Bachelor thesis
Computer Science

Radboud University

Properties of codings in Lambda Calculus

Author:
Nathan van Beusekom
s4571592

First supervisor/assessor:
Prof. Dr. J.H. Geuvers
herman@cs.ru.nl

Second assessor:
Prof. Dr. Erik Barendsen
e.barendsen@cs.ru.nl

July 1, 2018

Abstract

We compare two codings in the λ-calculus: the coding of Barendregt and the coding of Mogensen. We find that they are different in structure and this leads to a class of functions that work for Barendregt's coding but not for Mogensen's coding. On the other hand there exists a more complete self-interpreter for Mogensen's coding.

Contents

Chapter 1

Introduction

In 1936 Church and Kleene introduced the λ-calculus, which is a system that is used to research computable functions.
The λ-calculus is Turing-complete, meaning that it can simulate Turing-machines. One can use a code of a Turing machine to derive some properties of that Turing machine. Just like that we also need codes for λ-terms, so we can derive properties from them. There are some proposed ways of codings. In this thesis we will look at two currently well-known codings. One from Barendregt[3] and one from Mogensen[9]. We will analyze and compare these conceptually different ways of coding and uncover the subtle differences. This leads us the research question:

What possibilities do Mogensen's and Barendregt's coding offer?

When it becomes clear how the two differ and what different functions they make possible, one can easily choose which one is preferred to use for research or to implement in a λ-calculus-based system. If one does not know enough about a coding, one might choose one that does not support the functions that they intent to construct.
In this thesis we will analyze some of the properties of the codings. We will prove their correctness and set up a scheme that one can use to define functions on the codings. We then proceed to show that both codings have their downsides in different situations. Eventually we show that the codings behave differently regarding λ-terms that contain free variables. We find that for Barendregt's coding we cannot write a self-evaluator that works for λ-terms with free variables. On the other hand we find that for Mogensen's coding we cannot write a λ-term that computes the number of different free variables. Additionally we cannot write a λ-term that distinguishes different variables for Mogensen's coding.

Chapter 2

Preliminaries

2.1 Brief Introduction to λ-calculus

We will briefly recall the most important aspects of λ-calculus. For more information see Barendregt's book: The Lambda Calculus. Its Syntax and Semantics[4].

2.1.1 Basics

The λ-calculus is based on the set of λ-terms, Λ. This set is defined by the abstract syntax

$$\Lambda = V \mid \ \Lambda\Lambda \mid \lambda V.\Lambda$$

Here V is the set of variables. We will use lowercase letters for variables and uppercase letter for λ-terms. We only consider untyped λ-calculus, that means that variables have no type.
We can apply substitution to change the value of variables.

Definition 2.1. *A substitution $[x := N] : \Lambda \to \Lambda$ is a function such that for every $M, N \in \Lambda$, $M[x := N]$ replaces every variable x in M with N.*

An applcition is of the form $\Lambda\Lambda$. For example when we have the λ-term AB we say that A is applied to B. When a λ-term is an abstraction it can receive a λ-term by application and substitute the variable bound to the abstraction with the λ-term recieved by the application. For example $(\lambda x.xy)M$ becomes $M\,y$, since the λ-term M is substituted for the variable x. This process of calculating the result of an application is called β-reduction. We define:

Definition 2.2 (Barendregt[4])**.** *The binary relations $\to_\beta$, $\twoheadrightarrow_\beta$ and $=_\beta$ on Λ are defined inductively as follows.*

i *1* $(\lambda x.M)N \to_\beta M[x := N]$;
2 $M \to_\beta N \Rightarrow ZM \to_\beta ZN, MZ \to_\beta NZ$ *and* $\lambda x.M \to_\beta \lambda x.N$.

ii 1 $M \twoheadrightarrow_\beta M$;

2 $M \rightarrow_\beta N \Rightarrow M \twoheadrightarrow_\beta N$;

3 $M \twoheadrightarrow_\beta N, N \twoheadrightarrow_\beta L \Rightarrow M \twoheadrightarrow_\beta L$.

iii 1 $M = N \Rightarrow M =_\beta N$;

2 $M =_\beta N \Rightarrow N =_\beta M$;

3 $M =_\beta N, N =_\beta L \Rightarrow M =_\beta L$.

These relations are pronounced as follows.

$M \twoheadrightarrow_\beta N : M$ β-reduces to N;
$M \rightarrow_\beta N : M$ β-reduces to N in one step;
$M =_\beta N : M$ is β-convertible to N.

β-reduction is left-associative. That means that $ABC = (AB)C$.

If a λ-term contains multiple redexes (reducible applications) that can be contracted, one can choose which one to do first.The following theorem states that the order of reduction doesn't matter.

Definition 2.3 (Barendregt [4])**.** *Church-Rosser Theorem.*
If $M \twoheadrightarrow_\beta N_1$, $M \twoheadrightarrow_\beta N_2$, then for some N_3 one has $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$.

The Church-Rosser theorem implies that a λ-term has at most one normal form.

All the variables that we have considered so far were bound by a λ. In the λ-calculus there are *bound variables* and *free variables*. An abstraction binds all occurrences of a free variable, making it a bound variable.

Example 2.4. *If we look at the λ-term $\lambda x.xy$, then x is a bound variable, since it is bound by the abstraction λx, and y is a free variable, since it is not bound by an abstraction.*

Definition 2.5.

- *A λ-term is closed when all its variables are bound variables (i.e. the λ-term contains no free variables).*
- *A λ-term is open when not all its variables are bound variables (i.e. the λ-term contains free variables).*

The free variables are determined by the environment of the λ-term.

We can change the λ-terms with α-conversion. This is a renaming of the bound variables. An example of an α-conversion is

$$\lambda ab.aab \mapsto \lambda xy.xxy$$

We can do this because we alter the name of the variable as well as the binding λ. This means that the new λ-term will have the exact same effect as the original one. We cannot rename free variables, since we cannot alter the external environment where they are declared.

Definition 2.6 (Mogensen[9])**.**
We define identity and equality for our research in the same way Mogensen did for his.

- *Two λ-terms are considered identical if they only differ in names of bound variables (i.e. they are α-convertible).*
- *Two λ-terms are considered equal if they can be β-reduced to to identical λ-terms.*

2.1.2 Some basic λ-terms

In this section we will define some common functions that we use in our research.

Definition 2.7.

$$I = \lambda x.x$$

$$K = \lambda xy.x$$

$$K_* = \lambda xy.y$$

We represent booleans in the following way.

$$\text{TRUE} = K$$

$$\text{FALSE} = K_*$$

Since booleans are defined as K and K_* we can easily define a if-then-else structure in the following way.

$$\text{IF } A \text{ THEN } B \text{ ELSE } C = A\,B\,C$$

We define the NOT function that inverts a boolean.

$$\text{NOT} = \lambda x.x\,\text{FALSE}\,\text{TRUE}$$

We define the AND function that is true if and only if its two arguments are both true.

$$\lambda ab.a\,(b\ \text{TRUE FALSE})\ \text{FALSE}$$

2.1.3 Church numerals

In the λ-calculus we want to be able to define natural numbers.

Definition 2.8 (Barendregt[4])**.** *The Church numerals $c_0, c_1, c_2, ...$ are defined by*

$$c_n \equiv \lambda f\, x.f^n(x)$$

Example 2.9.

$$c_3 \equiv \lambda f\, x.f(f(f\, x))$$
$$c_5 \equiv \lambda f\, x.f(f(f(f(f\, x))))$$

For readability we define the function $\overline{\cdot} : \mathbb{N} \to \Lambda$

Definition 2.10. *We have the function $\overline{\cdot} : \mathbb{N} \to \Lambda$ such that*

$$\overline{n} \equiv c_n$$

We define basic operations on Church numerals.

Definition 2.11 (Wikipedia [1])**.** *There exist combinators* SUCC*,* PRED*,* ZERO*,* ADD*,* MIN *and* EQ *such that*

$$\begin{aligned}
\text{SUCC}\,\overline{n} &= \overline{n+1} \\
\text{PRED}\,\overline{n+1} &= \overline{n} \\
\text{ZERO}\,\overline{0} &= \text{TRUE} \\
\text{ZERO}\,\overline{n+1} &= \text{FALSE} \\
\text{ADD}\,\overline{m}\,\overline{n} &= \overline{m+n} \\
\text{MIN}\,\overline{m}\,\overline{n} &= \overline{m-n} \\
\text{EQ}\,\overline{m}\,\overline{n} &= \begin{cases} \text{TRUE}, & \textit{if } m = n \\ \text{FALSE}, & \textit{otherwise} \end{cases}
\end{aligned}$$

we construct them as follows.

$$\begin{aligned}
\text{SUCC} &\equiv \lambda n.\lambda f.\lambda x.f(n\, f\, x) \\
\text{PRED} &\equiv \lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\, f))\,(\lambda u.x)\,(\lambda u.u) \\
\text{ZERO} &\equiv \lambda n.n(\lambda x.\text{FALSE})\,\text{TRUE} \\
\text{ADD} &\equiv \lambda m.\lambda n.(n\,\text{SUCC})\,m \\
\text{MIN} &\equiv \lambda m.\lambda n.(n\,\text{PRED})\,m \\
\text{EQ} &\equiv \lambda m.\lambda n.\text{AND}\ (\text{ZERO}\ (\text{MIN}\ m\, n))\,(\text{ZERO}\ (\text{MIN}\ n\, m))
\end{aligned}$$

2.1.4 Fixed-point combinator

In the λ-calculus we can define recursive functions using the fixed-point combinator Y.

Definition 2.12.

$$Y = \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))$$

This might look very complicated, but it is really just a λ-term that produces recursion when given a function.

Example 2.13. *Say we have some function F that we want to be recursive. Let F be the Fibonacci function such that*

$$\begin{aligned} F\,\overline{0} &= \overline{0} \\ F\,\overline{1} &= \overline{1} \\ F\,\overline{n} &= F\,\overline{n-1} + F\,\overline{n-2} \end{aligned}$$

where $n > 1$.

We can define F using the fixed-point combinator Y as follows.

$$\begin{aligned} F \equiv Y\,(\lambda f.\lambda n.&\text{IF (ZERO } n) \\ &\text{THEN } n \\ &\text{ELSE IF (ZERO (PRED } n)) \\ &\quad\text{THEN } n \\ &\quad\text{ELSE ADD } (f(\text{PRED } n))\,(f(\text{PRED (PRED } n)))) \end{aligned}$$

We show that this definition of F satisfies the equations. Let's say

$$\begin{aligned} A \equiv \lambda f.\lambda n.&\text{IF (ZERO } n) \\ &\text{THEN } n \\ &\text{ELSE IF (ZERO (PRED } n)) \\ &\quad\text{THEN } n \\ &\quad\text{ELSE ADD } (f(\text{PRED } n))\,(f(\text{PRED (PRED } n))) \end{aligned}$$

Here we show that recursiveness is indeed achieved. For n with $n > 1$

$$
\begin{aligned}
F\,\overline{n} &= Y\,A\,\overline{n} \\
&= \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))A\,\overline{n} \\
&= (\lambda x.A(x\,x))(\lambda x.A(x\,x))\,\overline{n} \\
&= A((\lambda x.A(x\,x))\,(\lambda x.A(x\,x)))\,\overline{n} \\
&= A(Y\,A)\overline{n} \\
&= A\,F\,\overline{n} \\
&= (\lambda f.\lambda n.\text{IF (ZERO } n) \\
&\qquad \text{THEN } n \\
&\qquad \text{ELSE IF (ZERO (PRED } n)) \\
&\qquad\quad \text{THEN } n \\
&\qquad\quad \text{ELSE ADD } (f(\text{PRED } n))\,(f(\text{PRED (PRED } n)))\,F\,\overline{n} \\
&= \text{IF (ZERO } \overline{n}) \\
&\quad \text{THEN } \overline{n} \\
&\quad \text{ELSE IF (ZERO (PRED } \overline{n})) \\
&\qquad \text{THEN } \overline{n} \\
&\qquad \text{ELSE ADD } (F(\text{PRED } \overline{n}))\,(F(\text{PRED (PRED } \overline{n}))) \\
&= \text{ADD } (F(\text{PRED } \overline{n}))\,(F(\text{PRED (PRED } \overline{n}))) \\
&= \text{ADD } (F\,\overline{n-1})\,(F\,\overline{n-2}) \\
&= F\,\overline{n-1} + F\,\overline{n-2}
\end{aligned}
$$

We show that $F\,\overline{0}$ and $F\,\overline{1}$ are correct as well.

$$
\begin{aligned}
F\,\overline{0} &= \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))A\,\overline{0} \\
&\quad ... \\
&\quad = \text{IF (ZERO } \overline{0}) \\
&\qquad \text{THEN } \overline{0} \\
&\qquad \text{ELSE IF (ZERO (PRED } \overline{0})) \\
&\qquad\quad \text{THEN } \overline{0} \\
&\qquad\quad \text{ELSE ADD } (F(\text{PRED } \overline{0}))\,(F(\text{PRED (PRED } \overline{0}))) \\
&\quad = \overline{0}
\end{aligned}
$$

$$
\begin{aligned}
F\,\overline{1} &= \lambda f.(\lambda x.f(x\,x))(\lambda x.f(x\,x))A\,\overline{1} \\
&\ldots \\
&= \text{IF (ZERO } \overline{1}) \\
&\quad \text{THEN } \overline{1} \\
&\quad \text{ELSE IF (ZERO (PRED } \overline{1})) \\
&\qquad \text{THEN } \overline{1} \\
&\qquad \text{ELSE ADD } (F(\text{PRED } \overline{1}))\,(F(\text{PRED (PRED } \overline{1}))) \\
&= \overline{1}
\end{aligned}
$$

We see that we can use the fixed-point combinator in this format to perfectly define a recursive function. We will use this to define recursive functions later on as well.

2.1.5 Lists in λ-calculus

Barendregt defines some data structures and functions that are based on the functional language LISP. These mostly revolve around lists.

Definition 2.14 (Barendregt[3])**.**

$$
\begin{aligned}
\text{cons} &\equiv \lambda xyz.zxy; \\
\text{nil} &\equiv \lambda xyz.y; \\
\text{null} &\equiv \lambda x.x(\lambda abcd.d); \\
a : b &\equiv \text{cons}\,a\,b; \\
\langle\rangle &\equiv \text{nil}; \\
\langle x_1, ..., x_n\rangle &\equiv x_1 : \langle x_2, ..., x_n\rangle.
\end{aligned}
$$

Lemma 2.15 (Barendregt[3])**.**

(i) null nil $= \lambda xy.x \equiv$ TRUE;

(ii) null (cons $a\,b$) $= \lambda xy.y \equiv$ FALSE.

(iii) There exists terms car and cdr such that

$$\text{car } (\text{cons}\,a\,b) = a;$$

$$\text{cdr } (\text{cons}\,a\,b) = b.$$

Proof.

(i) null nil $\equiv (\lambda x.x(\lambda abcd.d))\lambda xyz.y = (\lambda xyz.y)\lambda abcd.d = \lambda yz.y$;

(ii) null (cons $a\,b$) $\equiv (\lambda x.x(\lambda abcd.d))(\lambda z.z\,a\,b) = (\lambda z.z\,a\,b)\lambda abcd.d = \lambda cd.d$

(iii) Take car $\equiv \lambda x.x(\lambda ab.a)$ and cdr $\equiv \lambda x.x(\lambda ab.b)$.

□

We define some helpful functions on lists.

Definition 2.16 (Barendregt [3])**.** *We define the function* rev *that reverts a list such that*

$$\text{rev}\langle x_1, ..., x_n\rangle = \langle x_n, ..., x_1\rangle.$$

as follows:

$$\text{rev} = \lambda L_1.\text{rev}' L_1 \langle\rangle,$$

with

$$\text{rev}'(a : b)L_2 = \text{rev}' b(a : L_2) \tag{2.1}$$

$$\text{rev}'\text{nil}L_2 = L_2. \tag{2.2}$$

So take $\text{rev}' \equiv Y(\lambda r L_1 L_2.\text{IF (null } L_1)\text{THEN } (L_2)\text{ELSE } (r(\text{cdr}L_1)((\text{car}L_1) : L_2)))$

Additionally, we define the function IN *that checks if some Church numeral is in a list such that*

$$\text{IN } list\, x = \begin{cases} \text{TRUE}, & \text{if } x \in list \\ \text{FALSE}, & \text{otherwise} \end{cases}$$

For example we can construct IN *as follows:*

$$\begin{aligned} \text{IN } \equiv Y(\lambda i.\lambda l x.&\text{IF EQ (car } l)x \\ &\text{THEN TRUE} \\ &\text{ELSE (IF null (cdr } l) \\ &\qquad \text{THEN FALSE} \\ &\qquad \text{ELSE } (i(\text{cdr } l)x))) \end{aligned}$$

And finally, we define the function CONCAT *that concatenates two lists such that*

$$\text{CONCAT } \langle x_1, ..., x_k\rangle \langle x_{k+1}, .., x_n\rangle = \langle x_1, ..., x_n\rangle$$

We can define CONCAT *by having it satisfy the following equations.*

$$\begin{aligned} \text{CONCAT nil}\, l &= l \\ \text{CONCAT } (\text{cons}\, a\, k)\, l &= \text{cons}\, a\, (\text{CONCAT } k\, l) \end{aligned}$$

We construct the λ*-term as follows.*

$$\begin{aligned} \text{CONCAT } \equiv Y(\lambda f.\lambda k\, l.&\text{IF null } k \\ &\text{THEN } l \\ &\text{ELSE cons car } k\, f\, (\text{cdr } k)l) \end{aligned}$$

2.2 Coding in λ-calculus

Our research revolves around coding in the λ-calculus. In the section we will explain what it is and define some properties of codings.

2.2.1 Motivation for coding

The λ-calculus is Turing complete. This means that all functions that can be computed by a Turing Machine can be computed by a λ-term. However, functions on properties of λ-terms cannot be defined in the λ-calculus. We know this because Turing Machines cannot take Turing Machines as an input, and therefore such functions are not computable. For example one cannot write a function that distinguishes the structure of a λ-term (see Appendix A1). This is a result of the β-reduction rule, which means that the λ-term can be reduced before being evaluated by a function. An example of such a function is a function that counts the number of abstractions.

Lemma 2.17. There is no λ-term F such that

$$FM = \overline{n}$$

where n is the number of abstractions in M.

Proof. Say F exists. That would imply:

$$F(II) =_\beta \overline{2}$$

$$II =_\beta I$$

$$F(I) =_\beta \overline{1}$$

Which implies:

$$\overline{1} =_\beta \overline{2}$$

$\overline{1}$ and $\overline{2}$ are both in normal form. The Church-Rosser theorem implied that a λ-term can only have one normal. So this is a contradiction and thus F cannot exist. □

However, it is possible to create such a λ-term for codes on λ-terms. More precisely a representation that cannot be reduced, and is in normal form. We will refer to such a representation as a *coding*. A coding is a way of representing λ-terms such that they are translated or *encoded* as another λ-term that is in normal-form and can be translated back to its original λ-term. This is similar to how a Turing Machine can take a code of a Turing Machine as input and process it, even though it cannot take Turing Machines as input. In λ-calculus this is not the case, but it still affects the computability of such functions.

2.2.2 Defining Codings

A coding is a way of representing a λ-term. Mogensen[9] describes a coding function as: "A *representation schema* for the lambda calculus is an injective (up to identity) mapping $\ulcorner\cdot\urcorner : \Lambda \to \mathrm{NF}_\Lambda$"[9]. This means that codes are always in normal form. To illustrate that with an example, $\ulcorner II \urcorner$ would be the encoded version of II. Here $\ulcorner II \urcorner$ is in normal form. A coding is such a mapping function, but we will add a few more requirements to it.
First of all we want there to be a *self-interpreter*. This is a λ-term that can undo the coding, or in other words, compute M from $\ulcorner M \urcorner$. The second requirement we state is that there exists a recursion scheme on which we can define functions to apply on coded terms. This recursion scheme will be in the same format as Geuvers presented [6]. In conclusion we define a coding in the following way.

Definition 2.18. *A coding is a mapping $\ulcorner\cdot\urcorner : \Lambda \to NF_\Lambda$"[9] such that:*

1. *There exists a self-interpreter $E \in \Lambda$ such that for all $M \in \Lambda$*

$$E\ulcorner M \urcorner \equiv M$$

2. *Given λ-terms $A_1, A_2, A_3 \in \Lambda$ there exists a $H \in \Lambda$ such that*

$$H\ulcorner c \urcorner = A_1 cH$$

$$H\ulcorner MN \urcorner = A_2 \ulcorner M \urcorner \ulcorner N \urcorner H$$

$$H\ulcorner \lambda x.M \urcorner = A_3 \ulcorner M \urcorner H$$

We can define some functions for codings in general. Barendregt defines the following functions:

Definition 2.19 (Barendregt[3])**.** *For a coding $\ulcorner\ \urcorner : \Lambda \to NF_\Lambda$*

(i) An interpreter (or evaluator) is an (external) function $E : \Lambda \to \Lambda$ such that

$$E(\ulcorner M \urcorner) \equiv M.$$

Definition 2.20 (Barendregt[3])**.** *(i) A quote is an (external) function $Q : \Lambda \to \Lambda$ such that*

$$Q(M) \equiv \ulcorner M \urcorner.$$

(ii) A self-quote is a λ-term Q such that

$$QM =_\beta \ulcorner M \urcorner.$$

It is important to note that a self-quote does not exist for any coding.

Fact 2.21 (Barendregt[3])**.** *A self-quote Q cannot exist, since it's existence implies*

$$\ulcorner II \urcorner =_\beta Q(II) =_\beta Q(I) =_\beta \ulcorner I \urcorner.$$

Barendregt also defines a self-interpreter, however his definition is specific for his coding. Therefore we will define two kinds of coding and with that two kinds of self-interpreters.

> **Definition 2.22.** *A weak coding is a coding for which there exists a self-interpreter $E \in \Lambda$ such that for all $M \in \Lambda^0$ one has*
>
> $$E\ulcorner M \urcorner =_\beta M$$
>
> *A strong coding is a coding for which there exists a self-interpreter $E \in \Lambda$ such that for all $M \in \Lambda$ one has*
>
> $$E\ulcorner M \urcorner =_\beta M$$

We see that for the weak coding we state that E only has to work for all $M \in \Lambda^0$. This means that the self-interpreter for a weak coding does not have to work for λ-terms with free variables, but only for closed λ-terms. For strong codings we require that the self-interpreter has to work for open λ-terms as well, which is an extra requirement, but also means that it covers more λ-terms that the self-interpreter works for. In this sense every strong coding is automatically a weak coding as well. Since, if there is a self-interpreter that works for all λ-terms, then there is also a self-interpreter that works for closed λ-terms.

Chapter 3

Comparing codings in λ-calculus

3.1 Barendregt's coding

The first coding we will look at is the coding as defined by Barendregt[3]. Barendregt describes a way of coding where every unique λ-term is a unique natural number.

Definition 3.1 (Barendregt[3])**.** *The λ-coding described by Barendregt is defined as follows:*

$$\#v^{(i)} = \langle 0, i\rangle,$$
$$\#(MN) = \langle 1, \langle \#M, \#N\rangle\rangle,$$
$$\#(\lambda x.M) = \langle 2, \langle \#x, \#M\rangle\rangle.$$

Here for $n, m \in \mathbb{N}$

$$\langle n, m\rangle := \frac{1}{2}(n+m)(n+m+1) + m.$$

Then finally we define $\ulcorner M \urcorner \equiv \#M$ where $\#M$ is a Church numeral.

To give an idea of what such a coding looks like we show some small examples.

Example 3.2. *Let $\#x = 1$. The coding of I in the style of Barendregt then is as follows.*

$$\begin{aligned}\ulcorner I \urcorner &= \ulcorner \lambda x.x \urcorner \\ &= \langle 2, \langle \#x, \#x\rangle\rangle \\ &= \langle 2, \langle\langle 0, 1\rangle, \langle 0, 1\rangle\rangle\rangle \\ &= \overline{117}\end{aligned}$$

Example 3.3. *Let $\#x = 1, \#y = 2$. The coding of $\lambda x.xy$ in the style of Barendregt is:*

$$\begin{aligned}
\ulcorner \lambda x.xy \urcorner &= \ulcorner \lambda x.xy \urcorner \\
&= \langle 2, \langle \#x, \#xy \rangle \rangle \\
&= \langle 2, \langle \langle 0, 1 \rangle, \langle 1, \langle \#x, \#y \rangle \rangle \rangle \rangle \\
&= \langle 2, \langle \langle 0, 1 \rangle, \langle 1, \langle \langle 0, 1 \rangle, \langle 0, 2 \rangle \rangle \rangle \rangle \rangle \\
&= \overline{19879482103}
\end{aligned}$$

We know that every unique λ-term gets a unique pair. We see that Barendregt converts every λ-term to a number using a pairing function. This function is known as the Cantor pairing function.

3.1.1 The Cantor pairing function

The Cantor pairing function is one-to-one. This means that every pair will result in a unique natural number. For the coding this means that every λ-term gets a unique code. Since it is one-to-one, it also means that the pairing function is invertible. The function and its properties are described on Wikipedia[2], but for completeness we will include the projection function and its proof.
The projection functions for the Cantor Pairing function are as follows:

Lemma 3.4. Let $\langle n, m \rangle = z$ and let

$$w = \left\lfloor \frac{\sqrt{8z+1} - 1}{2} \right\rfloor$$

Then:

$$\begin{aligned}
m &= z - \frac{w^2 + w}{2} \\
n &= w - m
\end{aligned}$$

with $\lfloor \cdot \rfloor$ being the floor function.

For the proof see Appendix A2

Since the λ-calculus is Turing-complete, we can construct any computable function in the λ-calculus. From this follows that we can define the projection functions as λ-terms.

Definition 3.5. *We define $P_1, P_2 \in \Lambda$ such that*

$$P_1 \langle \overline{n}, \overline{m} \rangle = \overline{n}$$

$$P_2 \langle \overline{n}, \overline{m} \rangle = \overline{m}$$

for all $n, m \in \mathbb{N}$

3.1.2 Self-interpreter

Barendregt presented a proof by his student, de Bruin, that there exists a self-interpreter for this coding[3]. The proof is constructive, meaning that it also presents a solution, or λ-term, for the self-interpreter.

Lemma 3.6. There exists a self-interpreter for Barendregt's coding.

Proof. (de Bruin[3]). By the representability of computable functions there is a term E_0 such that

$$\begin{aligned} E_0{\ulcorner x \urcorner}F &=_\beta F{\ulcorner x \urcorner}, \\ E_0{\ulcorner MN \urcorner}F &=_\beta F(E_0{\ulcorner M \urcorner}F)(E_0{\ulcorner N \urcorner}F), \\ E_0{\ulcorner \lambda x.M \urcorner}F &=_\beta \lambda x.(E_0{\ulcorner M \urcorner}F_{\ulcorner x \urcorner \to x}), \\ \text{where } F_{\ulcorner x \urcorner \to x} &= F'x, \text{ with} \\ F'x{\ulcorner x \urcorner} &=_\beta x, \\ F'x{\ulcorner y \urcorner} &=_\beta {\ulcorner y \urcorner}, \text{ if } x \not\equiv y \end{aligned}$$

By induction on the structure of $M \in \Lambda$ it can be shown that

$$E_0{\ulcorner M \urcorner}F =_\beta M[x_1 := F{\ulcorner x_1 \urcorner}, ..., x_n := F{\ulcorner x_n \urcorner}] \tag{3.1}$$

(simultaneous subsstitution), where $\{x_1, ..., x_n\} = FV(M)$. Now we can take

$$E \equiv \lambda m.E_0\, m\, I.$$

Indeed, for closed M it follows by equation (3.1) that

$$E{\ulcorner M \urcorner} =_\beta E_0{\ulcorner M \urcorner}I =_\beta M.$$

□

To explain the proof we clarify what E_0 could be exactly, to satisfy the proof.

Definition 3.7. *We can satisfy the proof by defining $E_0 \in \Lambda$ as follows.*

$$\begin{aligned} E_0 \equiv Y(\lambda e.\lambda m\, f.&\text{IF } (P_1 m == 0) \\ &\text{THEN } (f\, m) \\ &\text{ELSE } (\\ &\qquad \text{IF } (P_1 m == 1) \\ &\qquad \text{THEN } (f(e(P_1(P_2 m))f)(e(P_2(P_2 m))f)) \\ &\qquad \text{ELSE } (\lambda x.e\,(P_2(P_2 m))\,(H\,(P_1(P_2 m))\,x)) \\ &)) \end{aligned}$$

Here we define H is

$$H = \lambda zxc.\text{IF } (\text{EQ } c\, z) \text{ THEN } x \text{ ELSE } F\, c$$

This now satisfies:

$$H \ulcorner x \urcorner x \ulcorner x \urcorner = x$$
$$H \ulcorner x \urcorner x \ulcorner y \urcorner = \ulcorner y \urcorner, \quad \text{if } \ulcorner x \urcorner \neq \ulcorner y \urcorner$$

Now the proof holds for $F_{\ulcorner x \urcorner \to x} = H \ulcorner x \urcorner x$

This extensive explanation completes the proof.

3.1.3 α-conversion

We stated that every unique λ-term gets a unique code. But how unique is a λ-term? Barendregt codes every different variable with a different number. This means that λ-terms that are α-equivalent can have a different code. To give an example:

Example 3.8.

$$\ulcorner \lambda v^1.v^1 \urcorner = \overline{117}$$
$$\ulcorner \lambda v^2.v^2 \urcorner = \overline{260}$$
$$\ulcorner \lambda v^1.v^1 \urcorner \neq_\beta \ulcorner \lambda v^2.v^2 \urcorner.$$

We see that $\ulcorner I_{v^1} \urcorner \neq \ulcorner I_{v^2} \urcorner$, which means that Barendregt's coding is not stable under α-conversion.

3.1.4 Recursion scheme

We want to be able to define functions on Barendregt's coding. For this reason we define a recursion scheme on Barendregt's coding. We will do this in a similar way Geuvers did for the Böhm-Piperno-Guerini coding.

Lemma 3.9 (Geuvers [6])**.** For Barendregt's coding, given $A_1, A_2, A_3 \in \Lambda$ there is an $H \in \Lambda$ such that

$$H \ulcorner x \urcorner = A_1 \ulcorner x \urcorner H$$

$$H \ulcorner MN \urcorner = A_2 \ulcorner M \urcorner \ulcorner N \urcorner H$$

$$H \ulcorner \lambda x.M \urcorner = A_3 \ulcorner x \urcorner \ulcorner M \urcorner H$$

The lemma is slightly different from the lemma that Geuvers presented. We want to call A_1 on $\ulcorner x \urcorner$ rather than on x. This is because Barendregt's coding easily allows this and $\ulcorner x \urcorner$ offers more functionality than x itself.

Proof. With the fixed point combinator Y we can define H as follows

$$\begin{aligned} H \equiv Y(\lambda h.\lambda m.&\text{IF } (P_1 m == 0) \\ &\text{THEN } (A_1 m h) \\ &\text{ELSE } (\\ &\qquad \text{IF } (P_1 m == 1) \\ &\qquad \text{THEN } (A_2 (P_1 (P_2 m))(P_2 (P_2 m))h) \\ &\qquad \text{ELSE } (A_3 (P_1 (P_2 m))(P_2 (P_2 m))h) \\ &)) \end{aligned}$$

□

This proof very clearly does exactly what the recursion scheme lemma requires.

We can use the scheme to define functions on Barendregt's coding. We can, for example try to define a function that counts the numer of λ-abstractions, like we described in Section 2.

Lemma 3.10. There is a function F such that

$$F^{\ulcorner} M^{\urcorner} = \overline{n}$$

where n is the number of abstractions in M

Proof. The concept of this function can easily be represented as:

$$F^{\ulcorner} x^{\urcorner} = \overline{0}$$

$$F^{\ulcorner} M N^{\urcorner} = F^{\ulcorner} M^{\urcorner} + F^{\ulcorner} N^{\urcorner}$$

$$F^{\ulcorner} \lambda x.M^{\urcorner} = \overline{1} + F^{\ulcorner} M^{\urcorner}$$

This way the function will just recursively go through the lambda terms and add 1 when it encounters an abstraction. We can translate to the recursion scheme in the following way.

$$A_1 = \lambda x h.\overline{0}$$

$$A_2 = \lambda m n h.hm + hn$$

$$A_3 = \lambda x m h.\overline{1} + hm$$

Induction shows that we indeed count the number of abstraction with the scheme □

3.1.5 Digression based on Barendregt's paper from 1991

In this part Barendregt solves a problem stated by Dr Wim Vree. He does that by using his coding. This is quite interesting and thus we will go over the entire section. After defining the way of coding, Barednregt defines some data structures and function that are based on the functional language LISP. We defined these in Section 2.

A problem was presented by Vree which Barendregt solves using his coding. This shows an exceptional use of coding which can be useful for other problems as well.

Problem 3.11 (Dr Wim Vree [3])**.** Does there exist a λ-term F such that for all $n \in \mathbb{N}$ one has

$$F\overline{n} = \lambda x_1 ... x_n.\langle x_1, ..., x_n\rangle$$

Lemma 3.12. There is a λ-term F such that for all $n \in \mathbb{N}$ one has

$$F\overline{n} = \lambda x_1 ... x_n.\langle x_1, ..., x_n\rangle$$

Barendregt shows a solution for this using coding. He states that it is possible to compute the code of the answer and like that it is also possible to compute the actual answer, since we can evaluate the code. He gives the following solution:

Proof. (Barendregt [3])
Write $M_n \equiv \lambda x_1 ... x_n.\langle x_1, ..., x_n\rangle$. Clearly, $\#M_n$ is computable from n, say $\#M_n = g(n)$ with g recursive. Let g be λ-defined by G, say. Then

$$G\overline{n} = \ulcorner g(n) \urcorner = \ulcorner M_n \urcorner$$

Then $F = \lambda n.E(Gn)$ satisfies the stated problem

$$F\overline{n} = E(G\overline{n}) = E\ulcorner M_n \urcorner = M_n.$$

□

The concept of "if the code of M is computable, M is computable" is interesting, since it can more easily prove problem such as stated by Vree. We now show Vree's constructive solution

Solution 3.13. *(Vree[3])*
The function F where $F\ulcorner n \urcorner x_1 ... x_n = \langle x_1, ..., x_n\rangle$ is given by $F = \lambda n.V\, n\, \mathrm{nil}$

Here the λ-term V is defined by

$$V\ulcorner n + 1 \urcorner = \lambda L x.(V\ulcorner n \urcorner (x : L)),$$
$$V\ulcorner 0 \urcorner = \mathrm{rev}.$$

In Appendix A3 we elaborate on this by constructing V and showing that this indeed satisfies the equation from Lemma 3.12.

Barendregt left some exercises for the reader which we will answer for a complete understanding of Barendregt's paper.

Lemma 3.14. There is no λ-term G such that for all $n \in \mathbb{N}$ one has

$$Gx_1...x_n = \langle x_1, ..., x_n \rangle.$$

Proof. First we compute Gx_1 according to the function definition.

$$Gx_1 = \langle x_1 \rangle$$

Then we compute Gx_1x_2.

$$Gx_1x_2 = \langle x_1, x_2 \rangle$$

If G exists, $(Gx_1)x_2 = Gx_1x_2$ should hold. Therefore $\langle x_1 \rangle x_2 = \langle x_1, x_2 \rangle$ should then hold as well. Let's test this.

$$\begin{aligned}
\langle x_1 \rangle x_2 &= (\text{cons}\, x_1\, \text{nil})\, x_2 \\
&= ((\lambda xyz.zxy)x_1\, \text{nil})x_2 \\
&= (\lambda z.z\, x_1\, \text{nil})x_2 \\
&= x_2\, x_1\, \text{nil}. \\
\langle x_1, x_2 \rangle &= \text{cons}\, x_1\, (\text{cons}\, x_2\, \text{nil}) \\
&= \text{cons}\, x_1\, ((\lambda xyz.zxy)x_2\, \text{nil}) \\
&= \text{cons}\, x_1\, (\lambda z.z\, x_2\, \text{nil}) \\
&= (\lambda xyz_1.z_1xy)\, x_2\, (\lambda z.z\, x_2\, \text{nil}) \\
&= \lambda z_1.x_2\, (\lambda z.z\, x_2\, \text{nil})
\end{aligned}$$

And clearly $x_2\, x_1\, \text{nil} \neq \lambda z_1.x_2\, (\lambda z.z\, x_2\, \text{nil})$, which are both in normal form. The Church-Rosser theorem states, when a term can be reduced to two different terms, those two terms can be reduced to the same term. Here this is not the case since they are both reduced from Gx_1x_2 but cannot be reduced any further. This means that G cannot exist. □

We can even elaborate on this proof by contradiction by showing that we cannot add G to λ-calculus, since that would make λ-calculus inconsistent.

Lemma 3.15. Adding such a function G to λ-calculus makes λ-calculus inconsistent.

Proof. If we add G, then for all $x_1, x_2 \in \Lambda$, $\langle x_1, x_2\rangle =_\beta x_2 x_1 \text{nil}$ This would imply:

$$\begin{aligned}
\langle I, K\rangle IIII &=_\beta (KI\text{nil})IIII \\
&=_\beta IIIII \\
&=_\beta I \\
\langle I, K\rangle IIII &=_\beta (\lambda z.zI(\lambda z'.z'K\text{nil}))IIII \\
&=_\beta (\lambda z'.z'K\text{nil})III \\
&=_\beta K\text{nil}II \\
&=_\beta (\lambda y.\text{nil})II \\
&=_\beta (\lambda y.\lambda abc.b)II \\
&=_\beta (\lambda bc.b) \\
&=_\beta K
\end{aligned}$$

So the addition of G to the λ-calculus implies $I =_\beta K$ and that would make the λ-calculus inconsistent, therefore G cannot exist. □

The problem with creating G is that, if you want G to work on a variable number of λ-terms, applying G on the first argument should create a new function (since you might want to apply it again later). Then applying that new function on the second argument should create another new function. And so forth continuing with an infinite amount of functions. This should not be a problem if the desired result of G is in the form of such a function, but the proof shows that lists are not such a function, and therefore G can not exist.

One might wonder then how can F exist? Doesn't that functions get applied on a variable amount of arguments only to produce a list?
The trick here is that F is not being applied on $x_1, ..., x_n$, but $F^\ulcorner n^\urcorner$ gets is the real function. $F^\ulcorner n^\urcorner$ is not a constant function declaration and doesn't have to work with unexpectedly being applied more, since it requires the expected number of x arguments n at the beginning and adding one would mean that it had to be $n + 1$. So the function had to know this beforehand. This means that the function declaration more strict in a sense that we have to specify what we are going to do first, which basically means that we are not giving a variable amount of arguments but a set amount of arguments. The function $F^\ulcorner n^\urcorner$ then to work for that set amount of arguments.

The second exercise that Barendregt presented was:

Lemma 3.16. There is a λ-term H such that for all $n \in \mathbb{N}$ one has

$$H^{\ulcorner} n^{\urcorner} x_1 ... x_n = \lambda z.z x_1 ... x_n.$$

Proof. We can try to use F to solve this problem. So we assume F turns the arguments into $\langle x_1, ..., x_n \rangle$. We want to make a function J such that for all $n \in \mathbb{N}$ one has

$$J \, \langle x_1, ..., x_n \rangle \, z = z \, x_1 ... x_n$$

We do this by adding every element of the list to the resulting function one by one.

$$J \, (\mathrm{cons} \, a \, l) \, z = J \, l \, (z a)$$

$$J \, \mathrm{nil} \, z = z$$

Now we can construct H in the same way as F, using V. We just alter $V^{\ulcorner} 0^{\urcorner}$ since that is the final transformation. We define the altered version V' as follows.

$$V' \overline{n+1} = \lambda L x.(V' \, \overline{n} \, (x : L)),$$

$$V' \overline{0} = \lambda x.J \, (\mathrm{rev} \, x)$$

with

$$\begin{aligned} V = Y(\lambda v n.&\mathrm{IF} \ (\mathrm{ZERO} \ n) \\ &\mathrm{THEN} \ (\lambda x.J \, (\mathrm{rev} \, x)) \\ &\mathrm{ELSE} \ (\lambda L x.v(\mathrm{PRED} \ n)(x : L))) \end{aligned}$$

Then $F \equiv \lambda n.V \, n \, \mathrm{nil}$ indeed satisfies the equation from Lemma 3.12.

$$\begin{aligned} F \, \overline{n} \, x_1 \, ... \, x_n &= V \, \overline{n} \, \mathrm{nil} \, x_1 \, ... \, x_n \\ &= (\lambda L x.V \, \overline{n-1} \, (x : L)) \, \mathrm{nil} \, x_1 \, ... \, x_n \\ &= V \, \overline{n-1} \, (x_1 : \mathrm{nil}) x_2 \, ... \, x_n \\ &= (\lambda L x.V \, \overline{n-2} \, (x : L)) \, (x_1 : nil) \, x_2 \, ... \, x_n \\ &= V \, \overline{n-2} \, (x_2 : x_1 : \mathrm{nil}) \, x_3 \, ... \, x_n \\ &... \\ &= V \, \overline{0} \, (x_n : ... : x_1 : \mathrm{nil}) \\ &= (\lambda x.J \, (\mathrm{rev} \, x)) \langle x_n, ..., x_1 \rangle \\ &= J \langle x_1, ..., x_n \rangle. \\ &= \lambda z.J \, \langle x_2, ..., x_n \rangle (z \, x_1) \\ &= \lambda z.J \, \langle x_3, ..., x_n \rangle (z \, x_1 \, x_2) \\ &... \\ &= \lambda z.\mathrm{nil} \, (z \, x_1 \, ... \, x_n) \\ &= \lambda z.z \, x_1 \, ... \, x_n \end{aligned}$$

□

3.2 Mogensen's Coding

The second coding that we will consider is the coding of Mogensen. He created a coding that goes with a small and elegant self-interpreter. We will first look at the coding that Mogensen defined and then at the self-interpreter and self-reducer that he constructed in his paper. He showed both of those to be very efficient.[9]

Definition 3.17. *(Mogensen) The λ-coding of Mogensen is defined as follows[9]:*

$$\ulcorner x \urcorner = Var\, x$$

$$\ulcorner MN \urcorner = App \ulcorner M \urcorner \ulcorner N \urcorner$$

$$\ulcorner \lambda x.M \urcorner = Abs(\lambda x. \ulcorner M \urcorner)$$

He then uses a way to represent signatures to make these functions Var, App and Abs. This results in:

Definition 3.18 (Mogensen)**.**

$$Var\, x = \lambda abc.ax$$

$$App \ulcorner M \urcorner \ulcorner N \urcorner = \lambda abc.b \ulcorner M \urcorner \ulcorner N \urcorner$$

$$Abs(\lambda x. \ulcorner M \urcorner) = \lambda abc.c(\lambda x. \ulcorner M \urcorner)$$

Example 3.19. *The coding of $\lambda xy.yx$ now is as follows:*

$$Abs(\lambda x.Abs(\lambda y.App((Var\, y)(Var\, x))))$$

The full λ-term for this is:

$$\lambda abc.c(\lambda x.(\lambda abc.c(\lambda y.(\lambda abc.b(\lambda abc.ay)(\lambda abc.ax)))))$$

3.2.1 Self-interpreter

Mogensen requires his self-interpreter to do the following.

Lemma 3.20 (Mogensen[9])**.** There exists a λ-term E such that

$$E(Var\, x) = x$$

$$E(App\, M\, N) = E(M)E(N)$$

$$E(Abs\, M) = \lambda v.E(Mv)$$

Proof. (Mogensen[9]) By pattern matching on the structure of the coding we compute:

$$\begin{aligned} E \equiv Y(\lambda e.\lambda m.m(\lambda x.x) \\ (\lambda mn.(e\,m)(e\,n)) \\ (\lambda m.\lambda v.e(m\,v))) \end{aligned}$$

We show that this indeed satisfies the lemma.
For Var:

$$\begin{aligned} E(Var\,x) &= (\lambda abc.ax)(\lambda x.x) \\ &\quad (\lambda mn.(E\,m)(E\,n)) \\ &\quad (\lambda m.\lambda v.E(m\,v)) \\ &= (\lambda x.x)x \\ &= x \end{aligned}$$

For *App*:

$$\begin{aligned} E(App\,M\,N) &= (\lambda abc.b\,M\,N)(\lambda x.x) \\ &\quad (\lambda mn.(E\,m)(E\,n)) \\ &\quad (\lambda m.\lambda v.E(m\,v)) \\ &= (\lambda mn.(E\,m)(E\,n))\,M\,N \\ &= (E\,M)(E\,N) \end{aligned}$$

For *Abs*:

$$\begin{aligned} E(Abs\,M) &= (\lambda abc.b\,M\,N)(\lambda x.x) \\ &\quad (\lambda mn.(E\,m)(E\,n)) \\ &\quad (\lambda m.\lambda v.E(m\,v)) \\ &= (\lambda m.\lambda v.E(m\,v))\,M \\ &= \lambda v.E(M\,v) \end{aligned}$$

□

Mogensen states the following about his self-interpreter: "It easy to prove by induction that recursive of these equations and reduction of redexes of form (Mv) from the third equation, will reduce $E(\ulcorner N \urcorner)$ to a λ-term that is *identical* to N, for any λ-term N. Note that this includes λ-terms with free variables. This is not the case for the representation that Barendregt used, as explained in his article".[9]
Here he states that his self-interpreter is correct and that it can also decode free variables, unlike Barendregt's code could do.

3.2.2 α-conversion

Since Mogensen doesn't encode his variables as number but rather keeps them intact as a λ-term, he can easily define a self-interpreter that can decode free variables as well. This makes Mogensen's coding a strong coding. Obviously the advantage of this property is that the self-interpreter is complete for the entire λ-calculus. This way of variable encoding also has a disadvantge, which we will go over in the next section.

This way of variable encoding ensures that the variables are the same λ-term as they were. This results in Mogensen's coding being stable under α-conversion, because λ-terms are stable under α-conversion.

3.2.3 Recursion scheme

Again we want to define a recursion scheme so that we can define functions on the coding.

Lemma 3.21 (Geuvers [6])**.** For Mogensen's coding, given $A_1, A_2, A_3 \in \Lambda$ there is an $H \in \Lambda$ such that

$$H\ulcorner x \urcorner = A_1\ulcorner x \urcorner H$$

$$H\ulcorner MN \urcorner = A_2\ulcorner M \urcorner\ulcorner N \urcorner H$$

$$H\ulcorner \lambda x.M \urcorner = A_3\ulcorner M \urcorner H$$

Note that the lemma is slightly different for abstractions. This is a result of Mogensen's way of coding them. The λ-abstractions are still λ-abstractions in the form of $Abs\,\lambda x.\ulcorner M \urcorner$. Since the λx is still an abstraction and not a hard-coded variable, we cannot retrieve the x. We can show that H exists by imitating the way that Mogensen defined his self-interpreter.

Proof. By pattern matching on the structure of the coding we compute:

$$\begin{aligned} H \equiv Y\lambda h.\lambda m.m(&\lambda x.A_1\,(\lambda abc.ax)\,h) \\ &(\lambda mn.A_2\,m\,n\,h) \\ &(\lambda m.\lambda v.A_3\,(m\,v)\,h) \end{aligned}$$

□

Now we can easily define the abstraction counting function just like we did for Barendregt's coding.

Lemma 3.22. There is a function F such that

$$F(\ulcorner M \urcorner) = \overline{n}$$

where n is the number of abstractions in M

Proof. The concept of this function can easily be represented as:

$$F^{\ulcorner}x^{\urcorner} = \overline{0}$$

$$F^{\ulcorner}MN^{\urcorner} = F^{\ulcorner}M^{\urcorner} + F^{\ulcorner}N^{\urcorner}$$

$$F^{\ulcorner}\lambda x.M^{\urcorner} = \overline{1} + F^{\ulcorner}M^{\urcorner}$$

This way the function will just recursively go through the lambda terms and add 1 when it encounters an abstraction. We can translate to the recursion scheme in the following way.

$$A_1 = \lambda xh.\overline{0}$$

$$A_2 = \lambda mnh.hm + hn$$

$$A_3 = \lambda xmh.\overline{1} + hm$$

Induction shows that we indeed count the number of abstraction with the scheme □

We see that the proof is exactly the same as for Barendregt's coding. This just shows how useful recursion schemes are to define functions over codings. The representation is almost the same which means that functions can easily be defined over multiple codings.

3.3 Creating the free variable function

Since the codings are still different in some aspects, we cannot define every function on both codings. In this section we go over an example that will work for only Barendregt's coding and not Mogensen's coding. We will try to define a function that counts the number of different free variables of a λ-term.

3.3.1 Counting different free variables for Barendregt's coding

We can create this function by creating a helper function L, that gives back the list of coded free variables of the λ-term. When we have that list we can simply count the different variables in that list. For the helper function we simply want to keep a list of bound variables. When we reach a variable we want to check if that variable is in the list and add it to the resulting (or returning) list if it is not in the bound variables list. If it is in there we just return nil. When an application is encountered we simply want to concatenate the lists that result from $L^{\ulcorner}M^{\urcorner}$ and $L^{\ulcorner}N^{\urcorner}$. When we encounter an abstraction we want to add the bound variable to the list of

bound variables and call the function on the code that comes with it. To clarify this, we want the function to satisfy these equations:

$$L\,abslist\ulcorner v\urcorner \begin{cases} \text{nil}, & \text{if } \ulcorner v\urcorner \in abslist \\ \text{cons}\ulcorner v\urcorner\,\text{nil}, & \text{otherwise} \end{cases}$$

$$L\,abslist\ulcorner MN\urcorner = \text{CONCAT}\,M\,N$$

$$L\,abslist\ulcorner \lambda x.M\urcorner = F\,(\text{cons}\,x\,abslist)\,M$$

Since Barendregt's self-interpreter does not work for free variables, we will return a list of the coded free variables. This way we can more easily distinguish them.

Lemma 3.23. There exists a function $D \in \Lambda$ for Barendregt's coding such that

$$D\ulcorner M\urcorner = \overline{n}$$

where n is exactly the the number of different free variables in M.

Due to Barendregt's static way of coding we can easily obtain these variables.

We can define the function using the recursion scheme and the definition, similar to what we have done before. However, since we want to keep a list throughout the function, we will pass that on as well. We want our recursion scheme to look like this:

Lemma 3.24. For Barendregt's coding, given $A_1, A_2, A_3 \in \Lambda$ there is an $H \in \Lambda$ such that

$$H\,l\ulcorner x\urcorner = A_1\,l\ulcorner x\urcorner H$$

$$H\,l\ulcorner MN\urcorner = A_2\,l\ulcorner M\urcorner\ulcorner N\urcorner H$$

$$H\,l\ulcorner \lambda x.M\urcorner = A_3\,l\ulcorner M\urcorner H$$

We can prove this in a similar way that we did for the original recursion scheme.

Proof. With the fixed point combinator Y we can define H as follows

$$\begin{aligned} H \equiv Y(\lambda h.\lambda lm.&\text{IF } (P_1 m == 0) \\ &\text{THEN } (A_1 lmh) \\ &\text{ELSE } (\\ &\qquad \text{IF } (P_1 m == 1) \\ &\qquad \text{THEN } (A_2 l(P_1(P_2 m))(P_2(P_2 m))h) \\ &\qquad \text{ELSE } (A_3 l(P_1(P_2 m))(P_2(P_2 m))h) \\ &)) \end{aligned}$$

□

This proof very clearly does exactly what the recursion scheme lemma requires.

Now we can define A_1, A_2 and A_3.

$$\begin{aligned} A_1 &= \lambda lxh.\text{IF } (\text{IN } l\,x) \\ &\qquad\quad \text{THEN } (\text{nil}) \\ &\qquad\quad \text{ELSE } (\text{cons}\,x\,\text{nil}) \\ A_2 &= \lambda lmnh.\text{CONCAT } (h\,l\,m)(h\,l\,n) \\ A_3 &= \lambda lxmh.h(\text{cons}\,x\,l)m. \end{aligned}$$

3.3.2 D for Mogensen coding

Mogensen encodes variables in a different way. Due to his representation $Var\,x$ we can still replace the x with other free variables. Therefore we cannot make the function D for Mogensen's coding.

Lemma 3.25. We cannot make the function $D \in \Lambda$ for Mogensen's coding such that

$$D^{\ulcorner} M^{\urcorner} = \overline{n}$$

where n is exactly the the number of different free variables in M.

Proof. Assume that D exists for Mogensen. That would imply:

$$\begin{aligned} (\lambda x.(D(^{\ulcorner} xy^{\urcorner})))y &=_\beta (\lambda x.\overline{2})y \\ (\lambda x.(D(^{\ulcorner} xy^{\urcorner})))y &=_\beta \overline{2} \end{aligned}$$

On the other hand it implies:

$$\begin{aligned} (\lambda x.(D(^{\ulcorner} xy^{\urcorner})))y &=_\beta (\lambda x.(D(App(Var\,x)(Var\,y))))y \\ (\lambda x.(D(^{\ulcorner} xy^{\urcorner})))y &=_\beta D(App(Var\,y)(Var\,y)) \\ (\lambda x.(D(^{\ulcorner} xy^{\urcorner})))y &=_\beta D(^{\ulcorner} yy^{\urcorner}) \\ (\lambda x.(D(^{\ulcorner} xy^{\urcorner})))y &=_\beta \overline{1} \end{aligned}$$

Combine those, and the existance of D for Mogensen's coding implies:

$$\overline{1} =_\beta \overline{2}$$

And clearly this is not the case, and both $\overline{1}$ and $\overline{2}$ are in normal form. The Church-Rosser theorem states, when a term can be reduced to two different terms, those two terms can be reduced to the same term. Here this is not the case since they are both reduced from $(\lambda x.(D(^{\ulcorner} xy^{\urcorner})))y$ but cannot be reduced any further. This means that D does not exist for Mogensen's coding. □

3.3.3 Why is this the case

Barendregt and Mogensen are very distinct in design choices for their coding. The differences lie in the way of encoding abstractions and the way of encoding variables. Barendregt chooses a way of encoding that is very much like hard-coding. He converts the variables to numbers. Mogensen, on the other hand, preserves the variables in a data structure. This difference results that we cannot change Barendregt's coding using β-reduction, since it is a natural number and therefore always in normal form, however we can change Mogensen's coding using β-reduction. This emerges from the variable being a real variables that can be resolved using β-reduction. Note that in $\ulcorner xy \urcorner$ x is a free variable because it is declared in the external environment. We then use that external environment to show that D is impossible.

3.4 Creating the normal form function

In this section we will go over an example function that can be defined for both codings. We define a λ-term N that will test if a coded λ-term is in normal form. This λdoes not exist for regular λ-terms.

Lemma 3.26. There is no λ-term F such that

$$F\,M \begin{cases} \text{TRUE}, & \text{if } M \text{ is in normal form} \\ \text{FALSE}, & \text{otherwise} \end{cases}$$

Proof. Say such a function F exists. That would imply:

$$F\,I = \text{TRUE}$$

$$F\,II = \text{FALSE}$$

$$I = II$$

$$\text{TRUE} = \text{FALSE}$$

This is a contradiction and thus F cannot exist. □

We will define the λ-term N such that it satisfies the following equation for both codings:

$$N \ulcorner M \urcorner \begin{cases} \text{TRUE}, & \text{if } M \text{ is in normal form} \\ \text{FALSE}, & \text{otherwise} \end{cases}$$

We know that a λ-term is in normal form if either of the following holds:

- It is a variable

- It is an application, the applied λ-term is not an abstraction and both λ-terms are in normal form.
- It is an abstraction and the λ-term under the $\lambda$is in normal form.

This results in the following equations that N should satisfy:

$$N\ulcorner x\urcorner = \text{TRUE}$$

$$N\ulcorner AB\urcorner = N\ulcorner A\urcorner \,\&\&\, N\ulcorner B\urcorner \,\&\&\, \text{notAbstraction}\ulcorner A\urcorner$$

$$N\ulcorner \lambda x.A\urcorner = N\ulcorner A\urcorner$$

3.4.1 Making the normal form function for Barendregt's coding

We can easily statisfy these equations using the regular recursion scheme for Barendregt. We get:

$$\begin{aligned}
A_1 &= \lambda xh.\text{TRUE} \\
A_2 &= \lambda mnh.\text{AND (AND } hm\, hn)\,(\text{NOT(EQ } (P_1\, m)\,\overline{2})) \\
A_3 &= \lambda xmh.h\, m
\end{aligned}$$

Then for Barendregt's coding we construct N:

$$\begin{aligned}
N \equiv Y(\lambda h.\lambda m.&\text{IF } (P_1 m == 0) \\
&\text{THEN } (A_1 mh) \\
&\text{ELSE (} \\
&\qquad \text{IF } (P_1 m == 1) \\
&\qquad \text{THEN } (A_2(P_1(P_2 m))(P_2(P_2 m))h) \\
&\qquad \text{ELSE } (A_3(P_1(P_2 m))(P_2(P_2 m))h) \\
&\text{))}
\end{aligned}$$

We can show that N indeed satisfies the equations. We will denote $\overline{x}$ for the number that is assigned to the variable x.

$$\begin{aligned}
N\ulcorner x\urcorner &= N\,\langle 0, \overline{x}\rangle \\
&= A_1\ulcorner x\urcorner N \\
&= (\lambda xh.\text{TRUE})\ulcorner x\urcorner N \\
&= \text{TRUE}.
\end{aligned}$$

$$\begin{aligned} N^{\ulcorner} A\, B^{\urcorner} &= N\,\langle 2, \langle \#A, \#B\rangle\rangle \\ &= A_2{}^{\ulcorner} A^{\urcorner\ulcorner} B^{\urcorner} N \\ &= (\lambda mnh.\text{AND }(\text{AND } hm\, hn)\,(\text{NOT}(\text{EQ }(P_1\, m)\,\overline{2})))^{\ulcorner} A^{\urcorner\ulcorner} B^{\urcorner} N \\ &= \text{AND }(\text{AND }(N^{\ulcorner} A^{\urcorner})\,(N^{\ulcorner} B^{\urcorner}))\,(\text{NOT}(\text{EQ }(P_1{}^{\ulcorner} A^{\urcorner})\,\overline{2})) \end{aligned}$$

$$\begin{aligned} N^{\ulcorner}\lambda x.M^{\urcorner} &= N\,\langle 1, \langle \#x, \#M\rangle\rangle \\ &= A_3{}^{\ulcorner} x^{\urcorner\ulcorner} M^{\urcorner} N \\ &= (\lambda xmh.h\, m)^{\ulcorner} x^{\urcorner\ulcorner} M^{\urcorner} N \\ &= N^{\ulcorner} M^{\urcorner}. \end{aligned}$$

We will include some examples to show how N works.

Example 3.27.

$$\begin{aligned} N^{\ulcorner} K^{\urcorner} &= N^{\ulcorner}\lambda xy.x^{\urcorner} \\ &= N\langle 2, \langle {}^{\ulcorner} x^{\urcorner}, {}^{\ulcorner}\lambda y.x^{\urcorner}\rangle\rangle \\ &= A_3{}^{\ulcorner} x^{\urcorner\ulcorner}\lambda y.x^{\urcorner} N \\ &= (\lambda xmh.h\, m)^{\ulcorner} x^{\urcorner\ulcorner}\lambda y.x^{\urcorner} N \\ &= N^{\ulcorner}\lambda y.x^{\urcorner} \\ &= N\langle 2, \langle {}^{\ulcorner} y^{\urcorner}, {}^{\ulcorner} x^{\urcorner}\rangle\rangle \\ &= A_3{}^{\ulcorner} y^{\urcorner\ulcorner} x^{\urcorner} N \\ &= (\lambda xmh.h\, m)^{\ulcorner} y^{\urcorner\ulcorner} x^{\urcorner} N \\ &= N^{\ulcorner} x^{\urcorner} \\ &= N\langle 0, \overline{x}\rangle \\ &= A_1{}^{\ulcorner} x^{\urcorner} N \\ &= (\lambda xh.\text{TRUE})^{\ulcorner} x^{\urcorner} N \\ &= \text{TRUE}. \end{aligned}$$

Example 3.28. *For the readability of the example of $N^{\ulcorner} II^{\urcorner}$, we first compute $N^{\ulcorner}\lambda x.x^{\urcorner}$*

$$\begin{aligned} N^{\ulcorner}\lambda x.x^{\urcorner} &= A_3{}^{\ulcorner} x^{\urcorner\ulcorner} x^{\urcorner} N \\ &= N^{\ulcorner} x^{\urcorner} \\ &= \text{TRUE} \end{aligned}$$

$$
\begin{aligned}
N^{\ulcorner} II^{\urcorner} &= N^{\ulcorner}(\lambda x.x)\lambda x.x^{\urcorner} \\
&= N\langle 1, \langle {}^{\ulcorner}\lambda x.x^{\urcorner}, {}^{\ulcorner}\lambda x.x^{\urcorner}\rangle\rangle \\
&= A_2{}^{\ulcorner}\lambda x.x^{\urcorner}{}^{\ulcorner}\lambda x.x^{\urcorner}N \\
&= (\lambda mnh.\text{AND } (\text{AND } hm\, hn)\, (\text{NOT}(\text{EQ } (P_1\, m)\, \overline{2})))^{\ulcorner}\lambda x.x^{\urcorner}{}^{\ulcorner}\lambda x.x^{\urcorner}N \\
&= \text{AND } (\text{AND } N^{\ulcorner}\lambda x.x^{\urcorner} N^{\ulcorner}\lambda x.x^{\urcorner})\, (\text{NOT}(\text{EQ } (P_1\, {}^{\ulcorner}\lambda x.x^{\urcorner})\, \overline{2})) \\
&= \text{AND } (\text{AND TRUE TRUE})\, (\text{NOT}(\text{EQ } (P_1\, \langle 2, \langle {}^{\ulcorner}x^{\urcorner}{}^{\ulcorner}x^{\urcorner}\rangle\rangle\, \overline{2}))) \\
&= \text{AND TRUE } (\text{NOT TRUE}) \\
&= \text{AND TRUE FALSE} \\
&= \text{FALSE}.
\end{aligned}
$$

3.4.2 Making the normal form function for Mogensen's coding

We can easily statisfy these equations using the recursion scheme for Mogensen's coding. We get:

$$
\begin{aligned}
A_1 &= \lambda xh.\text{TRUE} \\
A_2 &= \lambda mnh.\text{AND } (\text{AND } hm\, hn)\, (m(\lambda x.\text{TRUE})(\lambda mn.\text{TRUE})(\lambda m.\text{FALSE})) \\
A_3 &= \lambda mh.h\, m
\end{aligned}
$$

Note that $m(\lambda x.\text{TRUE})(\lambda mn.\text{TRUE})(\lambda m.\text{FALSE})$ will be true if the coded λ-term is is a variable or an application and it will be false if it is an abstraction. Then for Mogensen's codings we construct N:

$$
\begin{aligned}
N \equiv Y\lambda h.\lambda m.m(&\lambda x.A_1\, (\lambda abc.ax)\, h) \\
&(\lambda mn.A_2\, m\, n\, h) \\
&(\lambda m.\lambda v.A_3\, (m\, v)\, h)
\end{aligned}
$$

We can show that N indeed satisfies the equations.

$$
\begin{aligned}
N^{\ulcorner}x^{\urcorner} &= N\, \lambda abc.ax \\
&= A_1(\lambda abc.ax)N \\
&= (\lambda xh.\text{TRUE})(\lambda abc.ax)N \\
&= \text{TRUE}.
\end{aligned}
$$

$$
\begin{aligned}
N^{\ulcorner}A\, B^{\urcorner} &= N\, \lambda abc.b^{\ulcorner}A^{\urcorner}{}^{\ulcorner}B^{\urcorner} \\
&= A_2{}^{\ulcorner}A^{\urcorner}{}^{\ulcorner}B^{\urcorner}N \\
&= (\lambda mnh.\text{AND } (\text{AND } hm\, hn)\, (m(\lambda x.\text{TRUE})(\lambda mn.\text{TRUE})(\lambda m.\text{FALSE})))^{\ulcorner}A^{\urcorner}{}^{\ulcorner}B^{\urcorner}N \\
&= \text{AND } (\text{AND } N^{\ulcorner}A^{\urcorner} N^{\ulcorner}B^{\urcorner})\, ({}^{\ulcorner}A^{\urcorner}(\lambda x.\text{TRUE})(\lambda mn.\text{TRUE})(\lambda m.\text{FALSE}))
\end{aligned}
$$

$$\begin{aligned} N^{\ulcorner}\lambda x.M^{\urcorner} &= N\,\lambda abc.c(\lambda x.^{\ulcorner}M^{\urcorner}) \\ &= A_3{}^{\ulcorner}M^{\urcorner}N \\ &= (\lambda mh.h\,m)^{\ulcorner}M^{\urcorner}N \\ &= N^{\ulcorner}M^{\urcorner}. \end{aligned}$$

We will include some examples to show how N works.

Example 3.29.

$$\begin{aligned} N^{\ulcorner}K^{\urcorner} &= N^{\ulcorner}\lambda xy.x^{\urcorner} \\ &= N\,Abs(\lambda x.^{\ulcorner}\lambda y.x^{\urcorner}) \\ &= A_3{}^{\ulcorner}\lambda y.x^{\urcorner}N \\ &= (\lambda mh.h\,m)^{\ulcorner}\lambda y.x^{\urcorner}N \\ &= N^{\ulcorner}\lambda y.x^{\urcorner} \\ &= N\,Abs(\lambda y.^{\ulcorner}x^{\urcorner}) \\ &= A_3{}^{\ulcorner}x^{\urcorner}N \\ &= (\lambda mh.h\,m)^{\ulcorner}x^{\urcorner}N \\ &= N^{\ulcorner}x^{\urcorner} \\ &= N\,Var\,x \\ &= A_1{}^{\ulcorner}x^{\urcorner}N \\ &= (\lambda xh.\mathrm{TRUE})^{\ulcorner}x^{\urcorner}N \\ &= \mathrm{TRUE}. \end{aligned}$$

Example 3.30. *For the readability of the example of $N^{\ulcorner}II^{\urcorner}$, we first compute $N^{\ulcorner}\lambda x.x^{\urcorner}$*

$$\begin{aligned} N^{\ulcorner}\lambda x.x^{\urcorner} &= A_3{}^{\ulcorner}x^{\urcorner}N \\ &= N^{\ulcorner}x^{\urcorner} \\ &= \mathrm{TRUE} \end{aligned}$$

$$
\begin{aligned}
N\ulcorner II\urcorner &= N\ulcorner(\lambda x.x)\lambda x.x\urcorner \\
&= N\,App\ulcorner\lambda x.x\urcorner\ulcorner\lambda x.x\urcorner \\
&= A_2\ulcorner\lambda x.x\urcorner\ulcorner\lambda x.x\urcorner N \\
&= (\lambda mnh.\text{AND}\ (\text{AND}\ hm\,hn)\,(m(\lambda x.\text{TRUE}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda mn.\text{TRUE}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda m.\text{FALSE})))\ulcorner\lambda x.x\urcorner\ulcorner\lambda x.x\urcorner N \\
&= \text{AND}\ (\text{AND}\ N\ulcorner\lambda x.x\urcorner\,N\ulcorner\lambda x.x\urcorner)\,((\lambda abc.c(\lambda x.\ulcorner x\urcorner))(\lambda x.\text{TRUE}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda mn.\text{TRUE}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda m.\text{FALSE})) \\
&= \text{AND}\ (\text{AND TRUE TRUE})\,((\lambda m.\text{FALSE})(\lambda x.\ulcorner x\urcorner)) \\
&= \text{AND TRUE FALSE} \\
&= \text{FALSE}.
\end{aligned}
$$

3.5 A class of functions

The fact that D exists for Barendregt's coding but not for Mogensen's coding implies that there is class of functions that can be constructed for Barendegts coding but not for Mogensen's coding. We know that at least D is in that class. One important reason that this function is in this class, is that we can't test variable equality for Mogensen's coding, but we can for Barendregt's coding.

Lemma 3.31. There exists a function $F \in \Lambda$ such that for Barendregt's coding

$$
F\ulcorner x\urcorner\ulcorner y\urcorner \begin{cases} \text{TRUE}, & \text{if } x = y \\ \text{FALSE}, & \text{otherwise} \end{cases}
$$

Proof. Since Barendregt's coding is a Church numeral and each variable gets a unique number, simply take

$$
F \equiv \text{EQ}
$$

□

Lemma 3.32. There is no function $F \in \Lambda$ such that for Mogensen's coding

$$
F\ulcorner x\urcorner\ulcorner y\urcorner \begin{cases} \text{TRUE}, & \text{if } x = y \\ \text{FALSE}, & \text{otherwise} \end{cases}
$$

Proof. Say F exists and $x \neq y$. That would impy

$$\begin{aligned}(\lambda x.F \ulcorner x \urcorner \ulcorner y \urcorner)y &= (\lambda x.F\,(Var\,x)\,(Var\,y))y \\ &= (F\,(Var\,y)\,(Var\,y)) \\ &= \text{TRUE} \\ (\lambda x.F \ulcorner x \urcorner \ulcorner y \urcorner)y &= (\lambda x.\text{FALSE})y \\ &= \text{FALSE}\end{aligned}$$

This is a contradiction because according to the Church-Rosser Theorem a λ-term has only one normal form and the existance of F implies otherwise. □

So even if we were able to produce a list of free variables for Mogensen's coding, which is questionable already on it's own, it would be impossible to derive the variables that occur twice.
The inclusion of F implies that there are many more functions in this class that need F. It's likely that there are other functions in this class as well, but which functions exactly, remains the question.

In general it seems like Mogensen's coding can only derive the general structure of a λ-term. Since the original variables and abstractions are preserved, it is hard to derive properties from them. Barendregt's coding on the other hand, supports these types of functions by completely encoding the variables and abstractions to church numerals. This makes them suited for operations such as comparison and modification.

Chapter 4

Related Work

A similar research has been conducted for the typed λ-calculus. Bruce, Cardelli and Pierce have compared the strengths and weaknesses of 4 different object encodings in typed λ-calculus[5]. This comparison is more oriented towards programming languages and uses the λ-calculus as a common basis.

The first one to map λ-terms to numerals was Kleene [7]. However, this way of coding and decoding was very complex.

A binary way of coding λ-terms was presented by Tromp[11]. He proved this coding useful for Algorithmic Information technology.

There has been research to efficiency of codings in the λ-calculus. Stump and Fu investigate the efficiency of different ways of coding natural numbers. They include the Church, Scott and Parigot way of coding and eventually include their own [10]. They compare the way of defining functions on the codings as well as the efficiency, which is measured in the size of the normal form of an encoded numeral.

A λ-expression could also be encoded as a graph, as Lamping did in his research. He uses this external representation for his optimal reduction algorithm[8].

Chapter 5

Conclusions

We analyzed and compared the codings of Barendregt and Mogensen. The major differences that we found are:

1. There exists no self-interpreter that works for free variables for Barendregt's coding, but there does exist one for Mogensen's coding.

2. There is a class of functions that cannot be constructed for Mogensen's coding, but can be constructed for Barendregt's coding.

These differences mean that we initially cannot prove properties on codings in general, since this might differ per way of coding. We can only require a coding to satisfy certain properties, and use those to prove other properties.

This also means that different codings might be suitable in different situations. We see that Mogensen's coding is more suited when we need to translate the coded λ-term back to the original λ-term, since Mogensen's self-interpreter can cover more λ-terms. Furthermore Barendregt's coding proves more useful when we need to extract properties about variables, that go deeper than just structure, from a λ-term.

Bibliography

[1] Church encoding. https://en.wikipedia.org/wiki/Church_encoding#Calculation_with_Church_numerals.

[2] Pairing function. https://en.wikipedia.org/wiki/Pairing_function.

[3] Henk Barendregt. Self-interpretations in lambda calculus. *J. Funct. Program.*, 1(2):229–233, 1991.

[4] Henk Barendregt and Erik Barendsen. Intoduction to lambda calculus, October 1994. http://www.nyu.edu/projects/barker/Lambda/barendregt.94.pdf.

[5] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In Martín Abadi and Takayasu Ito, editors, *Theoretical Aspects of Computer Software*, pages 415–438, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[6] Herman Geuvers. Self-interpretation in lambda calculus. 2015. Huygens College, http://www.cs.ru.nl/~herman/onderwijs/2015Reflection/lecture6.pdf.

[7] S. C. Kleene. λ -definability and recursiveness. *Duke Math. J.*, 2(2):340–353, 06 1936.

[8] John Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 16–30, New York, NY, USA, 1990. ACM.

[9] Torben Æ. Mogensen. Efficient self-interpretations in lambda calculus. *J. Funct. Program.*, 2(3):345–363, 1992.

[10] Aaron Stump and Peng Fu. Efficiency of lambda-encodings in total type theory. *Journal of Functional Programming*, 26:e3, 2016.

[11] John Tromp. Binary lambda calculus and combinatory logic. In Marcus Hutter, Wolfgang Merkle, and Paul M.B. Vitanyi, editors, *Kolmogorov Complexity and Applications*, number 06051 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Appendix A

Appendix

Proof. (A1) Say there exists a function F, that can distinguish the structure of λ-term. Take

$$F\,M \begin{cases} \overline{1}, & \text{if } M\text{is a variable} \\ \overline{2}, & \text{if } M\text{is an application} \\ \overline{3}, & \text{if } M\text{is an abstraction} \end{cases}$$

Then the existence of F would imply:

$$\begin{aligned} F((\lambda x.x)a) &=_\beta \overline{2} \\ F((\lambda x.x)a) &=_\beta F\,a \\ &=_\beta \overline{1} \end{aligned}$$

Which implies:

$$1 =_\beta 2$$

This is a contradiction thus F cannot exist. □

Proof. (A2, Wikipedia [2]) Let $\langle x, y\rangle = z$.
We define some intermediate values in the calculation:

$$w = x + y$$

$$t = \frac{1}{2}w(w+1) = \frac{w^2 + w}{2}$$

$$z = t + y$$

where t is the triangle number of w. If we solve the quadratic equation

$$w^2 + w - 2t = 0$$

for w as a function of t, we get

$$w = \frac{\sqrt{8t+1} - 1}{2}$$

which is a strictly increasing and continuous function when t is non-negative real. Since

$$t \leq z = t + y < t + (w+1) = \frac{(w+1)^2 + (w+1)}{2}$$

we get that

$$w \leq \frac{\sqrt{8t+1} - 1}{2} < w + 1$$

and thus

$$w = \Big\lfloor \frac{\sqrt{8z+1} - 1}{2} \Big\rfloor.$$

where $\lfloor\ \rfloor$ is the floor function. So to calculate x and y from z, we do:

$$w = \Big\lfloor \frac{\sqrt{8z+1} - 1}{2} \Big\rfloor$$

$$t = \frac{w^2 + w}{2}$$

$$y = z - t$$

$$x = w - y.$$

□

Proof. (A3, Vree[3]) We construct V as follows.

$$\begin{aligned} V = Y(\lambda vn.&\text{IF (ZERO } n) \\ &\text{THEN rev} \\ &\text{ELSE } (\lambda Lx.v(\text{PRED } n)(x : L))) \end{aligned}$$

Then $F := \lambda n.V\, n\, \text{nil}$ indeed satisfies the equation from Lemma 3.12.

$$\begin{aligned} F\,\overline{n}\,x_1 \ldots x_n &= V\,\overline{n}\,\text{nil}\,x_1 \ldots x_n \\ &= (\lambda Lx.V\,\overline{n-1}\,(x : L))\,\text{nil}\,x_1 \ldots x_n \\ &= V\,\overline{n-1}\,(x_1 : \text{nil})x_2 \ldots x_n \\ &= (\lambda Lx.V\,\overline{n-2}\,(x : L))\,(x_1 : nil)\,x_2 \ldots x_n \\ &= V\,\overline{n-2}\,(x_2 : x_1 : \text{nil})\,x_3 \ldots x_n \\ &\ldots \\ &= V\,\overline{0}\,(x_n : \ldots : x_1 : \text{nil}) \\ &= \text{rev}\,\langle x_n, \ldots, x_1 \rangle \\ &= \langle x_1, \ldots, x_n \rangle. \end{aligned}$$

□