# Abstract Syntax and Variable Binding

Skevi Anastasiou

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2009

# Abstract Syntax and Variable Binding

Submitted by: Skevi Anastasiou

## COPYRIGHT



## Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

Variable capture creates problems in the flow of programs in many programming languages, as illustrated by the subtle interplay required between applications of alpha and beta reduction in the lambda calculus. In this project we analyse a resolution to this problem proposed by N. De Bruijn, recently refined by use of category theory.

# Contents

# List of Figures

# List of Tables

# Acknowledgements

I would like to thank Dr. John Power for his invaluable support, help and feedback throughout the course.

In addition I would like to thank my family for their support and especially my sister for always being next to me all this time. Thank you.

# Chapter 1

# Introduction

## 1.1 Problem description

An awkward feature of many programming languages is that one must keep checking that variables are named consistently and that one has everything in correct scope.

Every programming language has its own, different concrete syntax, which is the source syntax of the language being compiled. From the perspective of semantics, what we can see from a source code it is just an abstract structure and not the concrete syntax. In concrete syntax, variable binding introduces a binder that defines a variable and the scope, where this variable can be referenced. Then the argument for that scope will be substituted for the specified variable [8].

As an example we can take the lambda expression (λx.x)N. In this expression λ is the binder, 'x' is the bound variable, N is the argument and the scope of x is inside that lambda expression. The meaning of this function is "replace all occurrences of x, with N" and the result of this reduction will be N. If we take another lambda expression (λy.y)N the result will be N again. That shows that the name of the bound variable is not relevant for the substitution that we want to perform. This is because (λx.x)N and (λy.y)N differ only to the name of their bound variables, they are α-equivalent. This will always happen with expressions that are syntactically different but semantically the same.

In order to resolve the above problem a new form of syntax has to be defined that will match syntax with semantics. A mathematically supported syntax that avoids the use of variables is a solution, since what we need is just the position of the variable where the argument should appear.

De Bruijn introduced something similar to this, a way of representing lambda calculus using numbers instead of variables. In de Bruijn's indices the lambda expressions λx.x and λy.y are both represented as λ.1. In this way de Bruijn introduced a new syntax, where each expression is represented by an α-equivalence class.

Later on, Fiore, Plotkin and Turi used sophisticated category theory in order to provide an axiomatic structure to the syntax of lambda calculus, something that was missing from the method that N. G. de Bruijn introduced. Their work was then modified by Tanaka to describe linear binders.

### 1.1.1 Aims and objectives

- Define a new form of syntax that avoids variable capture in lambda expressions.
- Implement a program that will support this new syntax.
- Understand the methods that de Bruijn, Fiore et al and Tanaka proposed
- Learn about linear lambda calculus, proposed by Tanaka and understand why this solves the problem described.

## 1.2 Report structure

Chapter two begins with the literature survey, an introduction to lambda calculus, as it is the main example in the project and then an introduction to basic category theory that was necessary for the understanding of the papers read. Chapter three follows the literature survey with an explanation of de Bruijn's indices and the process followed to convert a name-carrying expression to a name-free expression. Chapter four continues with linear binders as introduced by Tanaka and a discussion about their influence in state. Chapter five will give the documentation and the testing of the program implemented and chapter six will be the conclusion with project and self evaluation.

# Chapter 2

# Literature Survey

## 2.1 Introduction

This project aims to define a new form of syntax that will avoid variable capture in lambda expressions and then implement a program to support it. In order to do that, first I have to study the lambda calculus and understand its basic concepts. I have to begin with the most basic things, like its syntax and the meaning of an expression and then continue with the concept of scope, variable binding, substitution, equivalent expressions, normal forms and reduction of expressions.

This will be the first phase of the survey. It will provide me with the basic knowledge needed to understand and analyse most of the material required for the second and most important phase.

The second phase will be the identification of previous work that has been done in this area. Basic knowledge of category theory is necessary in order to understand a combination of academic journals, published papers, conference proceedings and books, relevant to the problem. This will enable me to read and understand the documents found with more ease. Also, understanding of the papers will help me to identify the major ideas, decide what I would like to use, do in a different or easier way and prevent me from repeating mistakes done in the past.

## 2.2 Lambda calculus

Lambda calculus is a fully expressive language of computable functions that was introduced by Alonzo Church in the 1930s as a way of formalising the concept of effective computability.

### 2.2.1 Syntax

Lambda calculus has a very simple syntax given by the following grammar:

<expression> := <variable>|<function>|<application>
<function> := λ<variable>.<expression>
<application> := <expression><expression>

According to this grammar an example of a function is λx.M, where x is the parameter of the function and M is its body. When this function is applied to an argument, M will be evaluated with x bound to the argument. An application example can be the expression MN where it is the application of a function M to an argument N.

Usually, expressions in lambda calculus are enclosed in parenthesis. Assuming that A is an expression, then (A) is the same expression. In order to be more efficient we say that function application associates from the left. By this we mean that if we have an expression

$$A_1\ A_2\ A_3\ A_4\ \ldots.\ A_n$$

it will be evaluated as

$$(\ldots.\ (((A_1\ A_2)\ A_3)\ A_4)\ \ldots.\ A_n)$$

### 2.2.2 Scope and variable binding

The scope of a lambda expression λx.N can be defined as the expression N and it extends as far to the right as possible. For example the expression λx.xy means λx.(xy) and not (λx.x)y.

A variable in a lambda expression can appear as *free* or *bound*. It is said to be bound if the *binding operator* λ binds it within a specific scope, otherwise it is free.

If we take the expression λx.(λy.xy)y we see that, reading from left to right the second y occurrence is bounded by a binding operator but the third one appears to be free, since it is outside the scope of the λy.

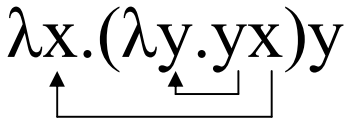



*Figure1: Free and bound occurrences of a lambda expression.*

Working with expressions that use the same variable names for a free and a bound variable is confusing, so it is preferable to state expressions with bound variables to have different names from the free ones. But, if we get an expression where free variables appear to be bound too we can rename the bound variables. This process is known as *α-conversion* (alpha conversion).

For example, if we α-convert the expression (λy.xy)y we will have to rename the bound variable y to w and the resultant expression will be (λw.xw)y. These two expressions now are the same up to their bound variables, so we say that they are *α-equivalent* (alpha equivalent).

## 2.2.3 Judgements and rules

Free and bound variables can be defined in a more precise way, using judgements and rules. A judgement is an inference of the form

$$x_1,\ldots..x_n, x_{n+1} \vdash t$$

and can be read as the implication "if $x_1,\ldots..x_n, x_{n+1} \vdash t$ then the free variables of t, is the set $\{ x_1,\ldots,x_n, x_{n+1}\}$". A valid judgement is derived form a set of inference rules. The above example was derived from the rule

$$\frac{x_1,\ldots..x_n, x_{n+1} \vdash t}{x_1,\ldots..x_n \vdash \lambda x_{n+1}.t}$$

Which means if $x_1,\ldots..x_n, x_{n+1}$ are the free variables of an expression t then $x_1,\ldots..x_n$ are the free variables of the function $\lambda x_{n+1}.t$. In a rule, if the hypothesis (numerator) is a valid judgement then the conclusion (denominator) is a valid judgement too.

Free variables are defined by the following rules [7]

$$\frac{1 \le i \le n}{x_1,\ldots..x_n \vdash x_i} \qquad \frac{x_1,\ldots..x_n, x_{n+1} \vdash t}{x_1,\ldots..x_n \vdash \lambda x_{n+1}.t} \qquad \frac{x_1,\ldots..x_n \vdash t_1 \quad x_1,\ldots..x_n \vdash t_2}{x_1,\ldots..x_n \vdash t_1 t_2}$$

## 2.2.4 Computation in lambda calculus

Computation in lambda calculus is given by transformation of expressions. A computational step is a transformation which manipulates an expression and yields a new one.

*B-reduction* (beta reduction) is a basic transformation that reduces an expression M to a new expression N.

$$M \xrightarrow[\beta]{} \mathrm{N}$$

B-reduction evaluates function applications based in substitution. For example, if we have the function λx.M applied to an argument N, after a β-reduction step the new expression will be M with all occurrences of x replaced by N. This can be shown as

$$(\lambda x.M)N \xrightarrow[\beta]{} M[N/x]$$

where $(\lambda x.M)N$ is a β-redex. B-reduction can be performed wherever we find a β-redex. β-reduction can be defined using a set of rules as follows

$$\frac{}{(\lambda x.M)N \xrightarrow[\beta]{} M[N/x]}$$

$$\frac{M \xrightarrow[\beta]{} \mathrm{N}}{MP \xrightarrow[\beta]{} \mathrm{N}P} \qquad \frac{M \xrightarrow[\beta]{} \mathrm{N}}{PM \xrightarrow[\beta]{} P\mathrm{N}} \qquad \frac{M \xrightarrow[\beta]{} \mathrm{N}}{\lambda x.M \xrightarrow[\beta]{} \lambda x.\mathrm{N}}$$

The first rule it is called an *axiom* because it does not have a hypothesis and the conclusion is always valid.

Some example reductions are the following:

$$\begin{aligned} (\lambda x.x)M &\xrightarrow{\beta} M \\ (\lambda x.xyz)k &\xrightarrow{\beta} kyz \\ (\lambda x.x)((\lambda y.y)(\lambda z.z)) &\xrightarrow{\beta} (\lambda y.y)(\lambda z.z) \xrightarrow{\beta} \lambda z.z \end{aligned}$$

In the last expression we performed two reduction steps, since after the first one there was still another β-redex in the expression. When it was reduced to λz.z no further reduction could be performed, so we say that the expression is in *normal form*; we don't have any β-redex to reduce.

Another approach to the reduction of that expression would be to choose to reduce the second redex in the expression first. This leads to the following expression,

$$(\lambda x.x)((\lambda y.y)(\lambda z.z)) \xrightarrow{\beta} (\lambda x.x)(\lambda z.z) \xrightarrow{\beta} \lambda z.z$$

where the normal form reached is the same as before, despite the fact that we used a different reduction order. This proves that an expression has at most one normal form.

In some cases, before performing β-reduction we have to perform α-conversion first. This happens if a variable appears to be free in the second expression and bound in the first one. If we consider the expression

$$(\lambda x.\lambda y.x)(\lambda z.yz)$$

y is free in λz.yz and bound in λx.λy.x. If we perform a β-reduction step we get the expression

$$\lambda y.(\lambda z.yz)$$

where y was captured by λy and it appears to be bound. This can not happen; free variables have to remain free after every reduction step. In order to prevent this we have to perform α-conversion first, as described above.

If we rename the bound y to n we get the following:

$$(\lambda x.\lambda n.x)(\lambda z.yz)$$

After a β-reduction step we get the expression

$$\lambda n.(\lambda z.yz)$$

where y appears free as before and the expression is in normal form.

# 2.3 Category theory

## 2.3.1 Introduction

Category theory is a branch of abstract algebra with many applications in computer science, logic and mathematics. It provides a basic conceptual mechanism and a set of formal methods useful for addressing certain kinds of commonly occurring problems, mostly those involving structural and functional considerations.

## 2.3.2 Basic concepts and definitions

### 2.3.2.1 Categories

A category consists of a pair of structures together with a collection of relations between them. Structures are often called the objects of the category and the relations between the structures are called morphisms.

**Definition 1.1:**

A *category C* is specified by the following data [12]:

- A collection obC of objects, usually denoted by capital letters of the alphabet.
- A collection morC of morphisms, usually denoted by lower case letters of the alphabet.
- Two operations assigning to each morphism f its *source,* src(f) which is an object of C and its *target,* tar(f) also an object of C. We shall write $f : A \longrightarrow B$ to indicate this, where A = src(f) and B = tar(f).
- Morphisms $f : A \longrightarrow B$ and $g : B \longrightarrow C$ are *composable* if tar(f) = src(g). There is an operation assigning to each pair of composable morphisms f and g their

*composition* which is a morphism denoted by gf or $g \circ f : A \longrightarrow C$ such that $src(g \circ f) = src(f)$ and $tar(g \circ f) = tar(g)$. There is also an operation assigning to each object A of C an *identity* morphism $id_A$: $A \longrightarrow A$. These operations are required to be *unitary*

$$id_{tar(f)} \circ f = f$$

$$f \circ id_{src(f)} = f$$

and *associative,* that is given morphisms $f : A \longrightarrow B$, $g : B \longrightarrow C$ and $h : C \longrightarrow D$ then

$$(hg)f = h(gf)$$

A morphism $f : A \longrightarrow B$ in a category C is said to be an *isomorphism* if there are some $g : B \longrightarrow A$ for which $fg = id_B$ and $gf = id_A$ [12]. Then we can say that f is an inverse of g and also g is an inverse for f. If such a mutually inverse pair of isomorphism exists we say that an object A in category C is isomorphic to an object B in the category C and we write $A \cong B$.

## 2.3.2.2 Functors

If we consider a category in which the objects are categories and the morphisms are mappings between categories, then these morphisms are called *functors*. Roughly, a functor from a category A to a category B is an assignment which sends each object of A to an object of B, and each morphism of A to a morphism of B.

An exact definition given by Roy L. Crole in his book “Categories for types” is as follows:

**Definition 1.2:**

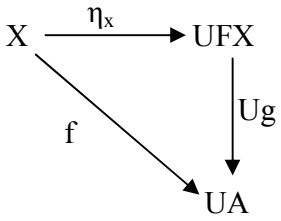
A functor F between categories C and D, written as $F: C \longrightarrow D$ is specified by

- An operation taking objects A in C to objects FA in D
- An operation sending morphisms $f : A \longrightarrow B$ in C to morphisms $Ff : FA \longrightarrow FB$ in D,

for which $F(id_A) = id_{FA}$, and whenever the composition of morphisms gf is defined in C we have $F(gf) = Fg \circ Ff$. Note that $Fg \circ Ff$ is defined in D whenever gf is defined in C, that is, Ff and Fg are composable in D whenever f and g are composable in C.

**Definition 1.3:**

A functor $U : C \longrightarrow D$ has a *left adjoint* if for all $X \in D$, there exist $FX \in C$ and $\eta_x : X \longrightarrow UFX$ in D such that for all A in C and for all $f : X \longrightarrow UA$, there exists a unique map $g : FX \longrightarrow A$ such that the diagram




commutes [6].

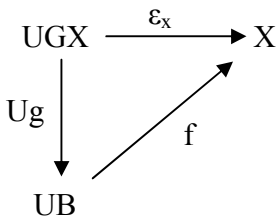
Given a category C, we can define the opposite category $C^{op}$ as follows:

- $obC^{op} = obC$
- $C^{op}(X, Y) = C(Y, X)$
- Composition and units of $C^{op}$ are given by those of C.

Spelling this out, we can now define the concept of right adjoint, as given below:

**Definition 1.4:**

A functor $U : C \longrightarrow D$ has a *right adjoint* if for all $X \in D$, there exist $GX \in C$ and $\varepsilon_x : UGX \longrightarrow X$ in D such that for all B in C and for all $f : UB \longrightarrow X$, there exists a unique map $g : B \longrightarrow GX$ such that the diagram



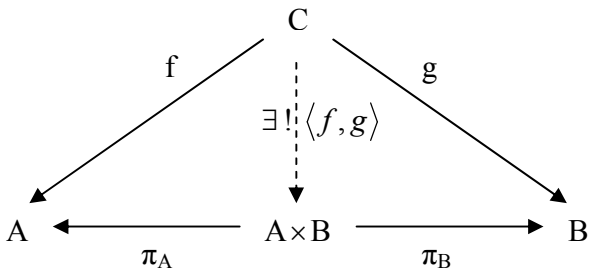

commutes.

### 2.3.2.3 Products and coproducts

As Roy L. Crole defined in his book "Categories for types" a *binary product* of objects A and B in a category C is specified by

- An object $A \times B$ of C
- Two projection morphisms $\pi_A : A \times B \longrightarrow A$ and $\pi_B : A \times B \longrightarrow B$

For which given any object C and morphisms $f : C \longrightarrow A$, $g : C \longrightarrow B$ there is a unique morphism $\langle f, g \rangle : C \longrightarrow A \times B$ for which $\pi_A \langle f, g \rangle = f$ and $\pi_B \langle f, g \rangle = g$.




A category C has binary products if the diagonal functor

$$\Delta : C \longrightarrow C \times C$$

has a right adjoint.

**Definition 1.5:**

A *terminal object* in a category C is an object T of C such that, for every object X of C, there is a unique map $!_X : X \longrightarrow T$ [6].

In the case where a category C has a terminal object and all binary products we can say that the category has all *finite products*.

A dual notion of product is *coproduct*. That means that the binary products of a pair of objects of the opposite category $C^{op}$ can give the *binary coproducts* of the category C.

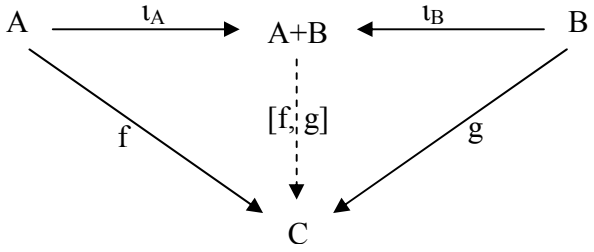
An exact definition given by Roy L. Crole in his book "Categories for types" is given below:

**Definition 1.6:**

A *binary coproduct* of objects A and B in a category C is specified by

- An object $A + B$ of C, together with
- Two insertion morphisms $\iota_A : A \longrightarrow A + B$ and $\iota_B : B \longrightarrow A + B$,

Such that for each pair of and morphisms $f : A \longrightarrow C$, $g : B \longrightarrow C$ there exists a unique morphism $[f, g] : A + B \longrightarrow C$ for which $[f, g]\iota_A = f$ and $[f, g]\iota_B = g$.
This definition can be pictured with the following commutative diagram.




## 2.3.2.4 Cartesian closed categories

A category C is said to be *cartesian closed category* if it has finite products, such that for all objects Y of C, the functor $- \times Y : C \to C$ has a right adjoint, denoted by $(-)^Y : C \to C$. This indicates that for all objects S, T and U of C an isomorphism

$$(*) \quad \mathrm{Hom}_C(S \times T, U) \approx \mathrm{Hom}_C(S, U^T)$$

exists and it is natural to S, T and U.

As an example we can take the category of *Sets*. In this category $S \times T$ is the cartesian product of sets and $U^T$ is the set of all functions $T \longrightarrow U$. The function $g : S \times T \longrightarrow U$ is sent to the function $g^* : S \longrightarrow U^T$, where $g^*(s)(t) = g(s, t)$ for all $s \in S$ and $t \in T$ by the bijection (*).

### 2.3.2.5 Presheaf category

The presheaf category was used on the free strict monoidal category on 1 by Miki Tanaka in her paper "Abstract Syntax and variable binding", in order to model binding signatures.

In order to explain what the presheaf category is we have to give the definition of the contravariant functor as described by Mac Lane and Saunders in their book "Categories for the working mathematician". The definition is as follows:

**Definition 1.7:**

Consider a functor $S : C^{op} \longrightarrow B$. By the definition of a functor, it assigns to each object $c \in C^{op}$ an object Sc of B and to each arrow $f^{op} : b \longrightarrow a$ of $C^{op}$ an arrow $Sf^{op} : Sb \longrightarrow Sa$ of B with $S(f^{op}g^{op}) = (Sf^{op})(Sg^{op})$ whenever $f^{op}g^{op}$ is defined. The functor S so described may be expressed directly in terms of the original category C if we write $\bar{S}\, f$ for $Sf^{op}$; then $\bar{S}$ is a contavariant functor on C to B, which assigns to each object $c \in C$ an object $\bar{S}\, c \in B$ and to each arrow $f : a \longrightarrow b$ an arrow $\bar{S}\, f : \bar{S}\, a \longrightarrow \bar{S}\, b$ (in the opposite direction), all in such a way that

$$\bar{S}\,(1_c) = 1_{(\bar{S}\, c)} \qquad \bar{S}\,(f\, g) = (\bar{S}\, g)\,(\bar{S}\, f)$$

Now, we are able to define the presheaf category as follows.

**Definition 1.8:**

The *presheaf category* is the contravariant functor $F : C^{op} \longrightarrow$ Sets where C is a small category.

# 2.4 Summary

In this section the basic lambda calculus and category theory notions needed for the development of the new form of syntax were introduced.

An example of work done before is “Abstract syntax and variable binding for linear binders” which constructs a category of models for a linear binding signature. This will be discussed in a later chapter.

Another important work done before is the de Bruijn indices, which is a form of syntax developed from N.G de Bruijn in order to get rid of variables in lambda expressions. A description of this method is given in the next chapter.

# Chapter 3

# De Bruijn Indices

As it was shown in the previous chapter, the need for renaming bound variables in order to prevent variable capture can make manipulations in the lambda calculus burdensome. Nicolaas Govert de Bruijn thought that it would be useful to try to get rid of variables and as a consequence, variable renaming and produce a syntax that will use integers to show the "distance to the binding lambda instead of a name attached to that lambda" [10].

This new syntax will be:

- Easy to read and write for the human reader
- Easy to handle in met lingual discussion
- Easy for the computer and the programmer [10].

## 3.1 Name-free expressions

In order to explain how to convert a variable-carrying expression to a name-free expression we will take the following expression from his paper [10].

$$\lambda x.a(\lambda t.b(x,t,f(\lambda u.a(u,t,z),\lambda s.w)),w,y)$$

In the above expression the set of letters {a, b, c…, r} are the *constants*, the set {s, t, u…, z} are the *variables* and the Greek letter $\lambda$ is a binder that can have any variable as a suffix.

By changing the names of the bound variables x, t, u, s to another variable names, that do not appear free in the expression, the expression generated will be α-equivalent to this expression. N. G. de Bruijn defined α-equivalent expressions to be the same and they belong to α-equivalent classes.

## Process:

**1.** Represent the expression as a tree so that it will be easier to read.

**2.** List the letters from which free variables will be taken.
Free variables: z, w, y, v.
The set of free variables has to have more or the same number of variables than the set of bound variables. It has to contain the variables z, y, w in any order and additional variables needed so that it will be long enough.

**3.** Draw λz, λv, λw, λy under the tree.

**4.** Label every variable with two numbers
- *Reference depth:* the number of $\lambda$ counted when going down from a variable 'a' until $\lambda a$ .
- *Level:* the number of $\lambda$ counted when going down from a variable until the root of the tree. The root of the tree is just before the addition of $\lambda z, \lambda v, \lambda w, \lambda y$ .

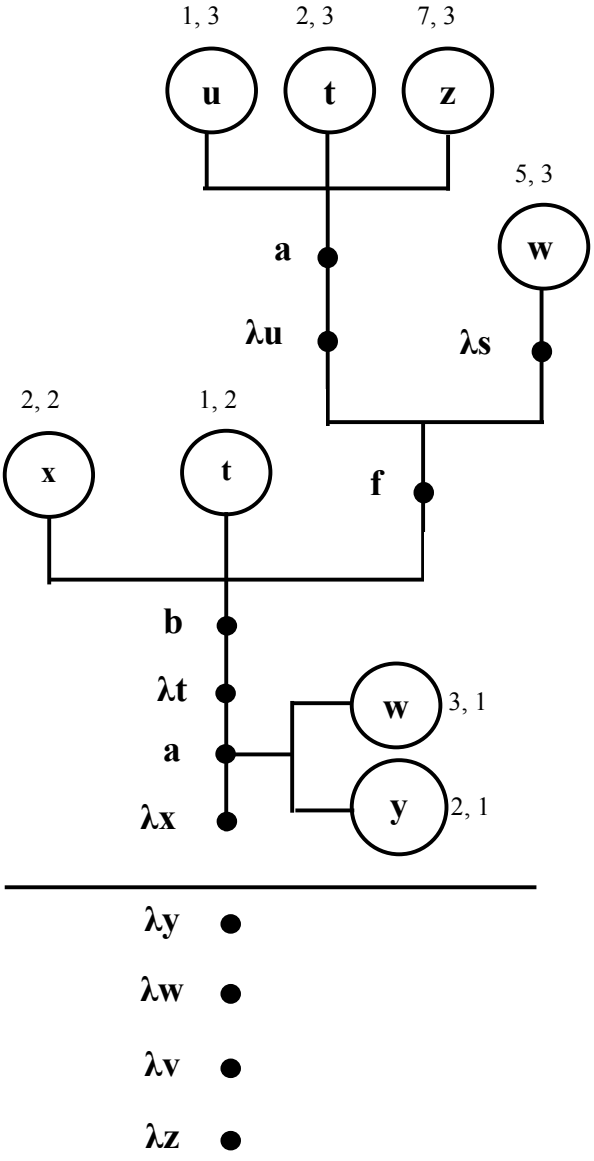



*Figure 2: Tree generated from the expression in the example above showing the relation between the variables and also indicating the depth and the level.*

Now that we constructed the tree we can easily construct a name-free expression of the name-carrying expression used. This can be done by erasing the variable names and the number indicating the level; we will keep the depth. If the reference depth of a variable in the tree exceeds the level then that variable is free.

The name-free expression will be as follows:

$$\lambda a(\lambda b(2,1, f(\lambda a(1,2,7), \lambda 5)),3,2)$$

with the list of free variables z, v, w, y in that order. This list is vital in the case where we want to construct the name-carrying expression.

Simpler expressions, where the scope of every variable is obvious can be represented in de Bruijn's indices without following the process described above. Some examples are:

$$\begin{aligned} \lambda x.x &\longrightarrow \lambda.1 \\ \lambda y.\lambda x.y &\longrightarrow \lambda.\lambda.2 \end{aligned}$$

## 3.2 De Bruijn indices critique

Although N.G. de Bruijn tried to develop that new form of syntax so that it would be easier for the human reader to read and write it, this method is not claimed to be good for this purpose. As we have seen in section 3.1, reading an expression in de Bruijn's indices is clear and well understood but on the other hand, the process to write a long and complicated expression in this form of syntax takes time.

# Chapter 4

# Linear Lambda Calculus

## 4.1 Introduction

De Bruijn's indices have proved to be clear and comprehensible, but his syntax is complicated and lacks the kind of axiomatic mathematical support that the lambda calculus has. So Fiore, Plotkin and Turi used sophisticated category theory, with the untyped lambda calculus as their leading example, to provide such axiomatic structure. Their work was modified by Tanaka to account for linear binders, using the presheaf category on the *free strict symmetric monoidal category P, on 1* instead of F that Fiore et al used. This work can be found in her paper "Abstract syntax for variable binding". Her work has recently been modified to account for mixtures of binders, as appear for instance in the Logic of Bunched Implications.

## 4.2 Linear binding

As we have discussed in Chapter 2 (section 2.2.2) a variable in a lambda expression can appear as bound, if the binding operator λ binds it within a specific scope, or free otherwise. Here, by the term *linear binding* we mean the variable binding where:

- No variable can occur free in an expression more than once.
- A binder cannot bind a variable that does not appear in the expression [8].

These conditions can be met by introducing well-formedness rules with special contexts on the usual, non-linear expressions.

**Definition 4.1:**

A *context* E, in untyped lambda calculus, is a finite list of distinct variables, usually written as

$$E : x_1, x_2, \ldots, x_n$$

In our case what we are interested in are linear contexts.

**Definition 4.2:**

A *linear context* is a sequence of variables, with no variables occurring twice [8].

The definition of usual context (definition 4.1) and linear context (definition 4.2) might seem the same, but what actually make them two different definitions are the operations on linear contexts which differ from those on usual ones. These operations are described as follows: Consider two linear contexts S and T with m and n variables respectively and with no variables in common.

$$S \cap T = \varnothing$$

A merge of them will produce a new linear context, written as S # T which will be a sequence of all the variables included in S and T in arbitrary order. Since the variables can be in arbitrary order we can have (m + n)! possible merges of these contexts.

Linear contexts can be used to form linear lambda expressions. This can be done if the free variables of the lambda expression are strictly controlled by the context. Every variable in the context must appear as a free variable in the lambda expression precisely once.

Assuming that S and T are linear contexts, we can obtain the expressions of untyped linear lambda calculus by imposing the two conditions for linear binding, described above on the non-linear expressions.

$$x \vdash x \qquad \frac{S, x \vdash t}{S \vdash \lambda x.t} \qquad \frac{S \vdash t_1 \quad T \vdash t_2}{S \# T \vdash t_1 t_2}$$

## 4.3 State

Linear binding and linear variables are very important when we come to the point of programming.

In a functional language, where no assignments are made, variables can be copied or deleted at will, as they are not stored anywhere. On the other hand, in an imperative language where assignments in variables are performed, it is not sensible to try to copy or delete state. From that we can see that variables for functions are not linear, but the state is.

For many years, scientists were discussing the linearity of state, but with no use of the word linear. As C. Strachey wrote, in his paper "The varieties of programming language", *"The state transformation produced by obeying a command is essentially irreversible and it is, by the nature of the computers we use, impossible to have more than one version of [the state] available at any one time,"*.

## 4.4 A better solution

Linear binders are a possible solution to our problem, described in chapter 1 (section 1.1). As every variable of the context given can appear in the linear expression as a free variable exactly once and a binder cannot bind a variable that does not appear in the expression there is no need to worry about variable capture. All variables in a linear expression are distinct with each other.

This method of avoiding variable capture is much easier than the method proposed by N. G. de Bruijn, and discussed in chapter 3. That method was found to be complicated and time consuming for the human reader, in contrast with linear binders which just involve checking for variable uniqueness.

# Chapter 5

# Implementation and Testing

## 5.1 Introduction

One of the aims of this project was to implement a program that will support a new form of syntax which will avoid variable capture. As discussed in chapter 4, such a solution to our problem is linear lambda calculus, which I have tried to implement by first implementing the untyped lambda calculus.

## 5.2 Chosen language

After a quick research in programming languages available for lambda calculus programming I decided to use Lisp. Lisp is one of the oldest high level programming languages and also it was originally created based on lambda calculus In addition to these, the fact that I was already familiar with the language made my decision easier.

## 5.3 Testing technique

Implementation and testing was performed in parallel, as each function in Lisp can be tested individually. I have chosen to use Black-Box testing which treats the software as a black-box without any understanding of internal behaviour and aims to test the functionality according to the requirements. Thus, I had to input data and only see the

output from the test object. This level of testing requires careful test cases that will enable me to verify that for a given input, the output value is the expected value.

## 5.4 Program documentation and testing

As discussed in chapter 2 (section 2.2.1), a lambda expression can be defined as a variable 'x', an abstraction 'λx.M' or an application MN where M and N can be any lambda expressions.

In Lisp we can define

- (VAR *x*) to represent variables with *x* is as the variable's symbol.
- (LAMBDA *x expression*) to represent an abstraction with *x* as the abstraction's symbol and *expression* as a representation of M.
- (APP *expression1 expression2*) to represent an application where *expression1* and *expression2* are the representations of M and N respectively.

As a start of the implementation I implemented functions for the identification of a valid variable, abstraction and application. Also functions for returning information about an expression and functions for the creation of an expression. Then I implemented alpha conversion, substitution, beta reduction and reduction to normal form. In addition to the above in order to implement the linear lambda calculus I had to implement a function to check valid contexts, a function to check for the linearity of an expression and also a function to reduce the linear expressions to normal form.

All the implementation for each function can be found in Appendix A.

### 5.4.1 Function isVar

This function has to identify valid variables. A valid variable is of the form (VAR *x*) where *x* is a symbol. Everything else has to return a false output.

| Input | Output | Pass/Fail |
|---|---|---|
| (isVar '(VAR x)) | #t | Pass |
| (isVar '(VA x)) | ( ) | Pass |

*Table 1: Testing for isVar function*

## 5.4.2 Function isAbs

This function has to identify valid abstractions. A valid abstraction is of the form (LAMBDA *x expression*) where *x* is a symbol and *expression* is a valid variable, application or another abstraction. Everything else has to return a false output.

| Input | Output | Pass/Fail |
|---|---|---|
| (isAbs '(LAMBDA x (VAR y))) | #t | Pass |
| (isAbs '(LAMDA x (VAR x))) | ( ) | Pass |
| (isAbs '(LAMBDA x (LAMBDA x (VAR y)))) | #t | Pass |
| (isAbs '(LAMBDA x (APP (VAR y) (VAR x)))) | #t | Pass |
| (isAbs '(VAR y)) | ( ) | Pass |
| (isAbs '(APP (VAR x) (VAR y))) | ( ) | Pass |

*Table 2: Testing for isAbs function*

## 5.4.3 Function isApp

This function has to identify valid applications. A valid application is of the form (APP *expression1 expression2*) where *expression1* and *expression2* can be any valid variable, abstraction or application. Everything else has to return a false output.

| Input | Output | Pass/Fail |
|---|---|---|
| (isApp '(APP (VAR x) (VAR y))) | #t | Pass |
| (isApp '(VAR x)) | ( ) | Pass |
| (isApp '(APP (LAMBDA x (VAR y)) (VAR x))) | #t | Pass |
| (isApp '(APP (APP (VAR y) (VAR x)) (LAMBDA x (VAR y)))) | #t | Pass |
| (isApp '(LAMBDA x (VAR y))) | ( ) | Pass |
| (isApp '(VAR y)) | ( ) | Pass |

*Table 3: Testing for isApp function*

## 5.4.4 Function return-symb

This function has to return the symbol of a variable. If the expression given is not a valid variable it has to return false.

| Input | Output | Pass/Fail |
|---|---|---|
| (return-symb '(VAR x)) | x | Pass |
| (return-symb '(LAMBDA x (VAR y))) | ( ) | Pass |
| (return-symb '(VAR k)) | k | Pass |
| (return-symb '(APP (VAR x) (VAR y))) | ( ) | Pass |

*Table 4: Testing for return-symb function*

### 5.4.5 Function return-absVar

This function has to return the variable of an abstraction. If the expression given is not a valid abstraction it has to return false.

| Input | Output | Pass/Fail |
|---|---|---|
| (return-absVar '(LAMBDA x (VAR y))) | x | Pass |
| (return-absVar '(APP (VAR x) (VAR y))) | ( ) | Pass |
| (return-absVar '(LAMBDA y (LAMBDA z (VAR z)))) | y | Pass |
| (return-absVar '(LAMBDA k (APP (VAR y) (VAR x)))) | k | Pass |
| (return-absVar '(VAR x)) | ( ) | Pass |

*Table 5: Testing for return-absVar function*

### 5.4.6 Function return-absBody

This function has to return the body of an abstraction. If the expression given is not a valid abstraction it has to return false.

| Input | Output | Pass/Fail |
|---|---|---|
| (return-absBody '(APP (VAR x) (VAR y))) | ( ) | Pass |
| (return-absBody '(LAMBDA x (VAR y))) | (VAR y) | Pass |
| (return-absBody '(VAR y)) | ( ) | Pass |

| Input |
|---|
| (return-absBody '(LAMBDA z (LAMBDA x (VAR x)))) |
| **Output** |
| (LAMBDA x (VAR x)) |

| Pass/Fail |
| --- |
| Pass |

| Input |
| --- |
| (return-absBody '(LAMBDA z (APP (VAR y) (VAR x)))) |
| **Output** |
| (APP (VAR y) (VAR x)) |
| **Pass/Fail** |
| Pass |

*Table 6: Testing for return-absBody function*

## 5.4.7 Function return-appFun

This function has to return the function of an application. If the expression given is not a valid application it has to return false.

| Input | Output | Pass/Fail |
| --- | --- | --- |
| (return-appFun '(APP (VAR y) (VAR x))) | (VAR y) | Pass |
| (return-appFun '(VAR x)) | ( ) | Pass |
| (return-appFun '(APP (VAR x) (LAMBDA x (VAR y)))) | (VAR x) | Pass |

| Input |
| --- |
| (return-appFun '(APP (LAMBDA x (APP (VAR x) (VAR k))) (VAR z))) |
| **Output** |
| (LAMBDA x (APP (VAR x) (VAR k))) |
| **Pass/Fail** |
| Pass |

| Input |
| --- |
| (return-appFun '(APP (APP (APP (LAMBDA x (VAR y)) (VAR k)) (VAR z)) (VAR x))) |
| **Output** |
| (APP (APP (LAMBDA x (VAR y)) (VAR k)) (VAR z)) |
| **Pass/Fail** |
| Pass |
| |
| *Table 7: Testing for return-appFun function* |

## 5.4.8 Function return-appArg

This function has to return the argument of an application. If the expression given is not a valid application it has to return false.

| Input | Output | Pass/Fail |
|---|---|---|
| (return-appArg '(APP (VAR y) (VAR x))) | (VAR x) | Pass |
| (return-appArg '(LAMBDA x (VAR y))) | ( ) | Pass |

| Input |
|---|
| (return-appArg '(APP (VAR k) (LAMBDA x (VAR x)))) |
| **Output** |
| (LAMBDA x (VAR x)) |
| **Pass/Fail** |
| Pass |

| Input |
|---|
| (return-appArg '(APP (APP (VAR x) (VAR y)) (APP (VAR z) (VAR k)))) |
| **Output** |
| (APP (VAR z) (VAR k)) |
| **Pass/Fail** |
| Pass |

*Table 8: Testing for return-appArg function*

## 5.4.9 Function create-var

This function has to create a new variable with variable symbol the symbol given by the user.

| Input | Output | Pass/Fail |
|---|---|---|
| (create-var 'x) | (VAR x) | Pass |
| (create-var '1) | ( ) | Pass |

*Table 9: Testing for create-var function*

## 5.4.10 Function create-abs

This function has to create a new abstraction with abstraction symbol and abstraction body the symbol and expression given by the user respectively.

| **Input** |
|---|
| (create-abs 'x '(LAMBDA x (VAR y))) |
| **Output** |
| (LAMBDA x (LAMBDA x (VAR y))) |
| **Pass/Fail** |
| Pass |

| **Input** |
|---|
| (create-abs 'y '(APP (VAR x) (VAR y))) |
| **Output** |
| (LAMBDA y (APP (VAR x) (VAR y))) |
| **Pass/Fail** |
| Pass |

*Table 10: Testing for create-abs function*

## 5.4.11 Function create-app

This function has to create a new application with an application and an argument given by the user.

| **Input** | **Output** | **Pass/Fail** |
|---|---|---|
| (create-app '(VAR x) '(VAR y)) | (APP (VAR x) (VAR y)) | Pass |

| **Input** |
|---|
| (create-app '(LAMBDA x (APP (VAR z) (VAR y))) '(APP (VAR k) (VAR w))) |
| **Output** |
| (APP (LAMBDA x (APP (VAR z) (VAR y))) (APP (VAR k) (VAR w))) |
| **Pass/Fail** |
| Pass |

*Table 11: Testing for create-app function*

## 5.4.12 Function alpha

This function has to perform alpha conversion to lambda expressions. Given a lambda expression, a symbol x and a symbol y, it has to replace all free occurrences of x inside the lambda expression with y.
(alpha *expression* x y) should return an expression computed by replacing all free occurrences of the variable (VAR x) in *expression* by (VAR y).

| Input | Output | Pass/Fail |
|---|---|---|
| (alpha '(LAMBDA x (VAR y)) 'y 'z) | (LAMBDA x (VAR z)) | Pass |
| (alpha '(VAR x) 'x 'y) | (VAR y) | Pass |
| (alpha '(VAR x) 'y 'z) | (VAR x) | Pass |
| (alpha '(APP (VAR u) (VAR u)) 'u 'x) | (APP (VAR x) (VAR x)) | Pass |

| Input |
|---|
| (alpha '(APP (VAR x) (LAMBDA x (VAR x))) 'x 'z) |
| **Output** |
| (APP (VAR z) (LAMBDA x (VAR x))) |
| **Pass/Fail** |
| Pass |
| |

*Table 12: Testing for alpha function*

## 5.4.13 Function subs

This fuction has to perform substitution. (subs M x N) should compute the expression M[N/x]. In order to avoid variable capture of free variables of N when performing substitution the Lisp's function *gensym* was used.
After the first test of the function I noticed that in the recursive step of the function the same gensym number was generated so all variables were the same. In order to fix this problem the helper function subsHelpFun(M x N variable) was implemeted.

This function does the substitution, after the actual subs function checks that M, N and x are valid.

The subs function is the following:

```
(defun subs (M x N)
  (if (and (or (isAbs M) (isApp M) (isVar M)) (or (isAbs N)
    (isApp N) (isVar N)) (isVar x))
      (subsHelpFun M x N (gensym))
      '()
  )
)
```

The new variable that was added to the helper function is a gensym number which is passed to it from the actual subs function. Now, every time the helper function calls itself again a new gensym number, different from the previous one will be passed to it.

| Input | Output | Pass/Fail |
|---|---|---|
| (subs '(VAR x) '(VAR x) '(VAR y)) | (VAR y) | Pass |
| (subs '(VAR k) '(VAR x) '(VAR y)) | (VAR k) | Pass |

| Input |
|---|
| (subs '(LAMBDA x (VAR z)) '(VAR z) '(VAR x)) |
| **Output** |
| (LAMBDA G1 (VAR x)) |
| **Pass/Fail** |
| Pass |
| |

*Table 13: Testing for subs function*

## 5.4.14 Function normalbeta

Normalbeta function has to perform a step of beta reduction using the normal order reduction strategy. As beta reductions is performed if the expression contains a redex it was useful to create two helper functions, one to check when an expression is a redex and another one to check when an expression contains a redex.

- *isRedex:* checks if the expression is a redex.
- *contains-Redex:* checks if the expression contains a redex

| Input | Output | Pass/Fail |
|---|---|---|
| (isRedex '(APP (VAR x) (VAR y))) | ( ) | Pass |
| (isRedex '(APP (LAMBDA x (VAR z)) (VAR y))) | #t | Pass |

*Table 14: Testing for redex function*

| Input | Output | Pass/ Fail |
|---|---|---|
| (contains-Redex '(APP (LAMBDA x (VAR y)) (VAR z))) | #t | Pass |
| (contains-Redex '(LAMBDA x (APP (LAMBDA x (VAR y)) (VAR z)))) | #t | Pass |
| (contains-Redex '(LAMBDA x (APP (VAR x) (VAR y)))) | ( ) | Pass |

*Table 15: Testing for contains-Redex function*

| Input |
|---|
| (normalbeta '(APP (VAR y) (VAR x))) |
| **Output** |
| (APP (VAR y) (VAR x)) |
| **Pass/Fail** |
| Pass |

| Input |
|---|
| (normalbeta '(APP (LAMBDA x (APP (VAR x) (VAR x))) (LAMBDA x (APP (VAR x) (VAR x))))) |
| **Output** |
| (APP (LAMBDA x (APP (VAR x) (VAR x))) (LAMBDA x (APP (VAR x) (VAR x)))) |
| **Pass/Fail** |
| Pass |

| Input |
|---|
| (normalbeta '(APP (LAMBDA x (VAR x)) (LAMBDA z (VAR x)))) |
| **Output** |
| (LAMBDA z (VAR x)) |
| **Pass/Fail** |
| Pass |
| |
| *Table 16: Testing for normalbeta function* |

## 5.4.15 Function normal-reduce

The normal-reduce function has to reduce an expression to its normal form, an expression that does not contain any redexes. If an expression does not contain any redexes then it can not be reduced more and it has to stay like it is.

| **Input** |
|---|
| (normal-reduce '(APP (LAMBDA y (LAMBDA x (VAR y))) (APP (LAMBDA w (APP (VAR w) (VAR w))) (LAMBDA k (VAR k))))) |
| **Output** |
| (LAMBDA G4 (LAMBDA k (VAR k))) |
| **Pass/Fail** |
| Pass |

| **Input** |
|---|
| (normal-reduce '(APP (LAMBDA x (APP (VAR x) (VAR x))) (LAMBDA x (APP (VAR x) (VAR x))))) |
| **Output** |
| *Infinite Loop* |
| **Pass/Fail** |
| Pass |

| **Input** |
|---|
| (normal-reduce '(APP (LAMBDA x (LAMBDA y (VAR y))) (APP (LAMBDA z (APP (VAR z) (VAR z))) (LAMBDA z (APP (VAR z) (VAR z)))))) |
| **Output** |
| (LAMBDA G23566 (VAR G23566)) |
| **Pass/Fail** |
| Pass |

*Table 17: Testing for normal-reduce function*

The first tested input is the expression (λy.λx.y)((λw.ww)(λk.k)) which after three reduction steps reduces to λx.λk.k.
The second tested input is the expression (λx.xx)(λx.xx) which is an infinite loop, as it always reduces to it self.
Finally, the third input is the expression (λx.λy.y)((λz.zz)(λz.zz)) which after one reduction step it reduces to λy.y.

### 5.4.16 Function linearReduce

The linearReduce function is a function that has to reduce an expression if that expression is linear and given with a valid context. The linearReduce function will take as an input an expression *((context)(expression))* and it has to check that the *(context)* is a valid context and that *(expression)* is linear.

In order to do that another two functions were implemeted.

- *isContext:* That checks if the given context is a valid context by comparing all the symbols with each other.
- *isLinear*: which given a context has to check that all variables in the context appear as free in the lambda expression precisely once.

After these two checks the function will reduce *(expression)* to its normal form.

The linearReduce function is the following:

```
(defun linearReduce(term)
  (if (isContext (car term))
    (if (isLinearTerm term)
      (normal-reduce (cadr term))
      '(not a valid linear term)
    )
    '(not a valid context)
  )
)
```

Unfortunately, although the linearReduce function works as expected, the isContext and isLinear functions do not. That has an impact to the linearReduce function which now reduces any expression without checking the linearity or the context as the two functions are now commented out and they just return *true*.

## 5.5 Summary

This chapter gave a documentation of the implementation and also the results of the testing. Some functions worked very well as expected, but on the other hand there are also some functions which did not. An evaluation of the testing and results will be given in the next chapter together with the project overview.

# Chapter 6

# Conclusion

## 6.1 Project summary

In conclusion, this project has attempted to address to the reader the problems that arise with variable capture, though lambda calculus examples. It explained the method used by N.G. de Bruijn in order to avoid this problem and then explained linear binders, which have been used by Miki Tanaka together with the presheaf category on the free strict symmetric monoidal category on 1 to model binding signatures.

Due to the fact of spending more time in researching new, unfamiliar aspects, necessary for the project, the implementation did not reach the desired standards, as two of the main functions are not completed. One of these unfamiliar aspects was category theory. Category theory was necessary in order to continue my research, as almost every document was based on it. It is a really abstract and hard topic to learn and that took longer than estimated**.**

## 6.2 Future work

This project has many possibilities for future work. Tanaka, in her paper "Abstract syntax and variable binding for linear binders" proposed linear binders and gave the theory to deal with them based on untyped language. The same was done previously by Fiore, Plotkin and Turi, but for the usual binders in their paper "Abstract Syntax for variable binding". As, so far we have seen the theory of binding algebra only in untyped

languages, a future work can be the extension of these theory to typed languages, like the simply typed lambda calculus. Also, we should try to find a joined structure for binding algebras for the linear binders and the usual binders.

## 6.3 Personal achievements

Despite the fact that the implementation was not completed successfully, I enjoyed researching that topic and I also managed to gather knowledge that I was not going to be taught in my academic environment.

Also, the fact that I was invited, among the 20 best ideas to present this project in the BCSWomen Undergraduate Lovelace Colloquium, which took place in Leeds on the 16th of April made the work done to seem more valuable to me, as now it is recognised nationally.

Overall I am satisfied with the work done, although I would want to have a successfully completed implementation and if I had more time I would like to give a brief explanation of the process followed by Tanaka in her paper “Abstract syntax and variable binding for linear binders”, to model the binding signature.

# Bibliography

[1] C. Strachey, The varieties of programming language. In Proceedings of the International Computing Symposium, pages 222-233. Cini Foundation, Venice, 1972.

[2] G. Mazzola, Comprehensive mathematics for computer scientists 2 : calculus and ODEs, splines, probability, Fourier and wavelet theory. Fractals and neural networks, categories and lambda calculus, Springer 2005, ISBN: 3540208615

[3] H. P. Barendregt, The lambda calculus: its syntax and semantics, North-Holland 1981, ISBN: 0444854908

[4] J. Lambek and P. J Scott, Introduction to higher order categorical logic, Cambridge University Press 1986, ISBN: 0521356539 paperback

[5] John C. Mitchell, Concepts in programming languages, Cambridge University Press, 2003, ISBN 0521780985

[6] John Power, CM30071: Logic and its Applications (Lecture Notes), Bath University 2008/9

[7] M. Fiore, G. Plotkin, and D. Turi, Abstract syntax and variable binding, In Proceedings of 14th Symposium on Logic in Computer Science, pages 193-202, IEEE Computer Society Press, 1999.

[8] M. Tanaka, Abstract Syntax and Variable Binding for Linear Binders, In Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science, pages 670 - 679, Springer-Verlag 2000, ISBN: 3540679014

[9] Mac Lane, Saunders, Categories for the working mathematician, Springer 1998, ISBN: 9780387984032

[10] N. de Bruijn, Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem - Indagationes Mathematicae , 34, 381-392, 1972

[11] Patrick Lincoln and John Mitchell, Operational aspects of linear lambda calculus, In Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science, pages 235-246 Santa Cruz, California, IEEE Computer Society Press, June 1992

[12] Roy L. Crole, Categories for types, Cambridge University Press 1993, ISBN 0521457017

# Appendix A

# Code

```
; Author: Skevi Anastasiou
; Bath University


; This is a function that returns true if the list is of the form (VAR
x)
(defun isVar (x)
   ; Check if the length of the list is 2
   (if (eq (length x) 2)
     ; If the car of the list is equal to VAR and the cdr is a symbol
it will return true
     (if (and (eq 'VAR (car x)) (symbolp (car(cdr x))))
       #t
     '()
     )
   '()
   )
)


; This is a recursive function that returns true if the list is of the
form (LAMBDA x term),
; where term can be another abstraction or application
(defun isAbs (list)
   ; Check if the length of the list is 3
   (if (eq (length list) 3)
     ; if the car of the list is equal to LAMBDA and x is a symbol we
then
```

```
     ; have to check if term is a symbol, another abstraction or an
application.
     (if (and (eq 'LAMBDA (car list)) (symbolp (car(cdr list))))
       (or (isVar (caddr list)) (isAbs (caddr list)) (isApp (caddr
list)))
       ; if something is not satisfied it will return false
       '()
     )
   ; if the length of the list in not 3 it will return false
   '()
   )
)

; This is a recursive function that returns true if the list is of the
form (APP term1 term2),
; where term1 and term2 can be another application or a variable (VAR
x)
(defun isApp (x)
   ; check if the length of the list is 3 and that the the car of the
list is equal to APP
   (if (and (eq (length x) 3) (eq 'APP (car x)))
     ; checks if term1 is a variable, application or abstraction.
     (if (or (isVar (cadr x)) (isApp (cadr x)) (isAbs (cadr x)))
       ; checks if term2 is a variable, application or abstraction
       (if (or (isVar (caddr x)) (isApp (caddr x)) (isAbs (caddr x)))
         #t
         '()
       )
     '()
     )
   '()
   )
)

; This is a function that returns the variable
(defun return-symb (list)
   ; It checks if the list is a variable, if it is it returns its cdr
   ; otherwise it returns empty
   (if (isVar list)
     (cadr list)
     '()
   )
)


; This is a function that creates a variable using a symbol given by
the user
```

```
(defun create-var (symbol)
  (if (symbolp symbol)
    ; pass the symbol at the beginning of an empty list
    ; and then push VAR to that list too
    (cons 'VAR (cons symbol '()))
  )
)

; This is a function that returns the variable of an abstraction
(defun return-absVar (list)
  ; Checks if the list is an abstraction, if it is it will return
  ; the x in the abstraction (LAMBDA x term)
  ; otherwise it will return null
  (if (isAbs list)
    (cadr list)
    '()
  )
)

; This is a function that returns the body of an abstraction
(defun return-absBody (list)
  ; Checks if the list is an abstraction, if it is it will return
  ; the term in the abstraction (LAMBDA x term)
  ; otherwise it will return null
  (if (isAbs list)
    (caddr list)
    '()
  )
)

; This is a function that creates an abstraction, given
; the variable and the body of it.
(defun create-abs (symbol list)
  ; Checks if the body given is a symbol or a valid application,
variable or
  ; application. It also checks if the given variable is a symbol.
  (if (and (or (symbolp list) (isApp list) (isVar list) (isAbs list))
(symbolp symbol))
    (cons 'LAMBDA (cons symbol (cons list '())))
    '()
  )
)

; This is a function that returns the function term of an application.
(defun return-appFun (list)
  ; It checks if the given list is a valid application.
  ; If it is it returns the function term, otherwise it returns the
```

```
empty list.
   (if (isApp list)
      (cadr list)
      '()
   )
)

; This is a function that returns the argument term of an application.
(defun return-appArg (list)
   ; It checks if the given list is a valid application.
   ; If it is it returns the argument term, otherwise it returns the
empty list.
   (if (isApp list)
      (caddr list)
      '()
   )
)

; This is a function that creates an application given the function and
argument terms.
(defun create-app (fn arg)
   ; Checks if the given function and argument terms are valid
variables or applications.
   (if (and (or (isVar fn) (isApp fn) (isAbs fn)) (or (isVar arg)
(isApp arg) (isAbs arg)))
      ; if they are it creates the application.
      (cons 'APP (cons fn (cons arg '())))
      ; Otherwise it returns the empty list.
      '()
   )
)


; This is a function that renames free variables.
(defun alpha (term x y)
   ; Checks if the term typed is a valid application, abstraction or
variable.
   (if (or (isVar term) (isAbs term) (isApp term))
      ; If it is a variable where its symbol is the one we want to
change it will rename it.
      (cond ((and (isVar term) (eq (return-symb term) x))
             (create-var y)
             )
        ; If it is an abstraction and the abstraction's variable is the
same
        ; with the one that we want to replace we are not going to
replace anything.
```

```
            ((isAbs term)
          (if (eq (return-absVar term) x)
            term
            ; call alpha again in order to convert the body of the
abstraction.
            (create-abs (return-absVar term) (alpha (return-absBody
term) x y))
            ))
       ; If it is an application we are going to alpaconvert the
argument and the function.
             ((isApp term)
          (create-app (alpha (return-appFun term) x y) (alpha (return-
appArg term) x y))
            )
            ; if nothing from the above is true it will return the
term.
             (#t term)

     )
     ; return the empty list is what was typed is not a valid
application, abstraction, variable.
     '()
   )
)

; This is a helper function for subs function which avoids variable
capture.
(defun subsHelpFun(M x N variable)
       ; If M is a variable and the variable symbol is =x it has to be
replaced.
     (cond ((and (isVar M) (eq (return-symb M) (return-symb x))) N)
       ; If M is an application it has to create an application which
will invoke this functions again,
       ; once for the application function and once for the argument.
           ((isApp M) (create-app (subsHelpFun (return-appFun M) x N
(gensym)) (subsHelpFun (return-appArg M) x N (gensym))))
       ; If M is an abstraction and the abstraction variable is not the
same with x we
       ; have to create a new abstraction with a gensym variable and as
a body we will call Helper function again
       ; and alpha the body of M.
           ((and (isAbs M) (not (eq (return-absVar M) (return-symb
x)))) (create-abs variable (subsHelpFun (alpha (return-absBody M)
(return-absVar M) variable) x N (gensym))))
       ; If nothing from the above occures it will return M.
           (#t M)
     )
```

```
)

; This function substitutes in M all free ocureneces of x inside N.
(defun subs (M x N)
  ; Checks if M and N are valid lambda-terms and that x is a variable.
  (if (and (or (isAbs M) (isApp M) (isVar M)) (or (isAbs N) (isApp N)
(isVar N)) (isVar x))
     (subsHelpFun M x N (gensym))
     '()
  )
)

; This is a function that checks if the given term is a redex.
(defun isRedex (term)
  (if (and (isApp term) (isAbs (return-appFun term)))
    #t
    '()
  )
)

; This function checks if the given term contains a redex.
(defun contains-Redex (term)
     ; If the term is an application it has to check if it is a redex
     ; or if the argument or the function of the application contain a
redex.
  (cond ((isApp term)
    (or (isRedex term) (contains-Redex (return-appArg term))
(contains-Redex (return-appFun term))))
     ; If it is a abstraction it has to check if its body contains a
redex.
         ((isAbs term)
     (contains-Redex (return-absBody term)))
     ; If it is a variable the empty list has to be returned since is
not a redex.
         ((isVar term) '())
     ; if nothing from the above is true then the term is an invalid
lambda-term.
         (#t '(Not a Valid Lambda Term!))
  )
)

; This is a function that performs one step beta-reduction.
(defun normalbeta (term)
  ; Checks if the term is a valid lambda term.
  (if (or (isApp term) (isAbs term))
    ; Check if it contains a redex.
    (if (contains-Redex term)
```

```
        ; Check if it is a redex.
        (if (isRedex term)
          ; A beta-reduction step using substitution.
          (subs (return-absBody (return-appFun term)) (create-var
(return-absVar (return-appFun term))) (return-appArg term))
          ; Checks if it is an abstraction.
          (if (isAbs term)
            ; Creates an abstraction and does another beta-reduction
step to the body of the term.
            (create-abs (return-absVar term) (normalbeta (return-
absBody term)))
            ; Checks if the application's argument is a redex.
            (if (isRedex (return-appArg term))
              ; Creates an application and does a b-reduction step to
the application argument.
              (create-app (return-appFun term) (normalbeta (return-
appArg term)))
              ; Creates an application and does a b-reduction step to
the application function.
              (create-app (normalbeta (return-appFun term)) (return-
appArg term))
            )
          )
        )
      ; Returns the term.
      term
      )
   '(Not a valid lambda term!)
   )
)

; This function reduces a term in its normal form.
(defun normal-reduce (term)
   (if (contains-Redex term)
     (normal-reduce (normalbeta term))
     term
   )
)

; This function checks if the context given has distinct variables.
(defun isContext (term)
   ;(setq i 1)
   ;(print (cdr term))
   ;(if (and (not (null term)) (not (null (cdr term))))
   ;   (if (not (eq (car term) (cadr term)))
   ;     ((print (+ i 1))(print (cdr term)) (isContext (cdr term)))
   ;     '()
```

```
   ;    )
   ;'empty
   ;)
   #t
)


(defun isLinearTerm (term)
   ;if the car of the rerm is a valid context it has
   ;to check if all the variables in the context
   ;appear to be free exactly one in the cdr of the term
   #t
)

; This function will reduce a linear term to its normal form.
(defun linearReduce(term)
   (if (isContext (car term))
      (if (isLinearTerm term)
         (normal-reduce (cadr term))
         '(not a valid linear term)
      )
      '(not a valid context)
   )
)
```