)x)(\z.q)` is a redex, and the function of the application is an abstraction. Therefore, the *substitution generation* $\overrightarrow{b}$ is performed. Further, the substitution is pushed into the application and multiple explicit substitution steps are performed. At step (4), another *substitution generation* $\overrightarrow{b}$ is performed Since there is no free occurrence of y in `x<x:=\z.q>`, the substitution `<y:=x<x:=\z.q>>` is the garbage and removed. Finally, the free occurrence of x is substituted by `\z.q`.

***Example* 1.6.2** Below is another sample run by applicative order reduction with explicit substitution and garbage collection:

```
(1)          (\x.(\y.x)x)(\z.q)      reduce the innermost redex (\y.x)x
(2)      ==> (\x.x<y:=x>)(\z.q)      xgc-reduction
(3)      ==> (\x.x)(\z.q)
(4)      ==> x<x:=\z.q>
(5)      ==> \z.q
```

Recall that, the applicative order reduction reduces the leftmost innermost redex first. The redex `(\y.x)x` is the leftmost and innermost redex at beginning. Therefore, the *substitution generation* is performed inside the function. Since there is no free occurrence of y in x, the substitution `<y:=x>` is the garbage. At step (3), another *substitution generation* is performed and finally we get the term `\z.q`.

1.7 Normal β-reduction vs Explicit Substitution ans garbage collection

As might be expected, it takes more reduction steps to reduce a λ-term to normal form using $\underset{bxgc}{\longrightarrow}$ than it takes using $\underset{\beta}{\rightarrow}$. In this section, it compares the difference between the number of reduction steps needed by normal β-reduction and the explicit substitution and garbage collection. In addition, it is separated into two parts: β-reduction versus explicit substitution(to see the number of steps increased), and explicit substitution versus $\underset{bxgc}{\longrightarrow}$(to demonstrate the efficiency improved by gargage collection).

1.7.1 $\underset{\beta}{\rightarrow}$ vs $\underset{bx}{\longrightarrow}$

To illustrate the differences, we use normal order reduction and list the number of steps required to normalize the term by normal reduction and explicit substitution. There is also a line chart to demonstrate the gap between these two methods. We firstly use some simple terms that cost several steps and then we use the term in the form $\lambda v.(\lambda x.I(Ix))(Iv)$; we gradually enlarge the term I, then the reduction steps needed grows.

λ-term	Normal Reduction	Explicit Substitution
$(\lambda x.x)y$	1	2
$(\lambda abc.ac(bc))(\lambda xy.x)$	3	12
$I = \lambda y.y$	4	14
$(\lambda fx.f(fx))(\lambda fx.f(fx))$	6	45
$I = (\lambda yw.w)z$	7	27
$I = (\lambda ywv.wv)z$	9	67
$I = (\lambda ywvs.wvs)z$	11	123
$I = (\lambda ywvsd.wvsd)z$	13	195

Table 1.2: Reduction steps needed for normal reduction and explicit substitution

In Table 1.2, it lists 8 terms from simple to complex. The simple application $(\lambda x.x)y$ only takes one reduction step that substitutes the free occurrence of x in $x[y/x]$ by y. The explicit substitution takes two steps: substitution generation and substitution. The second term $(\lambda abc.ac(bc))(\lambda xy.x)$ needs more steps for explicit substitution since it has to push the substitution into abstraction and further into application until it reaches a variable. The sample run of this term with explicit substitution can also be found in Figure 5.9. When the term $(\lambda fx.f(fx))(\lambda fx.f(fx))$ is normalized with explicit substitution, it needs 45 steps whereas all the β-reduction paths of it only consists 6 steps. Finally, a series of terms in the form $\lambda v.(\lambda x.I(Ix))(Iv)$ are used. We gradually enlarge the term I to increase the number of β-reduction steps and to evaluate the gap between two calculus. As it illustrated in the table, the number of steps needed for explicit substitution grows larger than the square of normal β-reduction when $I = (\lambda ywvs.wvs)z$. When $I = (\lambda ywvsd.wvsd)z$, the number of steps by explicit substitution is 195 which is much larger than the square of the number of β-reductions 13.

We can see that, the number of steps needed by explicit substitution mainly depends on the number of β-reduction required and the structure of abstraction's body. It needs more steps to push a substitution into an application if the structure is complex. For example, the λxgc-term

$(xyz)\langle x := z\rangle$ needs 2 steps to push the substitution into the application xyz: $(xy)\langle x := z\rangle z\langle x := z\rangle$ and then $x\langle x := z\rangle y\langle x := z\rangle z\langle x := z\rangle$. If an application only contains n variables, then it needs $n-1$ steps to push a substitution into it, and it also takes n steps to reduce each λxgc-term in form $x\langle x := y\rangle$. It needs more steps if the application also contains abstractions with complex body.

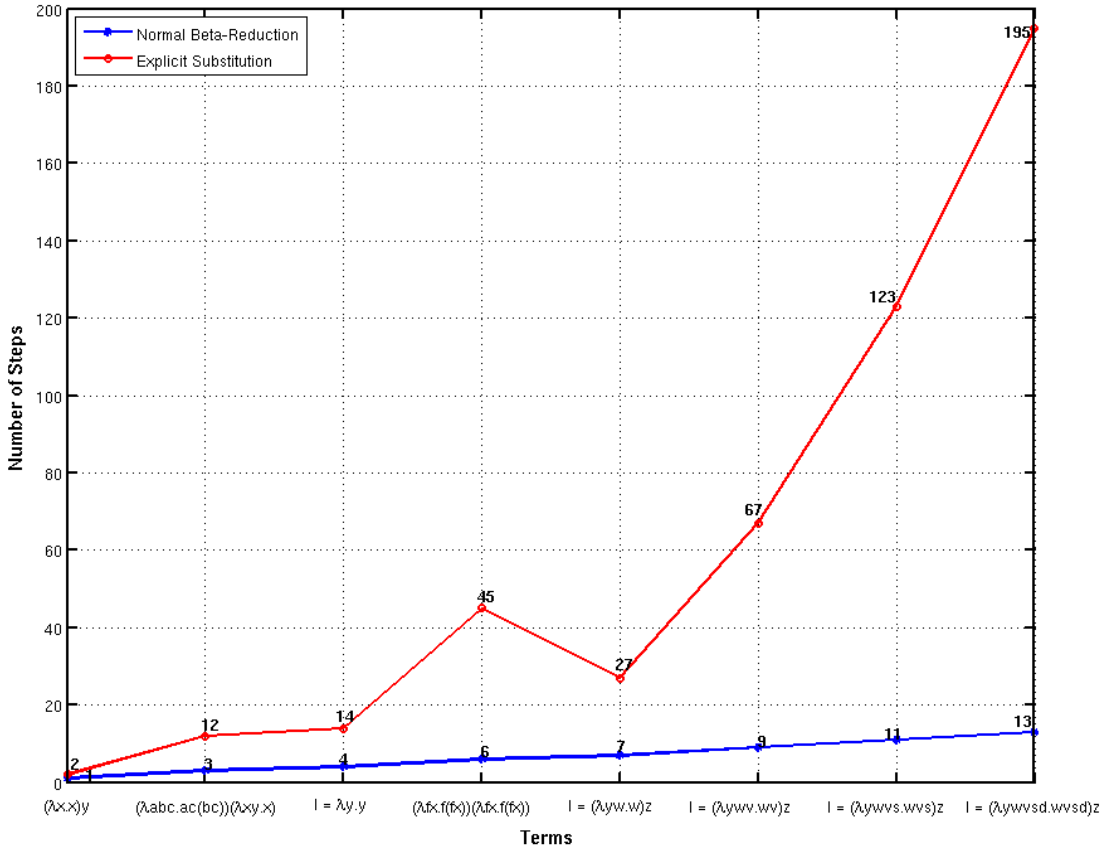


Figure 1.1: Reduction steps needed for normal β-reduction and explicit substitution

The figure above illustrates the difference in the number of steps needed between normal β-reduction and $\underset{bx}{\longrightarrow}$. As the term I grows larger, the number of steps in normal β-reduction grows linearly, while xgc grows approximately exponentially. We can see that the β-reduction steps needed for term $(\lambda fx.f(fx))(\lambda fx.f(fx))$ is less than the steps needed when $I = (\lambda yw.w)z$, however, the number of explicit substitution steps is larger. The number of reduction steps mainly depends on the free occurrences of bound variables in an abstraction. For the term $(\lambda fx.f(fx))$, the free occurrence of f and x in $f(fx)$ lead to repeated substitution and increase the number of reduction steps.

1.7.2 $\underset{\beta}{\rightarrow}$ vs $\underset{bx}{\longrightarrow}$ vs $\underset{bxgc}{\longrightarrow}$

The explicit garbage collection simplifies several proofs. For the λxgc-term $(xy)\langle z := M\rangle$, since there is no free occurence of z in the term xy, so the substitution $\langle z := M\rangle$ can be removed. Then it no longer needs to be pushed into the application xy which costs more explicit substitution steps. The following table and chart would show the number of steps decreased, or in other words, the efficiency improved by garbage collection.

Compared with Table 1.2, Table 1.3 lists additional the number of steps needed by explicit substitution and garbage collection. As we can see, there is no garbage collection performed for the term $(\lambda x.x)y$ and $(\lambda abc.ac(bc))(\lambda xy.x)$. When $I = \lambda y.y$, the garbage collection takes place at the 2nd and 6th reduction step when the term is `\v.((\y.y)<x:=(\y.y)v>)(((\y.y)x)<x:=(\y.y)v>)`

λ-term	Normal Reduction	Explicit Substitution	bxgc
$(\lambda x.x)y$	1	2	2
$(\lambda abc.ac(bc))(\lambda xy.x)$	3	12	12
$I = \lambda y.y$	4	14	12
$(\lambda fx.f(fx))(\lambda fx.f(fx))$	6	45	45
$I = (\lambda yw.w)z$	7	27	18
$I = (\lambda ywv.wv)z$	9	67	35
$I = (\lambda ywvs.wvs)z$	11	123	58
$I = (\lambda ywvsd.wvsd)z$	13	195	87

Table 1.3: Reduction steps needed for normal reduction, explicit substitution, and bxgc

and `\v.((\y.y)<x:=(\y.y)v>)(x<x:=(\y.y)v>)`. There is no free occurrence of x in `\y.y` in both cases. So the substitution is the garbage and is removed. Therefore, the substitution would no be pushed into the abstraction and it reduces 2 reduction steps.

For the term $(\lambda fx.f(fx))(\lambda fx.f(fx))$, there is no garbage collection performed because of the repeated free occurence of f and x. When the term I is gradually enlarged, the number of steps reduced by garbage collection becomes larger. It decreases more than a half when $I = (\lambda ywv.wv)z$. The more the number of reduction steps needed by explicit substitution, the more steps reduced by garbage collection. We can see the difference below:

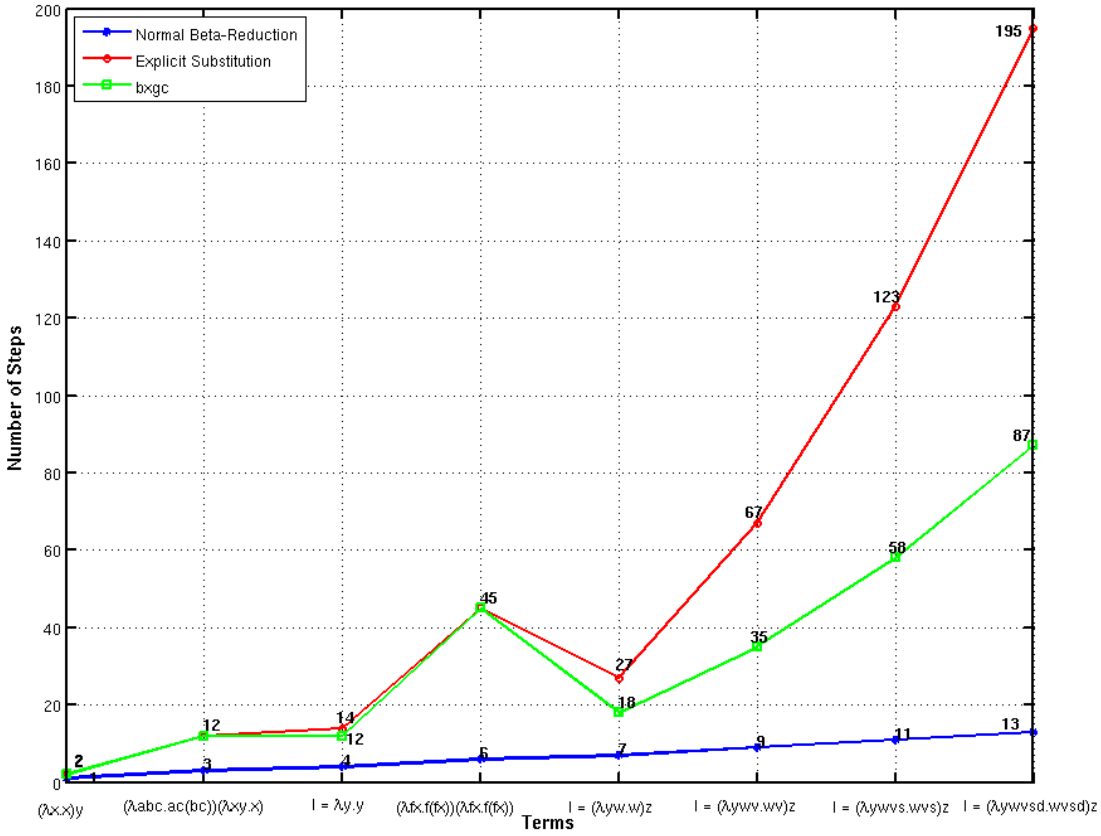


Figure 1.2: Reduction steps needed for normal reduction, bx, and bxgc

The figure above illustrates the difference in the number of steps needed among normal β-reduction, $\underset{bx}{\longrightarrow}$, and $\underset{bxgc}{\longrightarrow}$. There is not too much difference between $\underset{bx}{\longrightarrow}$ and $\underset{bxgc}{\longrightarrow}$ for the first 4 terms. The garbage collection has a significant impact as the term I grows larger. As we can

see from the graph, the difference between $\underset{bx}{\longrightarrow}$ and $\underset{bxgc}{\longrightarrow}$ becomes larger from $I = (\lambda yw.w)z$ to $I = (\lambda ywvsd.wvsd)z$.

Although the garbage collection can remove any useless substitutions from λxgc-terms and avoid redundant explicit substitution steps. It takes much more reduction steps than normal β-reduction. The number of steps needed by explicit substitution and garbage collection is approximately $nlogn$(only rough estimated).

It is worth mentioning that the reduction path of a term by different reduction strategies also affects the number of reduction steps for $\underset{bx}{\longrightarrow}$ and $\underset{bxgc}{\longrightarrow}$. The shortest $\underset{bxgc}{\longrightarrow}$ path of the term $I = \lambda y.y$ is 8, using applicative order reduction. It costs 12 steps for the normal order reduction whereas all the β-reduction paths of it consist of 4 steps.

Chapter 2

The Curry type assignment system

A type system consists a set of rules that assign a property called a *type* to the various constructs — such as variable, expression, functions or modules. There are several reasons to add types to a program, one of the most important reason to add type properties to a program is to reduce bugs. It is possible to build an abstract interpretation of programs by treating terms as objects. The abstractions can be regarded as the interface of objects, by checking whether the interfaces have been connected properly we can decide whether the program is defined in a consistent way. Moreover, type systems provide a form of documentation and improve the readability of a program. Since a program can be abstracted, there are less information in the structure of a program. Furthermore, it also enables independent compilation before the implementation of a program runs. The types of functions and arguments can be checked during the generation of codes. Type-checking during implementation enables dynamic debugging which largely reduces the debugging workload at run-time. If a program is error-free, it is safe to run:"Typed programs cannot go wrong". It also allows multiple dispatch which is the feature of some objected-oriented programming languages in which a function or method can be dynamically dispatched based on the run time type of more than one of its arguments.

An exmaple of a type system is the Java language. A Java program consists of classes which contains a set of function definitions. A function is invoked by an instance of a class that the function belongs to. The interface of a function states the type of the return value, the name of the function and a list of values that are passed to the function. The code of an invoking function states the object instance on which this particular method is to be invoked, and the name of this function along with the names of variables that hold values to pass to it. The Java compiler checks the type of each argument that passed into the function, against the type declared for each varaible in the interface of the invoked function. If the types do not match, the compiler throws a compile-time error.

The lambda calculus as treated in Chapter 1 is referred to as a *type-free* theory. Because every expression may be applied to every other expression. For example, the $\lambda x.x$ may be applied to any argument x and generates the same x.

The typed version of lambda calculus(or called Combinatory Logic, a variant of the lambda calculus) is introduced essentially in Curry[3] and in Church(1940). Types are objects of a syntactic nature and may be assigned to lambda terms. If M is a lambda term and a type A is assigned to M, then we say *'M has type A'*; the notation used is '$M : A'$. For example, in a typed system one has $\lambda x.x : (A \to A)$, which means the function $\lambda x.x$ should take an argument in type A and return a value of type A. In general, $A \to B$ is the type of functions from A to B.

2.1 The System $\lambda \to$-Curry

In Curry and Feys(1958) and Curry et al. the type assignment theory was modified in a natural way to the lambda calculus assigning elements of a given set $\mathbb{T}$ of types to type free lambda terms. For this reason these calculi are sometimes called systems of type assignment. If the type $\sigma \in \mathbb{T}$ is assigned to the term $M \in \Lambda$ which writes as $\vdash M : \sigma$, sometime with a under subcript such

as $\vdash_\lambda$ to denote the particular system. Usually a set of assumptions Γ is needed to derive a type assignment write as $\Gamma \vdash M : \sigma$(read as 'Γ yields M in σ').

Definition 2.1 (i) The set of *types*, notation $\mathbb{T}$,

Notation. (i) If $\sigma_1, ..., \sigma_n \in \mathbb{T}$ then

$$\sigma_1 \to \sigma_2 \to ... \to \sigma_n$$

stands for

$$(\sigma_1 \to (\sigma_2 \to ... \to (\sigma_{n-1} \to \sigma_n)))$$

we use association to the right.
(ii) $\alpha, \beta, \gamma, ...$ denote arbitrary type variables.

Definition 2.2 (i) A *statement* is of the form $M : \sigma$, where $M \in \Lambda$ and $\sigma \in \mathbb{T}$(pronounced as 'M *in* σ'). M is called the *subject* and A the *predicate* of $M : A$.

(ii) A *context* Γ is a set of statements with only distinct (term) variables as subjects; In [8], the notion $\Gamma, M : \sigma$ is used for the context $\Gamma \cup \{M : \sigma\}$ where either $M : \sigma \in \Gamma$ or M does not occur in Γ, and we use $M : \sigma$ as shorthand for $\emptyset, M : \sigma$.

Definition 2.3 (CF. [[3], [4]]) Curry type assignment and *derivations* can be defined using following derivation rules.

$$(1) : \frac{}{\Gamma, M : \sigma \vdash_c M : \sigma}$$

$$(2) : \frac{\Gamma, x : \sigma \vdash_c M : \beta}{\Gamma \vdash_c (\lambda x.M) : \sigma \to \beta}$$

$$(3) : \frac{\Gamma \vdash_c M_1 : \sigma \to \beta \qquad \Gamma \vdash_c M_2 : \sigma}{\Gamma \vdash_c (M_1 M_2) : \beta}$$

***Example* 2.1.1** (i)Let $\sigma \in \mathbb{T}$. Then $\Gamma \vdash \lambda fx.f(fx) : (\sigma \to \sigma) \to \sigma \to \sigma$, is shown by the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, f : \sigma \to \sigma \vdash f : \sigma \to \sigma}\,(1) \quad \dfrac{\dfrac{}{\Gamma, f : \sigma \to \sigma \vdash f : \sigma \to \sigma} \quad \dfrac{}{\Gamma, x : \sigma \vdash x : \sigma}}{\Gamma \vdash fx : \sigma}}{\Gamma \vdash f(fx) : \sigma}}{\Gamma \vdash \lambda x.f(fx) : \sigma \to \sigma}\,(1)}{\Gamma \vdash \lambda fx.f(fx) : (\sigma \to \sigma) \to \sigma \to \sigma}\,(2)$$

(ii)Notice that, we cannot type *'self-application'* xx. In order to type the application xx, the derivation should be as following:

$$\frac{\dfrac{}{\Gamma \vdash x : A \to B} \quad \dfrac{}{\Gamma \vdash x : A}}{\Gamma \vdash xx : B}$$

As we can see, the term variable x is assigned to two types: $A \to B$ and A, and either of them can be substituted by the other one. So, $A \to B \neq A$, and there are two statements of subject x in context Γ with distinct predicates. Those two statements would be conflict according to the definition of context. Therefore, the 'self-application' is untypable.

2.2 Subject Reduction

2.3 Properties of $\lambda \rightarrow$

Definition 2.4 (Type substitution(van Bakel[8])) The substitution $(\varphi \mapsto C)$:, where φ is a type variable and $C \in \mathbb{T}$ is defined as following:

$$\begin{array}{ll} (\varphi \mapsto C)\varphi & = C \\ (\varphi \mapsto C)\varphi' & = \varphi', if\ \varphi' \neq \varphi \\ (\varphi \mapsto C)A \rightarrow B & = ((\varphi \mapsto C)A) \rightarrow ((\varphi \mapsto C)B) \end{array}$$

Definition 2.5 Let Id_s be the substitution that replaces all type variables by themselves:

(i) Robinson's unification algorithm. Unification of Curry types is defined by:

$$\begin{array}{llll} unify & \varphi & \varphi & = (\varphi \mapsto \varphi) \\ unify & \varphi & B & = (\varphi \mapsto B), if\ \varphi\ does\ not\ occur\ in\ B \\ unify & A & \varphi & = unify\ \ \varphi\ \ A \\ unify & (A \rightarrow B) & (C \rightarrow D) & = S_1 \circ S_2 \\ & & & \quad where\ S_1 = unify\ \ A\ \ C \\ & & & \qquad\ \ S_2 = unify\ (S_1 B)\ (S_1 D) \end{array}$$

(ii) UnifyContexts

$$\begin{array}{llll} UnifyContexts & (\Gamma_0, x:A) & (\Gamma_1, x:B) & = S_1 \circ S_2 \\ & & & \quad where\ S_1 = unify\ \ A\ \ B \\ & & & \qquad\ \ S_2 = UnifyContexts\ (S_1\Gamma_0)\ (S_1\Gamma_1) \\ UnifyContexts & (\Gamma_0, x:A) & \Gamma_1 & = UnifyContexts\ \Gamma_0\ \Gamma_1, if\ x\ does\ not\ occur\ in\ \Gamma_1 \\ UnifyContexts & \varnothing & \Gamma_1 & = Id_s \end{array}$$

Definition 2.6 Principle Pair Algorithm for Λ

$$\begin{array}{llll} pp_c & x & = & \langle x:\varphi, \varphi\rangle \\ & & & where\ \ \varphi = fresh \\ pp_c & (\lambda x.M) & = & \langle \Pi, A \rightarrow P\rangle,\ if\ pp_c M = \langle \Pi \cup \{x:A\}, P\rangle \\ & & & \langle \Pi, \varphi \rightarrow P\rangle,\ if\ pp_c M = \langle \Pi, P\rangle \quad x \notin \Pi \\ & & & \qquad\qquad where\ \varphi = fresh \\ pp_c & (MN) & = & S_1 \circ S_2\langle \Pi_1 \cup \Pi_2, \varphi\rangle \\ & & & where\ \ \varphi\ \ \ \ = fresh \\ & & & \qquad\ \langle \Pi_1, P_1\rangle = pp_c M \\ & & & \qquad\ \langle \Pi_2, P_2\rangle = pp_c N \\ & & & \qquad\ S_1\ \ \ \ = unify\ P_1\ \ (P_2 \rightarrow \varphi) \\ & & & \qquad\ S_2\ \ \ \ = UnifyContexts\ (S_1\Pi_1)\ (S_1\Pi_2) \end{array}$$

Definition 2.7 Principle Pair Algorithm for Λx

$$\begin{array}{llll} pp_c & (M\langle x := N\rangle) & = & S_1 \circ S_2\langle \Pi_1 \cup \Pi_2, P_1\rangle \\ & & & where\ \ P_3\ \ \ \ = fresh\ \ \ \ \ \ \ \ if\ x\ \notin FV(M) \\ & & & \qquad\ \ \Pi_1\ \ \ \ = \Pi_3 \cup \{x:P_3\}\ \ if\ x\ \in FV(M) \\ & & & \qquad\ \langle \Pi_1, P_1\rangle = pp_c\ M \\ & & & \qquad\ \langle \Pi_2, P_2\rangle = pp_c\ N \\ & & & \qquad\ S_1\ \ \ \ = unify\ P_2\ \ P_3 \\ & & & \qquad\ S_2\ \ \ \ = UnifyContexts\ (S_1\Pi_1)\ (S_1\Pi_2) \end{array}$$

$$(4) : \frac{\Gamma, x:\sigma \vdash M:\beta \qquad \Gamma \vdash_c N:\sigma}{\Gamma \vdash_c M\langle x := N\rangle\beta}$$

Chapter 3

A Purely-Functional Programming Language — Haskell

This chapter will introduce functional programming and a purely-functional programming language-Haskell. It will demonstrate how the lambda calculus and Curry type assignment system is implemented in Haskell by self-defined data types, pattern matching, list processing, and recursive functions. As an implementation and usable software, the issues and weaknesses would also be discussed in this chapter.

3.1 Functional Programming

Functional programming uses functions, other than objects or procedures, as the fundamental building blocks of a program. It is a programming paradigm that treats computation as the evaluation of expressions and avoids mutable data and state, in contrast with imperative programming where programs are composed of statements which change global state when executed. Functional programming has its roots in lambda calculus. Lambda calculus provides a theoretical framework for describing functions and their evaluation. Although it is a mathematical abstraction rather than a programming language, it forms the basis of almost all functional programming languages. There are several functional programming languages such as Miranda, Haskell, and OCaml that is defined based on lambda calculus.

3.2 Haskell

Haskell is an advanced purely-functional programming language, which has non-strict semantcis and strong static typing. In imperative languages(C, C++, Java...), a sequence of tasks are generated and executed in a program. When they are executed, it can change state. It has a control flow structures for doing some action several times. In purely functional programming, we do not need to tell the computer what to do but rather to tell it what each function is. It also cannot set a variables to something and then set it to something else later. In purely functional languages, a function has no side-effects. The only task for a function is to take the arguments, perform operations, and return the result.

Haskell is lazy, which means that it will not execute functions and calculate things unless it is specifically told to do so. If you compile and run a haskell program, it does not call any functions nor perform any execution. The only thing you can do is call a specific function(it may call other functions) and pass parameters into it, then that function is executed and a result is returned.

Haskell is statically typed. When we compile a haskell program, the compiler knows which variable is a number, which is a string and so on. That means errors can be found at compile time. Haskell uses a type system that has type inference. That means we do not have to explicitly label every piece of code with a type because the type system can automatically type it. Type inference also allows the code to be more general. If we add two parameters together without explicitly stating their types, the function will work on any two parameters that can be added as numbers.

Haskell is elegant and concise, because it uses a lot of high level concepts. Haskell programs are usually shorter than their imperative language equivalents. In addition, shorter programs are easier to develop than longer ones and have less bugs.

In this project, Haskell is self-studied thanks to the course note of COMP120.1 of Imperial College. In this section, some main features of Haskell would be introduced by examples.

3.2.1 Pattern Matching

Pattern matching consists of listing patterns to which some data should conform and then checking to see if it does conform to any of them and deconstructing the data according to those patterns. When defining functions, we can define separate function bodies for different patterns. This leads to really neat code that is simple and readable. We can pattern match on any data type. A successful match binds the formal parameters in the pattern. The matching process itself occurs "top-down" and "left-right". Failure of a pattern anywhere in one function body results in failure of the whole matching process, and the next function body is then tried to match. For instance, if [2, 3] is matched against [1, a], then 2 fails to match with 1, so the result is a failed match. Following is another example that defines a function using pattern matching:

```
factorial :: (Integral a) => a -> a
factorial 1 = 1
factorial n = n * factorial (n-1)
```

The code above defines a funtion that calculate the factorial of the input. It uses pattern matching that lists all the possible patterns that can be matched. Here, only two cases, 1 or n. It firstly matches the first function body to check whether the input is 1; if not, goes to next function body and calls itself again.

3.2.2 Guards

Patterns are used to specify data should form to some form and deconstruct it, whereas guards are used to check whether some properties of an argument are true or false. It is similar to an if statement, and more likely to a 'swtich' block in Java. Following is an example of using guards:

```
max :: (Ord a) => a -> a -> a
max a b | a > b     = a
        | otherwise = b
```

The function above returns the maximum value between the inputs a and b. Notice that, 'Ord' here enables ordering on a general type a, so we can apply '¿' '¡' to compare two variables in type a.

3.2.3 Local Definition

The 'where' block is used to simplify complex codes with newly defined variables. It avoids repeated expressions: an expression is only calculated onece and it is binded to a name than then the name can be referred to instead of the whole expression.

```
divA :: Float -> Char
divA a | a/2 <= 10  = 'A'
       | a/2 <= 20  = 'B'
       | a/2 <= 30  = 'C'
       | otherwise  = 'D'
```

The function above firstly calculte the input parameter a divided by 2, and decides which sector it is in and returns a character. It repeatly defines the expression `a/2` for three times. We can avoid that by using 'where' statement as below:

```
divA :: Float -> Char
divA a | b > 10  = 'A'
       | b > 20  = 'B'
       | b > 30  = 'C'
       | otherwise = 'D'
       where b = a/2
```

As we can see, it improves the readability by giving names to an expression and can accelerate programs since expressions like `a/2` is calculated only once. Mention that, these names like `b` are visible across the guards.

3.2.4 Self-defined Types

We can also create our own data types in a very straight forward way. One way is to use the **data** keyword.

```
data Bool = False | True
```

data means that we are defining a new data type. The part on the left of = denotes the type name, which is **Bool**. The parts after the = are value constructors. They specify the different values that this type can have. The symbol | stands for 'or', so the type `Bool` can either be **False** or **True**.

Following is another example that defines a data type **Shape**

```
data Shape = Rectangle Float Float | Circle Float | Triangle Float Float Float
             deriving(Show, Eq)
```

A **Shape** can either be a **Rectangle** or a **Circle** or a **Triangle**. The **Rectangle** value constructor has two fields which are floats. Fields after a value constructor are basically the parameters that type contains. Here, the two fields of **Rectangle** are length and width. Similarly, **Circle** has one field which is the radius. **Triangle** has three fields which stands for the length of its three edges. The 'deriving' statement after the data type declaration enables the defined data type to be shown and to be compared to check if two shapes are the same. For exmaple, 'Circle 2.4 == Circle 2.5' would return false.

3.3 Model the λ-calculus and the Curry Type Assignment System in Haskell

Since the lambda calculus lies at the basis of functional languages like Haskell. Implement lambda calculus in Haskell is quite a straight forward way. In this project, both the untyped and typed lambda calculus are implemented. The implementatoin is mainly in three parts: data types declaration, interpretation, and main mechanism.

3.3.1 Data Types

```
type Name = Char
data Term = Var Name | Abs Name Term | App Term Term
            deriving (Show, Eq)

data Type = TP Type Type | TVar [Char]
            deriving (Show, Eq)

type Predicate = Type
type Context = [Statement]
type Statement = (Subject, Predicate)
```

```
type Subject = Term
type Derivation = (Context, [Statement], Statement)
type PPc = (Context, Type)

data TSub = Sub (Type, Type) | So (TSub, TSub)
            deriving (Show, Eq)
```

3.3.2 Interpretation

3.3.3 Main mechanism

3.4 Issues and Weakness of Implementation in Haskell

The functional programming languages are elegant and concise. The implementation of untyped lambda calculus and Curry type assignment system only contains several hundred lines of code in Haskell. Each function is defined by pattern matching and readable. It clearly shows what each function do according to the input pattern. Local definitions are utilized to avoid duplicate execution on expressions, it makes the program run faster. However, it has some issues; and as an application or software, it has less usability.

Chapter 4

The λ-calculus and Curry Type Assignment System Implementation in Java

The Java version of implementation is transferred directly from Haskell. However, the semantics and operational order of Haskell is difference from Java, especially the 'where' statement of Haskell. The 'where' declaration is called when the defined variables are used, however, in Java, we need to separate a 'where' block into parts according to where it is called. An example is shown in Section 4.1. This differece is worth taking care of when transfer the Haskell into Java, otherwise, NullPointerException may occur.

In all data types, the method toString() has been implemented – more precisely overridden, as it is a method of the Java primary class `Object`. This function is used to print out the data type as a string. There are more method signatures defined in the interfaces as a reference to instantiable classes.

4.1 Different operational order between Haskell and Java

As mentioned above, the operaional order of Haskell is different from Java, not only the pattern matching, but also the 'where' declarations. Since the code in Haskell cannot be transferred into Java directly, it largely abandoned the simplicity of codes. Following is a piece of code that defines principle pair algorithm:

```
ppc :: Term -> [Char] -> (PPc, [Char])
ppc (Abs x y) r | contains (Var x) pi = ((removeItem (Var x, a) pi, (TP a p)), tl1)
                | otherwise = ((pi, (TP f p)), tl1)
                            where (f, tl) = fresh r                    (1)
                                  ((pi, p), tl1) = ppc y tl            (2)
                                  (_, a) = search (Var x) pi           (3)
```

'Guards' are used as an if-then-else block. There are three declarations in the 'where' block each with a distinct number. For the first condition of ppc function, declaration (1)(2)(3) are called since all the declared variables are used. In the 'otherwise' condition, only declaration (1)(2) are called. In this case, in order to transfer the Haskell function into Java methods, we need to repetitively define the variables which largely reduces the simplicity of codes.

```
public PPC ppc(Term term, String counter){
     ...
     else if(term instanceof Abstraction){
        Variable xv = new Variable(((Abstraction) term).getName());
        PPC receiver = ppc(((Abstraction) term).getTerm(), counter);
```

```
        if(receiver != null){
           if(contains(xv, receiver.getSubject())){
             Type searchType = search(xv, receiver.getSubject()).getPredicate();
             ArrayList<Statement> original = receiver.getSubject().getContext();
             ArrayList<Statement> remove = new ArrayList<>();

             for(Statement s: original){
               if(s.getSubject().equals(xv)) remove.add(s);
             }

             for(Statement rm: remove){
                original.remove(rm);
             }

             TP tp = new TP(searchType, receiver.getPredicate());
             return new PPC(new Context(original), tp, receiver.getCounter());
           }
           else{
              TVar f = new TVar(receiver.getCounter().substring(0, 1));
              return new PPC(receiver.getSubject(), new TP(f, receiver.getPredicate()),
                             receiver.getCounter().substring(1));

           }
         }
         else {
             return null;
         }
      }
      ...
}
```

4.2 λ-calculus Representation

4.2.1 λ-terms

A Lambda term can either be a variable, an abstraction or an application. An interface 'Term' is defined as a reference type, which has three lambda term implementations. This is the generic representation of all lambda terms.

This interface defines the method `equals(Term t)`, which return a boolean states whether it is equal to the input term. This method overrides the `equals(Object o)` method of the Java primary class `Object`. It is essential in the reduction procedure, since we can decide whether a term is reducible by compare the term before reduction and after. If the term is the same as it before reduction, then it cannot be reduced. Therefore, we know when we can stop the reduction procedure.

As mentioned in Section 1.4, there are five reduction strategies enabled in the reducer. So there are five method signatures defined in the interface: evaluateNormal(Bool exsub), evaluateCbn(Bool exsub), evaluateCbv(Bool exsub), headReduction(Bool exsub), applicativeOrder(Bool exsub). Each of these method refers to a specific reduction strategy as the method name.

The method mirror() is used to create a mirror term with exactly the same class variables. This method is used when the substitution is performed. For example, if we have a lambda term $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$, it is reduced to $\lambda f.f((\lambda x.f(xx))(\lambda x.f(xx)))$. If we do not create a mirror term of $(\lambda x.f(xx))$, both of these two $(\lambda x.f(xx))$ would refer to the same memory space. In other words, the abstraction $(\lambda x.f(xx))$ is used twice to form an application. Therefore, if operations performed on the function, it is also performed on the argument. It is essential to create a mirror

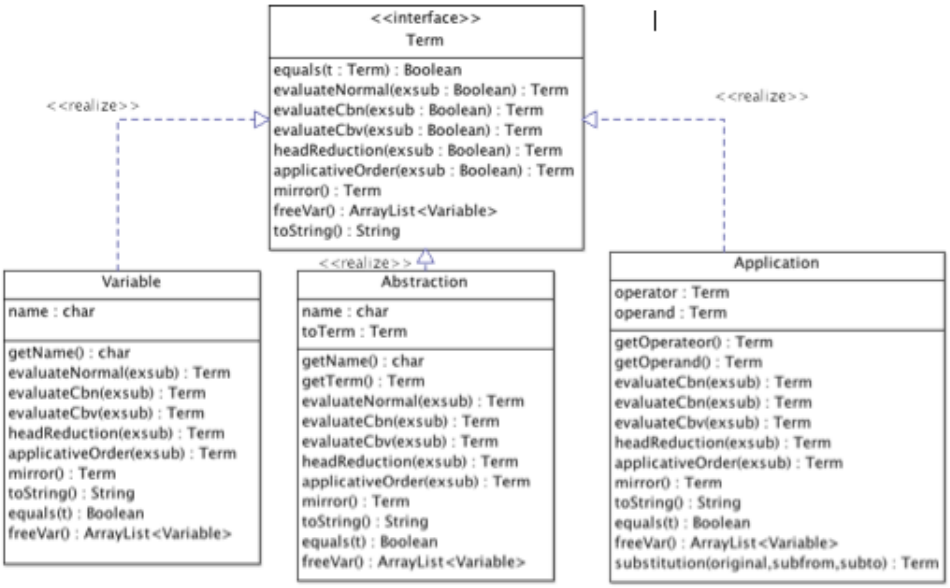


Figure 4.1: Class diagram of Lambda Calculus data structures

term that separate those two terms although they look exactly the same.

The method freeVar() is used to get all the free variables of a λ-term. It is used to perform α-conversion when we substitute bound variables in an abstraction. The free variables are defined in Definition 1.1.

Finally, the toString() method of class Object is overridden to transform a lambda term to a string.

Notice that, in the implementation class `Application`, there is a substitution(Term original, Term subfrom, Term subto):Term method. It is used to substitute bound variables in an abstraction, which is an application function, to the argument. The first parameter of the method is the body of abstraction, the second argument states which bound variable could be substituted and the third argument states what it could be substituted to.

4.2.2 Interpreter

To allow interactions and inputs from users, the program needs to interpret the input term into a data structure in the system. Figure 4.2 illustrates all the methods that used to interpret an input into a lambda term data structure.

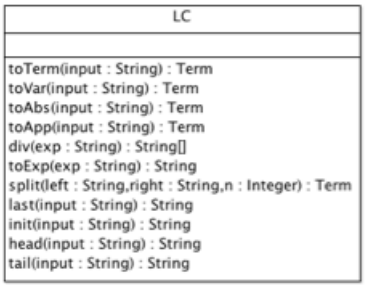


Figure 4.2: Interpretation methods in LC class

The method toTerm(String input) takes a string as input and returns a Term. It calls toVar(String input), toAbs(String input) or toApp(String input) according to type of the outermost term. Then, those three methods iteratively call each other until the whole term is interpreted.

div(String exp) divides an application into the function and argument. Basically, it return an array with 2 string elements. The method toExp(String exp) is used to remove brackets of

applications and returns an expression that is bracker-free. Finally, the most important method split(String left, String right, Int n) implements the bracket removal mechanism. It marks the right most right bracker ')' and find the matching left bracket '(' and extract the term out without brackets.

Since the Java implementation is directly transferred from Haskell, there are also four basic string operations which are Haskell builit-in functions. last(String input) takes a string and returns the last character of the string, it returns itself if the length of input string is 1. init(String input) accepts a string and returns the string without its last character, it returns an empty string if it only contains one character. head(String input) takes a string and returns the first element of the string, it returns itself when the input string length is 1. tail(String input) takes a string and returns the string without its first element, it returns an empty string when the length of input is 1.

Following is the example of how those Haskell built-in function work:

```
init "Hello"   "Hell"          head "Haskell"   "H"
init "A"       ""              head "A"         "A"

last "String"   "g"            tail "Lambda"   "ambda"
last "A"        "A"            tail "A"         ""
```

4.3 Curry Type Assignment System Representation

4.3.1 Types

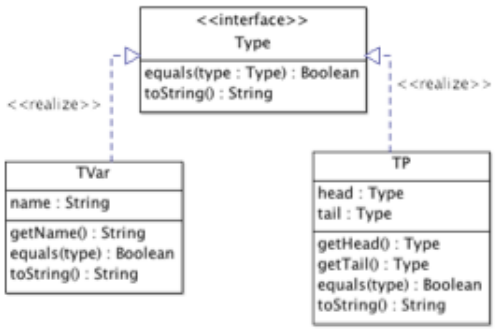
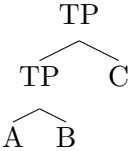


Figure 4.3: Class diagram of type representation

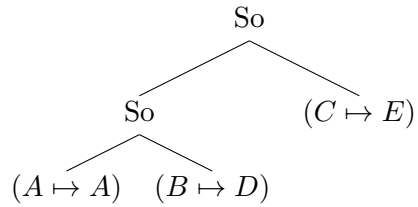
Type assignment system assigns types to λ-terms. In $\mathbb{T}$, a type can either be a single type$(A, B, ...)$ or a complex type$(A \to B)$. Since the complex type is formed by single types, it can be represented using the binary tree structure, where single types are leafs and complex types are branches. For example, the type $(A \to B) \to C$ is represented as TP(TP (TVar A)(TVar B))(TVar C), or in a more intuitionistic way:



Illustrated in Figure 4.3, the interface Type is defined as a reference type. It defines two basic method signatures: equals(Type type) and toString() which override the primary methods in Object. There are two implementation class: TVar and TP which stand for single type and complex type. TVar contains a class variable name which states the type name. It implements method signatures defined in the interface and a get method that returns the type name. The implementation class TP is similar to a Node of a tree structure, which contains two class variables: left branch and right branch. Method signatures and get methods are implemented in the class.

4.3.2 Type substitution

Principal types for λ-terms are defined using the notion of unification of types that defined by Definition 2.5. Similar to types structure in Section 4.3.1, a type substitution can either be a single substitution($\varphi \mapsto B$) or a complex substitution($S_1 \circ S_2$). It could also be represented as a binary tree structure. For example, when we unify $(A \to B) \to C$ and $(A \to D) \to E$, we would get the complex substitution$((A \mapsto A) \circ (B \mapsto D)) \circ (C \mapsto E)$ which also can be represented as a binary tree:



As shown in Figure 4.4, an interface TSub is needed as a reference type. It doesnt have any method signatures, operation on substitutions are defined as auxiliary function in class LC.

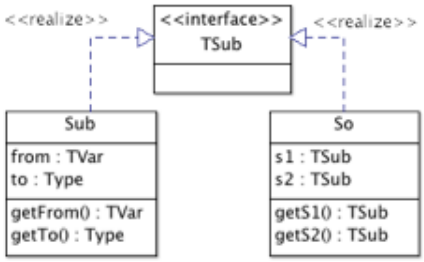


Figure 4.4: Class diagram of type substitutions representation

4.3.3 Other data types

The Curry type assignment system defines other concepts besides types and substitutions. Statement is essential to describe the type assigned to a λ-term; Context is used to collect all statements used for free variables of a term when typing that term; and PPC is a data type that stores context and the type assigned to that term typed.

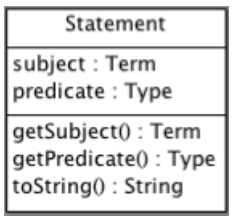


Figure 4.5: Class diagram of Statement

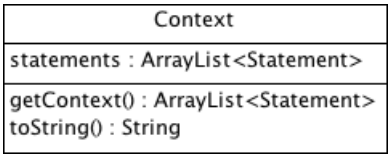


Figure 4.6: Class diagram of Context

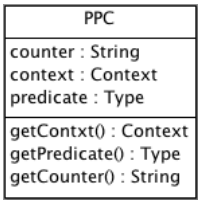


Figure 4.7: Class diagram of PPC

The Statement class has two attributes which are subject and predicate. It contains basic get method that return private attributes, and it also overrides the toString() method to transform a Statement into a string as "subject : predicate"

The Context class only contains an ArrayList of Statement as attribute. A context could have none statement if all the variables are bound. The toString() method returns the context a string in the format: "statement1, statement2, ..."

The PPC class, storing principle pair algorithm results, is a little more complex. As defined in Definition 2.6, the ppc function return a context and the type assigned to the term typed as ⟨context, type⟩. A PPC instance, stores the result of ppc function, therefore it contains two attributes as context and type. In additional, it also has a counter attribute. Since principle pair

algorithm always create fresh types, we should keep track of what type names are available that have not been used. It basically states all the available type names.

4.3.4 Core type assignment mechanism implementation

The Curry type assignment implementation is built based on the lambda calculus implementation. As it extends more operation on λ-terms for type assignment, there are more auxiliary methods defined in class LC.

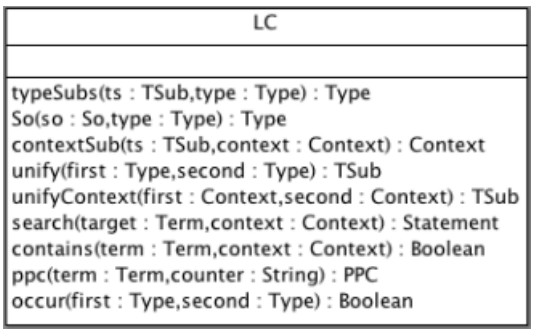


Figure 4.8: Class diagram of Lambda Calculus data structures

The methods listed in Figure 4.8 are the core type assignment mechanism implementations. The top-level method is ppc(Term term, String counter), when a term comes in and needs to be typed, the ppc method is called and it iteratively calls other auxiliary methods. Finally, it will return a PPC instance which contains a context and the typed assigned.

Given a type substitution and a type, the method typeSubs(TSub, Type) substitutes all the type variables according to the subsitution. It implements the type substitution defined in Definition 2.4. Notice that, since a TSub can either be a single substitution Sub or a complex So as defined in Section 4.4, it calls So(So, Type) method that implements the So substitution described in Definition 2.4 (b).

The method contextSub(TSub, Context) implements the context substitution defined in Definition 2.5 (c).

The method unify(Type, Type) implements the type unification algorithm defined in Definition 2.4. And unifyContext(Context, Context) implements the context unification in Definition 2.5 (ii). It should be mentioned that, since Id_s stands for the substitution that replaces all type variables by themselves, it could be simplified as a substitution that only contains the substitution $(A \mapsto A)$. We do not care what the type A is and whether it is a valid type variable, becuase the substitution $(A \mapsto A)$ would not affect any unifications.

The search(Term, Context) methods is an auxiliary method that given a λ-term and a context, it returns the corresponding statement of that term in the context. This method is used in two places. Firstly, it is used when we want to unify the context Γ_0 and Γ_1. We need to find the statements with the same subject in these two context and unify them. Therefore, when we traverse through the context Γ_0, the search method is neccessary to find the corresponding statement in Γ_1. Secondly, it is used in the principal pair algorithm when we type an abstraction. The temporary type assigned to bound variables should be removed from the context. Therefore, given a variable, the search method can find the corresponding statement and further be removed.

The method contains(Term, Context) takes a λ-term and a context as arguments and return a boolean states whether the context contains a statement for the term.

ppc(Term, String) is the core principal pair algorithm implementation method. The definition of principal pair algorithm can be found in Definition 2.6. The second argument in String type is the counter of all available type names.

Finally, the method occur(Type, Type) is used to decide whether a type occurs in another. It is used in the second type unificatin algorithm defined in Definition 2.5.

4.4 Explicit substitution and garbage collection

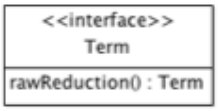


Figure 4.9: Class diagram of Lambda Calculus data structures

4.5 Syntax

Chapter 5

GUI development

Haskell has at least four toolkits for programming a graphical interface: wxHaskell, Gtk2Hs, hoc, and qtHaskell. However, Java Graphics APIs–AWT and Swing–provide a huge set of reusable GUI components, such as button, text field, label, panel and frame for building GUI applications. So, it is more flexible to develop a GUI by reusing these classes in Java. I believe it would also be an interesting work to implement a comprehensive GUI in Haskell. In this project, a useable and functional GUI is developed in Java. The GUI design is based on Sestoft's 'lambda term reducer' which could be found at "http://www.itu.dk/people/sestoft/lamreduce/", and the draft report describing the implementation in Sestoft[6]. Some improvements have been made such as to deal with non-terminating terms in trace enviroment. In addition, the Curry type assignment system is also enable in the GUI which is run in parallel with λ-term reduction.

5.1 Layout

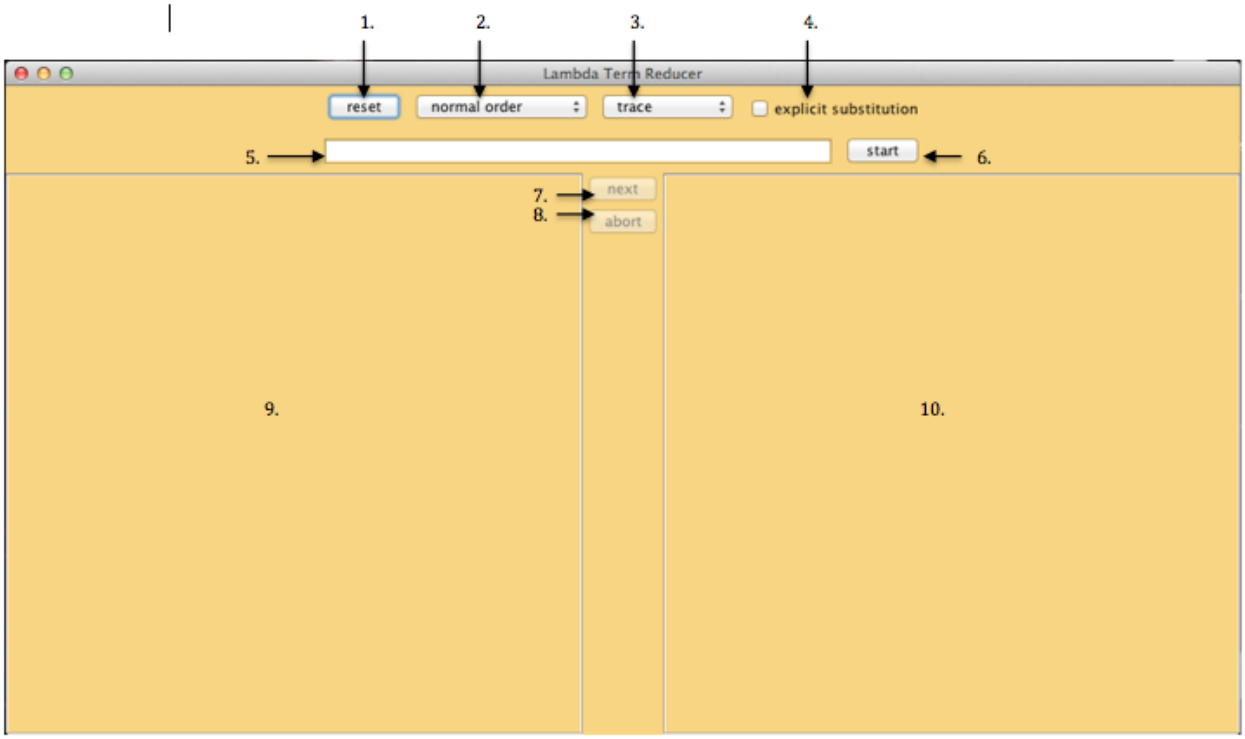


Figure 5.1: Layout of the Graphical User Interface

The function of each button or text area label in Figure 5.1:

1. reset button: reset all the button, drop-down lists values to default and empty the output areas 9 and 10.

2. reduction strategy drop-down list: 5 reduction strategies can be selected by the user.

3. reduction step drop-down list: the user can select how the reduction is performed: normalize: directly output the reduction result by a specific reduction strategy and states how many β-reductions performed. trace: output the reduction procedure for each β-reduction performed. single step: it only outputs the reduction of a single step, moreover, the buttons 7 and 8 are enabled to let user choose whether continue reduction or abort the procedure.

4. explicit substitution check box: user can tick to enable explicit substitution and garbage collection.

5. input field: user input the term to be reduced and typed in this field.

6. start button: press the button to start the procedure with pre-specifed reduction strategy(2) and reduction step option(3).

7. next button: carry on reduction when single step is selected.

8. abort button: abort the reduction when the input term is non-terminate or in the middle of single step reduction.

9. reduction output area: the output of reduction procedure is printed out in this area. The format is decided by the reduction step option(2).

10. type assignment output area: the output of type assignment result is printed out in this area. For normalize and trace, only the input term is typed. However, if the single step strategy is selected, intermediate terms during reduction would also be typed.

5.2 Functionalities

In this section, the functionalities of the GUI would be shown by some sample runs. We mainly test the difference in different tracers: normalize, trace, and single-step. Moreover, the improvements compared to Sestoft's lambda term reducer would be shown by example. Here, we will not give any examples on the differences between outputs by different reduction strategies, since it has already been shown in Chapter 1.

5.2.1 'Normalize' tracer

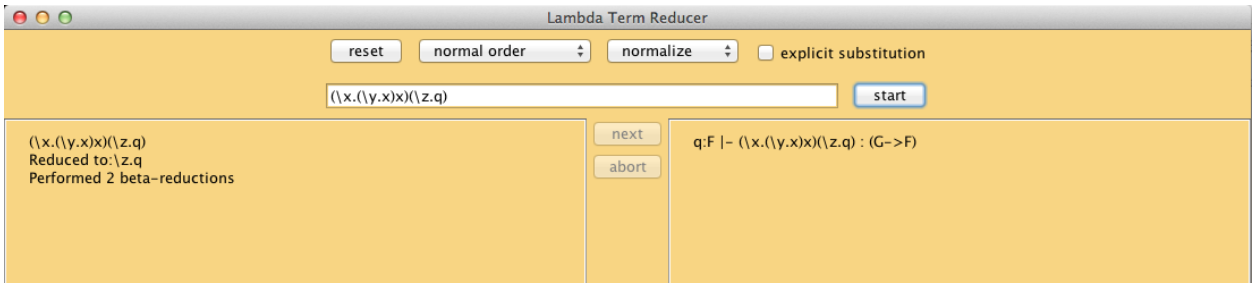


Figure 5.2: Sample run in normalize tracer

When 'normalize' tracer is selected, the reducer tries to normalize the input λ-term with a specific reduction strategy. In other words, it reduces the input term with a reduction strategy until no more β-reduction can be performed. The intermediate terms and the reduction procedure would not be shown in this case. It outputs the final term reached and displays the total β-reduction performed. On the right output area, the type assignment result is displayed. It is displayed in the derivation format: context $\vdash$ statement, which means the statement is derivable from the context. Recall that, the context is used to collect all statements used for the free variables of a term when typing that term. In this example, only q is free others are all bound.

5.2.2 'Trace' tracer

The tracer 'trace', as its name, trace over the reduction procedure and display every reduction steps, intermediate terms and corresponding type assigned to that term.

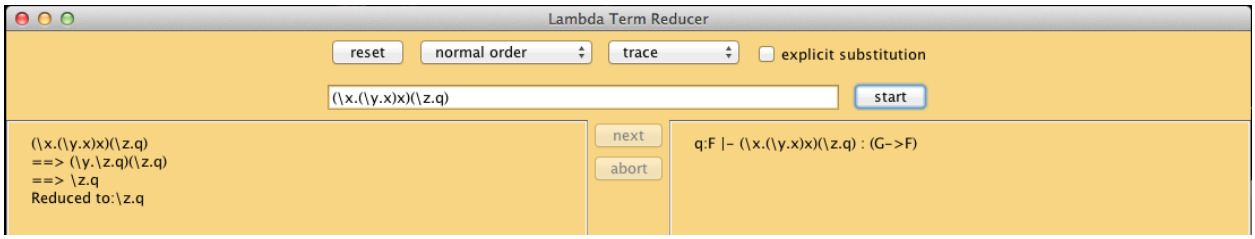


Figure 5.3: Sample run using 'trace' tracer

5.2.3 'Single-step' tracer

The 'single-step' tracer performs β-reduction once per time and user can control whether continue or abort.

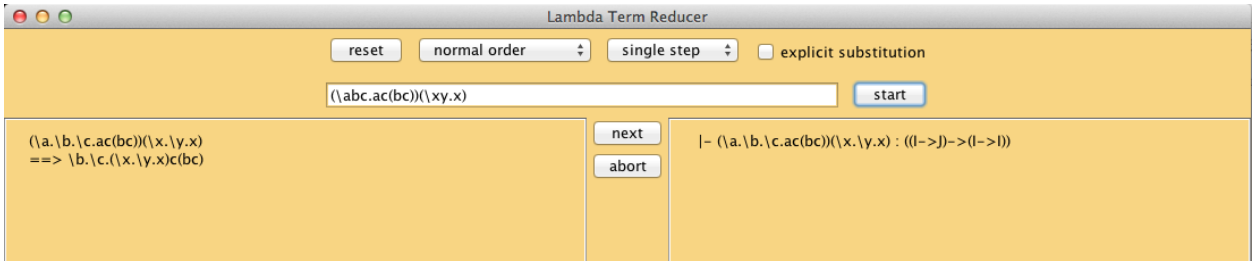


Figure 5.4: Sample run in single-step tracer

As shown in Figure 5.4, after the 'start' button is pressed, a single β-reduction is performed and the corresponding type assignment result is displayed. As we can see, the 'next' button and 'abort' button are enabled now. The user can press the 'next' button to perform another single β-reduction step, or press the 'abort' button to abort the whole procedure.

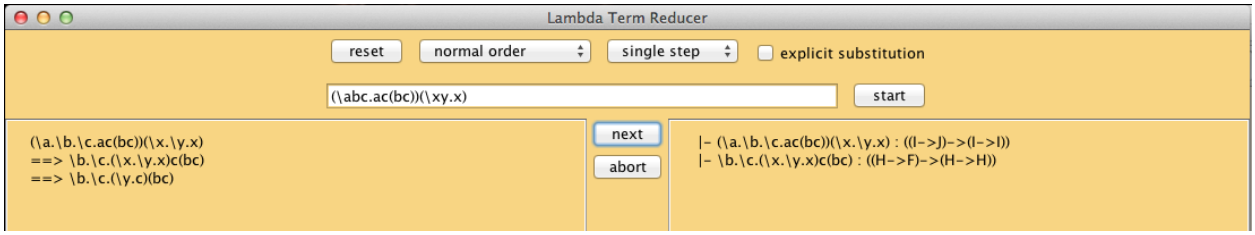


Figure 5.5: Sample run by single-step tracer–second step

Each time the 'next' button is pressed, a single β-reduction would be performed until no reduction can be performed according to the reduction strategy. In Figure 5.5, another β-reduction is performed and the corresponding type assignment result is displayed.

When there is no β-reduction to perform on the λ-term, the reduction procedure stops and both 'next' and 'abort' buttons are disabled.

The 'single-step' tracer in Sestoft's implementation is slightly different from this implementation. As shown in Figure 5.7, Sestoft's reducer highlights the redex in the term and let the user to click on the redex to perform next reduction step. The highlight of redex in the term is useful since it shows which redex the reducer chooses to reduce next. However, it gives less details of the reduction as a complete procedure. The user may not remember the reduction procedure since previous intermediate terms are not displayed. Therefore, in this project, the whole procedure is shown and

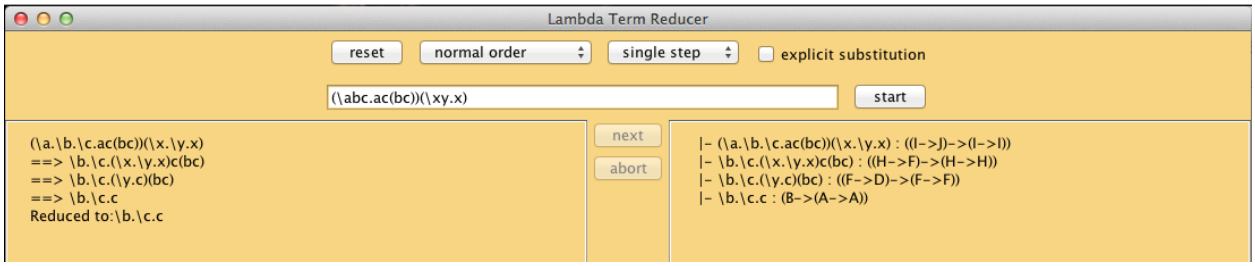


Figure 5.6: Sample run in single-step tracer–final result

each single-step reduction adds a new line in the procedure. The final output of reduction is the same as 'trace', except that each intermediate terms are also typed.

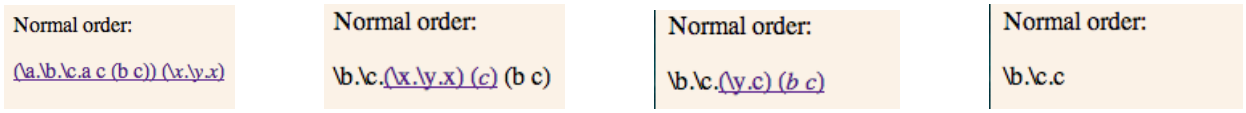


Figure 5.7: Sestoft's lambda term reducer using 'single-step' tracer

5.2.4 Deal with non-terminating terms by trace

In Sestoft's implementation, the engine enters an infinite loop and stuck when the input term is non-terminating and the tracer is 'trace'. There is no intermediate terms or steps displayed, so the user would not know what is happending and what the problem is. In this project, when the input term is non-terminating and the tracer is 'trace', all the intermediate steps would be displayed. As we can see from Figure 5.8, the left output area is a a scroll pane, where the user can scroll down to view the whole procedure. Notice that, the input term `\f.(\x.f(xx))(\x.f(xx))` is untypable since there is a self-application xx, the reason why it cannot be typed could be found in Example 2.1.1.

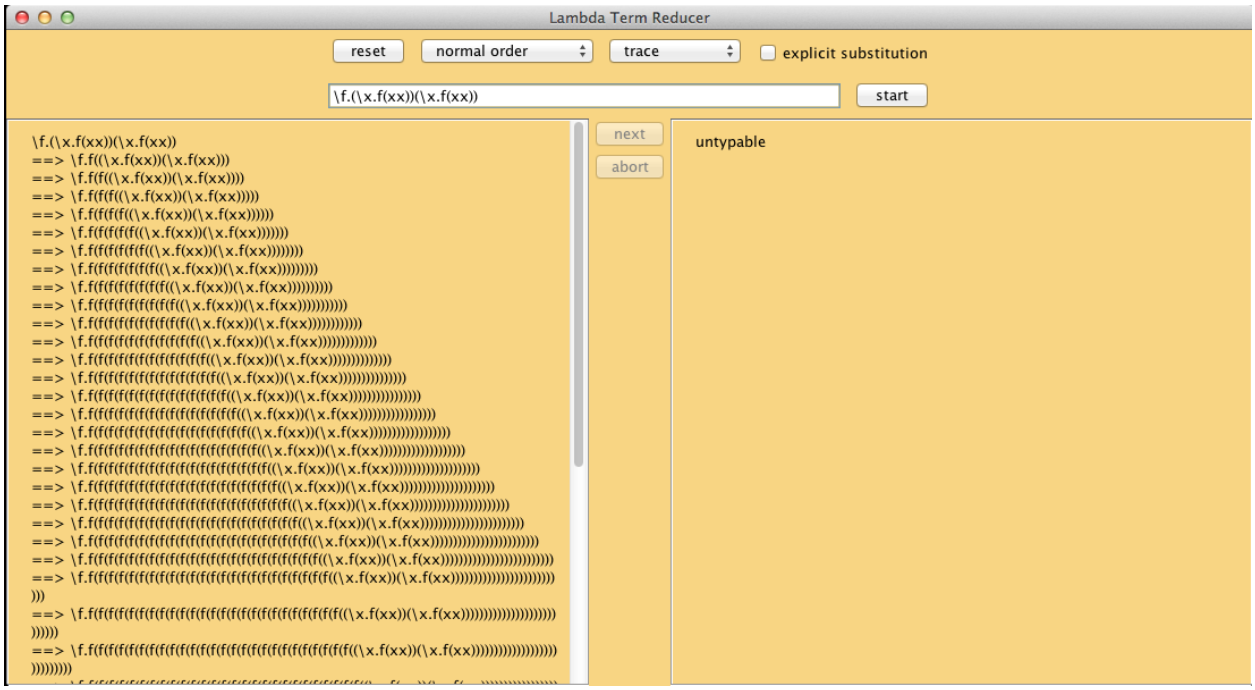


Figure 5.8: Sample run of a non-terminate term by trace

5.2.5 Explicit Substitution

Figure 5.9 illustrates the output when explicit substitution is enabled.

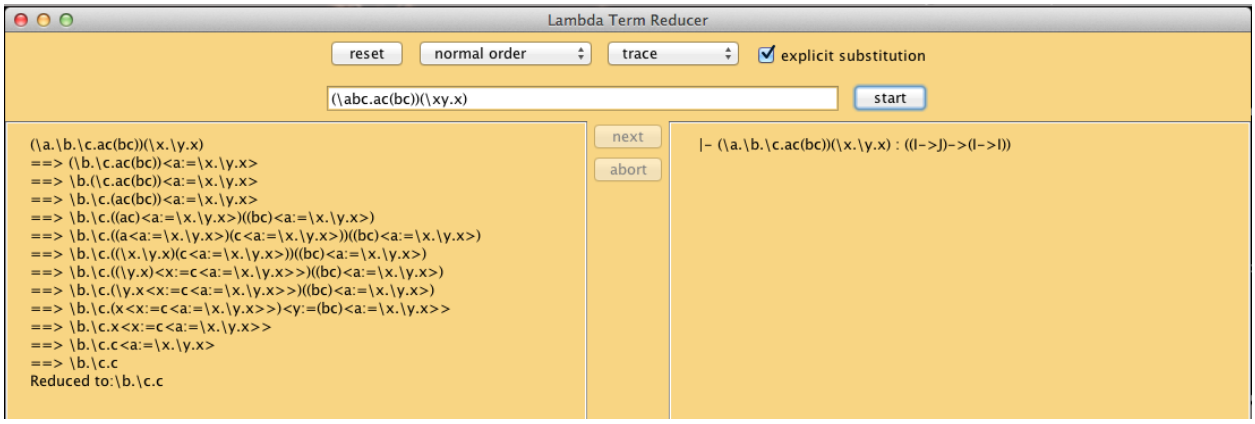


Figure 5.9: Sample run when explicit substitution enabled

Chapter 6

Java vs Haskell, Pros and Cons

6.1 Pros

6.2 Cons

Chapter 7

Conclusion

Bibliography

[1] Hendrik Pieter Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[2] Roel Bloo and Kristoffer H Rose. Preservation of strong normalisation in named lambda calculi with explicit substitution and garbage collection. In *In CSN-95: Computer Science in the Netherlands*. Citeseer, 1995.

[3] Haskell B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[4] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 2. North-Holland Amsterdam, 1972.

[5] Chris Hankin. Lambda calculi: a guide for computer scientists. 1994.

[6] Peter Sestoft. Standard ml on the web server: Visualizing lambda calculus reduction. 1996.

[7] Peter Sestoft. Demonstrating lambda calculus reduction. In *The essence of computation*, pages 420–435. Springer, 2002.

[8] Steffen van Bakel. Type systems for programming language, course notes. 2001.