Bachelor's Thesis Computing Science

Comparative Analysis of Encodings in Lambda Calculus

Exploring Mogensen's and Barendregt's Encodings through Self-Interpretation

Jens de Reus
s1124054

June 8, 2026

First supervisor/assessor:
Prof. dr. J.H. Geuvers

Second assessor:
Prof. dr. R.J. Krebbers

Radboud University

Abstract

This thesis presents a comparative analysis of two encodings in the lambda calculus: Mogensen's encoding and Barendregt's encoding. The goal is to understand how these encodings represent syntax internally, and how their structural differences affect their capabilities in creating functions about properties on encoded lambda terms. In particular, it studies the possibility of computing the normal form in the Mogensen encoding and aims to improve the understanding of self-interpretation in Barendregt's encoding.

Contents

Chapter 1

Introduction

Lambda calculus is a formal system in theoretical computing science that models computation using only functions. Introduced by Alonzo Church in the 1930s, it was designed to capture the notion of computation in a precise and minimal setting, long before modern computers existed. The system is Turing-complete, which means that it can compute any function that a Turing machine can compute [1]. Its syntax consists of only three basic constructs: variables, application, and abstraction. Despite this simplicity, many familiar concepts such as numbers, booleans, and recursion can be encoded within the system.

Lambda calculus is also one of the theoretical foundations of functional programming. More importantly for this thesis, it can be viewed as a rewriting system in which terms are reduced step by step until they reach a normal form if one exists. A central limitation of ordinary lambda terms is that they cannot directly refer to their syntax. This means that we cannot study properties of lambda terms internally. We therefore need encodings: representations of lambda terms as data inside lambda calculus itself.

In this thesis, we study two well-known encodings: Mogensen's encoding [10] and Barendregt's encoding [3]. Although both represent lambda terms internally, they do so in fundamentally different ways. These differences affect the kinds of functions that can be defined on encoded terms, as well as the way self-interpretation and normalization can be expressed.

To make these differences concrete, we consider five problems about lambda terms.

1. Counting the number of free variables in a term.
2. Computing the normal form of a term, when it exists.

3a. Decoding an encoded term back into a lambda term for open lambda terms.

3b. Decoding an encoded term back into a lambda term for closed lambda terms.

4. Distinguishing open terms from closed terms.

We use these problems to compare Barendregt and Mogensen's encodings side by side. This also brings us to the *research question*:

What are the structural and expressive differences between Barendregt and Mogensen's encodings, and how do these differences affect normalization and self-interpretation?

At the end of the thesis, it will be clear what these differences are, why they are there, and what this means for future problems. For now we show the results in Table 1.

Problem	Barendregt encoding	Mogensen encoding
Counting # of different free variables	Possible: there is a function that counts the free variables of a term	Impossible: no function can count free variables for all encoded terms
Computing the normal form	Easy: a normalization function is straightforward to define	Possible: there is a normalization function, but its construction is non-trivial (main result of this thesis)
Decoding open λ-terms	Impossible: open terms cannot be decoded from the self-interpeter	Possible: all lambda terms can be decoded from their encoding, this includes open terms
Decoding closed λ-terms	Possible: closed can be decoded using the self-interpreter	Possible: the fact that open terms can be decoded implies that closed terms can also be decoded
Distinguishing open from closed λ-terms	Possible: there is a function that decides whether a term is closed	Open problem: currently unknown; we conjecture that no such function exists

Table 1.1: Capabilities of Barendregt and Mogensen's encodings on five problems.

The rest of this thesis returns to each row and justifies it more formally. Here we give a brief technical overview of what is going on in each case.

Counting free variables

In Barendregt's encoding, every variable is represented by a distinct natural number, and abstractions store the code of their bound variable explicitly.

$$\ulcorner x_i \urcorner := \langle 0, i \rangle,$$
$$\ulcorner \lambda x.M \urcorner := \langle 2, \langle \ulcorner x \urcorner, \ulcorner M \urcorner \rangle \rangle$$

Using the recursion scheme, one can traverse through an encoded term while maintaining a list of currently bound variables and thus count how many variable occurrences are free.

In the Mogensen's encoding, something else happens, because of the way an abstraction is defined:

$$\ulcorner \lambda x.M \urcorner := \mathbf{Abs}\ (\lambda x. \ulcorner M \urcorner).$$

As you can see, when we go under the abstraction, there is no way to know if a variable is open or closed. This means that we also cannot compute the number of free variables.

Computing the normal form

For Barendregt's encoding, it is relatively straightforward to define a normalization function. The numeric code tells us exactly where the applications and abstractions are, so we can follow a relatively easy reduction strategy (using, for example, leftmost reduction) just by doing recursion over this number-based representation.

In the Mogensen encoding, the situation is more subtle. In Chapter 4 we will construct a term G that repeatedly applies a leftmost reduction function L to an encoded term. However, this is not the difficult part. To make a leftmost reduction function, we need a substitution function S that has the following property.

$$S((\lambda x. \ulcorner M \urcorner) \ulcorner N \urcorner) =_\beta \ulcorner M[x := N] \urcorner$$

This might look like an easy function to define; however, we **cannot** simply say that:

$$\begin{aligned} S((\lambda x.\ulcorner M\urcorner)\ulcorner N\urcorner) &=_\beta (\lambda x.\ulcorner M\urcorner)\ulcorner N\urcorner) \\ &=_\beta \ulcorner M[x := N]\urcorner. \end{aligned}$$

The problem is that here, you would replace all occurrences of x by $\ulcorner N\urcorner$ in $\ulcorner M\urcorner$. However, that is not what you want, you actually want to replace $\ulcorner x\urcorner$ by $\ulcorner N\urcorner$. We need a clever trick to go around this, in Chapter 4, it will become clear that it is indeed possible to compute the normal form in Mogensen's encoding. The construction and correctness proof of this function is the central contribution of this thesis.

Decoding λ-terms

Mogensen's encoding is a strong encoding. Its self-interpreter works for all lambda terms, including those with free variables. This means its encoding preserves enough structure about abstractions and variable occurrences to reconstruct an open term from its encoding, yielding a decoder that works for both open and closed terms.

In Barendregt's encoding, we have a weak self-interpreter, meaning that it only works for closed terms. For both encodings, a self-interpreter exists; however, the explanation for Barendregt's encoding is minimal. In Chapter 3 we therefore provide a more detailed explanation of the self-interpreter.

Distinguishing open from closed λ-terms

As we have seen before, we can make a function that counts the number of free variables in the Barendregt encoding. This trivially means that we can also tell whether a term is open or closed.

For Mogensen's encoding, things are more difficult. Free variables cannot be inspected as easily. This would make it plausible that no function exists to distinguish open from closed terms. This remains an open problem, although we conjecture that such a function cannot be defined.

Chapter 2

Preliminaries

In this section, we introduce the basic notions of lambda calculus that will be used throughout the thesis. We define lambda terms, substitution, β-reduction, normal forms, and the distinction between free and bound variables. We also recall α-conversion and the standard reduction conventions needed later in the analysis of encodings.

Most of the definitions and theoretical foundations presented here follow Barendregt's work *The Lambda Calculus: Its Syntax and Semantics* [2]. This text is the standard reference for lambda calculus syntax, semantics, and reduction theory, and we will use the notations and conventions below.

2.1 Basic notions

The pure lambda calculus consists of three kinds of expressions: variables, applications, and abstractions. Terms are built recursively from these constructors. Variables stand for names, applications represent the function applications, and abstractions represent the functions.

Definition 1. *[2] The set of all valid lambda terms is Λ. It is defined as*

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

Where V is the set of variables. This can be any variable from an infinite set, usually we use x, y, z, u, v, w, etc. Lambda terms are usually noted with a capital letter, like M, N, P, Q, R, etc. $(\Lambda\Lambda)$ represent the application. The first lambda term is the function and is applied to the second lambda term, which is the argument. To apply a function to an argument, we need an abstraction. This is the third case. This is a function that has the parameter(s) V, and a body Λ. A term needs to be of the form $(\lambda x.M)N$ for us to do a reduction. Here we replace all the free occurrences of x in M by N. This reduction step is called β-reduction. A term containing such a reducible expression is called a redex.

Definition 2. *A substitution is a function such that for every $M, N \in \Lambda$ it holds that $M[x := N]$ replaces every free variable x in M with N.*

To better understand how this works, we show an example.

Example 1. *Assuming we have the redex (reducible expression) $(\lambda x.x)y$, we can see this as $x[x := y]$ meaning that after applying the definition, we get as a result y. This happens because every free occurrence of x in x is substituted by y.*

To actually use this in practice, we need some relation.

Definition 3. *The binary relation $M \to_\beta N$ says that the term M reduces to the term N by reducing one of the reducible expressions (redexes) in M. The relation is defined inductively by the following rules.*

1. $(\lambda x.M)N \rightarrow_\beta M[x := N]$;
2. $M \rightarrow_\beta N \Rightarrow MP \rightarrow_\beta NP$;
3. $M \rightarrow_\beta N \Rightarrow PM \rightarrow_\beta PN$;
4. $M \rightarrow_\beta N \Rightarrow \lambda x.M \rightarrow_\beta \lambda x.N$.

*We use the binary relation $\rightarrow_\beta^*$ for reducing in **zero** or more steps and $\rightarrow_\beta^+$ for reducing in **one** or more steps.*

In some situations, it is convenient to consider two lambda terms that can be reduced to each other. To capture this notion, we use lambda convertibility, which is defined as follows.

Definition 4. *Written as $M =_\beta N$, the lambda terms M and N can be connected using zero or more β-reduction steps going in either direction. We call M and N lambda convertible.*

This relation will be used most frequently, since it allows one lambda term to be transformed into another lambda term by any number of reduction steps, in either direction.

Furthermore, we use the following notational conventions to ensure that lambda terms are read unambiguously.

1. Application is left associative
   MNP means $(MN)P$, not $M(NP)$.
2. Application has higher precedence than abstraction
   $\lambda x.MN$ means $\lambda x.(MN)$, not $(\lambda x.M)N$.
3. $\lambda xy.M$ is used as an abbreviation for $\lambda x.\lambda y.M$.

It may also be the case that a lambda term cannot be reduced, simply because there is no redex. Such a term is then in normal form.

Definition 5. *A lambda term M is in **normal form** if it does not contain a redex.*

When a lambda term contains more than one redex, the order in which these redexes are reduced may be chosen freely. By the Church-Rosser theorem, this choice does not affect whether a normal form can be reached. [9]

Theorem 2.1.1. *Church-Rosser theorem.*
If $M \rightarrow_\beta^* N_1$ and $M \rightarrow_\beta^* N_2$ then there exists a lambda term Q s.t. $N_1 \rightarrow_\beta^* Q$ and $N_2 \rightarrow_\beta^* Q$.

This also implies that a lambda term can have at most 1 normal form.

Corollary 2.1.1.1. *If $M =_\beta N$ and $M =_\beta P$ and $P \in NF$, then $N =_\alpha P$.*

Here $=_\alpha$ means alpha convertible, which will be explained later. Note that it is perfectly possible for a lambda term to not have a normal form.

Example 2. *If we take the lambda term $\omega := \lambda x.xx$, and apply it to itself, we get an infinite reduction sequence.*

$$\begin{aligned} \omega\omega &\equiv (\lambda x.xx)\omega \\ &\rightarrow_\beta \omega\omega \\ &\rightarrow_\beta \omega\omega \\ &\rightarrow_\beta \ldots \end{aligned}$$

However, a term that has a normal form may still result in an infinite reduction sequence and therefore fail to reach a normal form. This is because Church Rosser tells us that a normal form "can" be reached, however it does not say that a normal form "will" be reached. To avoid this issue and to obtain a deterministic strategy for reduction, we later introduce leftmost reduction.

In lambda calculus, a variable can be either free or bound. In a lambda term, the variables that are not bound by any enclosing lambda abstraction are free. All variables that are not free, are bound. These free variables represent parameters that must be provided from the outside when evaluating the term. Think of it like a function where these bound variables are the input.

Definition 6. *The set of free variables is defined as follows.*

$$\begin{aligned} FV(x) &= \{x\}, \\ FV(MN) &= FV(M) \cup FV(N), \\ FV(\lambda x.M) &= FV(M) \setminus \{x\}. \end{aligned}$$

A lambda term M is closed when $FV(M) = \phi$, i.e. all its variables are bound. When this is not the case, M is open. The variables in a lambda term can be renamed. This is called α-conversion, written as $=_\alpha$.

Example 3. *An example of α-conversion can be as follows.*

$$\lambda x.xy =_\alpha \lambda z.zk.$$

This is possible because not only the variable inside the abstraction is changed but also the binding λ. Two lambda terms are called α-equivalent if they can be converted using α-conversion.

Definition 7. *Two λ-terms are called **identical** when they are α-convertible, they are called **equal** if they can be β-reduced to the same λ-term.*

β-Reduction is not unique because there might be more than one redex that can be reduced. To have a unique reduction we define a **leftmost reduction** which reduces the leftmost redex in a lambda term. To be even more precise, the leftmost reduction is the redex where the λ is the most to the left. This is also known as the leftmost-outermost redex. An essential property is that if you keep reducing with the leftmost redex, you will find the normal form if it exists. This is called **complete**. Again, we need a relation to use this in practice.

Definition 8. *The binary relation for reducing one step with the leftmost redex is defined as follows.*

$$M \to_l^\beta N.$$

We show an example to make it more clear what a leftmost reduction does.

Example 4. *Take $(\lambda x.(\lambda y.M)N)P$. This term contains two redexes. Applying one reduction step using the leftmost reduction we get the following.*

$$(\lambda x.(\lambda y.M)N)P \to_l^\beta ((\lambda y.M)N)[x := P]$$

2.2 Functions

Lambda calculus is based on the idea that computation can be described entirely in terms of functions. Functions are the central building blocks of the system, where more complex objects are made by combining simple function-like lambda terms. This makes the lambda calculus both minimal and powerful. Even though it does not contain built-in booleans, numbers, or conditionals and loops, these can still be represented using functions alone. In later chapters, we will be using various functions. Here we define some of these functions.

To make this concrete, we first introduce a number of standard combinators that will be used throughout the thesis. These combinators form the basis for creating more complex functions. We begin with the combinators I, K, K_*, S, C and ω:

Definition 9. *The standard combinators are defined as follows.*

$$\begin{aligned} I &\equiv \lambda x.x; \\ K &\equiv \lambda xy.x; \\ K_* &\equiv \lambda xy.y; \\ S &\equiv \lambda xyz.xz(yz); \\ C &\equiv \lambda xyz.xzy; \\ \omega &\equiv \lambda x.xx. \end{aligned}$$

The combinators K and K^* can be used to represent the boolean values.

Definition 10. *We define*

$$\begin{aligned} \mathsf{True} &\equiv K; \\ \mathsf{False} &\equiv K_*. \end{aligned}$$

Using these definitions, conditionals can be expressed in lambda calculus as follows.

Definition 11.

$$\mathsf{If}\ L\ \mathsf{then}\ M\ \mathsf{else}\ N \equiv LMN;$$

$$\begin{aligned} \mathsf{Not}\ M &\equiv \mathsf{If}\ M\ \mathsf{then\ False\ else\ True} \\ &\equiv M\ \mathsf{False\ True}; \end{aligned}$$

$$M\ \mathsf{Or}\ N \equiv M\ \mathsf{True}\ N;$$

$$\mathsf{And}\ MN \equiv M\ (N\ \mathsf{True\ False})\ \mathsf{False}.$$

With the same idea, natural numbers can be represented as functions. One common encoding for this is given by the Church numerals.

Definition 12. *For each natural number n, the corresponding **Church numeral** c_n is defined by*

$$\begin{aligned} c_0 &\equiv \lambda fx.x; \\ c_1 &\equiv \lambda fx.fx; \\ c_2 &\equiv \lambda fx.f(fx); \\ c_n &\equiv \lambda fx.f^n x. \end{aligned}$$

Where f^n is the function f applied n times. We then define

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

These numerals allow us to define basic arithmetic operations. The key idea is that a numeral n represents the repeated application of a function f. In later sections, we will make use of these constructions when defining more advanced functions such as recursion schemes and encodings of lambda terms. For now we can already make some more advanced functions using these numerals.

Definition 13. *Starting with the **successor** function, which adds one extra application of the function f, and can therefore be defined as follows.*

$$\mathsf{Succ} \equiv \lambda nfx.f(nfx),$$

which results in

$$\mathsf{Succ}\,\overline{n} =_\beta \overline{n+1}.$$

Addition *combines two numerals by first applying one numeral and then the other. Equivalently, addition can also be written as repeated successor application.*

$$\mathsf{Add} \equiv \lambda mn.(n\,\mathsf{Succ})m,$$

which results in

$$\mathsf{Add}\,\overline{m}\,\overline{n} =_\beta \overline{m+n}.$$

Multiplication *is defined by iterating addition, or more directly by nesting function application.*

$$\mathsf{Mult} \equiv \lambda mnfx.m(nf)x.$$

This term applies the function f a total of $m*n$ times, and therefore represents multiplication like

$$\mathsf{Mult}\,\overline{m}\,\overline{n} =_\beta \overline{m*n}.$$

The ***predecessor*** *function is less direct, because normally with Church numerals we count upwards rather than downward.*

$$\mathsf{Pred} \equiv \lambda nfx.n(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u).$$

As you can see, this function is more difficult than the previous ones, however it is sufficient to understand that

$$\mathsf{Pred}\,\overline{n+1} =_\beta \overline{n}.$$

Lastly, we have the ***zero*** *function, which returns whether the input numeral is equal to zero or not. It is defined as follows.*

$$\mathsf{Zero} \equiv \lambda n.n(\lambda x.\mathsf{False})\,\mathsf{True},$$

which results in

$$\mathsf{Zero}\,\overline{0} =_\beta \mathsf{True},$$
$$\mathsf{Zero}\,\overline{n+1} =_\beta \mathsf{False}.$$

Since we now established that arithmetic functions can be expressed in lambda calculus, we can now define more advanced functions, including recursive ones.

2.3 Recursion and Fixed points

Some functions, such as Factorial or Fibonacci, refer to themselves during their definition. This is called recursion. Since lambda calculus does not contain recursion as a built-in feature, it must be encoded using other lambda terms. The standard way to do this is through a fixed-point combinator, which makes it possible for a function to refer to itself indirectly. A term X is called a fixed point of a function F if $FX =_\beta X$. This means that applying F to X returns X again. This leads to the notion of a fixed-point combinator.

Definition 14. *A lambda term Y is called a fixed-point combinator if, given any term F, it produces a fixed point of F. I.e. if for all terms F we have*

$$F(YF) =_\beta YF.$$

The term YF can be unfolded into $F(YF)$, which means that the function F receives its result as an argument. This is precisely what we want to express recursion in the lambda calculus. The property given in this definition can be achieved when taking Y to be the following.

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)).$$

Indeed, for any lambda term F, we have

$$\begin{aligned} YF &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F \\ &=_\beta (\lambda x.F(xx))(\lambda x.F(xx)) \\ &=_\beta F((\lambda x.F(xx))(\lambda x.F(xx))) \\ &=_\beta F(YF). \end{aligned}$$

Example 5. *As an example, consider the factorial function. We want to find an F such that it satisfies the following.*

$$\begin{aligned} &F\,\overline{0} =_\beta \overline{1}, \\ &F\,\overline{1} =_\beta \overline{1}, \\ &F\,\overline{n} =_\beta \overline{n} * F\,\overline{n-1} \qquad \text{for } n \geq 1. \end{aligned}$$

Instead of defining the factorial directly, we first define a function T that expects a function f representing the recursive call.

$$\begin{aligned} T \equiv \lambda f n.&\mathsf{If}\ (\mathsf{Zero}\, n) \\ &\mathsf{then} c_1 \\ &\mathsf{else}\ \mathsf{Mult}\, n(f(\mathsf{Pred}\, n)). \end{aligned}$$

The factorial function F can then be obtained as a fixed point of T, namely

$$F \equiv YT.$$

To show that this is correct and satisfies the equation of F, we need to make a case distinction on n. Starting with $n = 0$.

$$\begin{aligned} F\,\overline{0} &=_\beta T\,F\,\overline{0} \\ &\equiv (\lambda f n.\mathsf{If}\ (\mathsf{Zero}\, n) \\ &\qquad \mathsf{then}\ c_1 \\ &\qquad \mathsf{else}\ \mathsf{Mult}\, n(f(\mathsf{Pred}\, n)))F\,\overline{0} \\ &=_\beta \mathsf{If}\ (\mathsf{Zero}\,\overline{0}) \\ &\qquad \mathsf{then}\ c_1 \\ &\qquad \mathsf{else}\ \mathsf{Mult}\,\overline{0}(F(\mathsf{Pred}\,\overline{0})) \\ &=_\beta c_1 \\ &=_\beta \overline{1}. \end{aligned}$$

Now for the second base case $F\,\overline{1}$.

$$\begin{aligned}
F\,\overline{1} &=_\beta T\,F\,\overline{1} \\
&=_\beta \mathsf{If}\ (\mathsf{Zero}\,\overline{1}) \\
&\qquad\qquad \mathsf{then}\ c_1 \\
&\qquad\qquad \mathsf{else}\ \mathsf{Mult}\,\overline{1}(F(\mathsf{Pred}\,\overline{1})) \\
&=_\beta \mathsf{Mult}\,\overline{1}(F(\mathsf{Pred}\,\overline{1})) \\
&=_\beta \mathsf{Mult}\,\overline{1}(F(\overline{0})) \\
&=_\beta \mathsf{Mult}\,\overline{1}\,\overline{1} \\
&=_\beta \overline{1}.
\end{aligned}$$

And lastly for the case where $n > 1$.

$$\begin{aligned}
F\,\overline{n} &=_\beta T\,F\,\overline{n} \\
&=_\beta \mathsf{If}\ (\mathsf{Zero}\,\overline{n}) \\
&\qquad\qquad \mathsf{then}\ c_1 \\
&\qquad\qquad \mathsf{else}\ \mathsf{Mult}\,\overline{n}(F(\mathsf{Pred}\,\overline{n})) \\
&=_\beta \mathsf{Mult}\,\overline{n}(F(\mathsf{Pred}\,\overline{n})) \\
&\equiv \overline{n} * F\,\overline{n-1}.
\end{aligned}$$

This shows that with this fixed-point combinator Y, we can define a recursive function.

Chapter 3

Encodings in the Lambda Calculus

To study properties of lambda terms within the lambda calculus, the terms themselves must first be represented as data. An encoding provides such a representation by assigning to every lambda term a code that can be manipulated internally. This makes it possible to define functions on syntax, including things like self-interpreters and recursion schemes. The encodings in this thesis exploit this idea in different ways, and their differences are essential for the analysis carried out in this chapter.

Without encodings, lambda terms cannot directly be treated as data inside the lambda calculus. This means that functions inspecting the structure of a term, such as a function counting the number of bound variables, cannot be defined on ordinary lambda terms alone.

Lemma 3.0.1. *There is no λ-term F such that*

$$FM = \overline{n},$$

where n is the number of bound variables in M.

Proof. Suppose that F exists. That would imply that:

$$\begin{aligned} F(IK) &=_\beta \overline{3} \\ IK &=_\beta K \\ F(K) &=_\beta \overline{2}. \end{aligned}$$

Which would imply:

$$\overline{2} =_\beta \overline{3}.$$

However since both $\overline{2}$ and $\overline{3}$ are in normal form and the Church-Rosser theorem says that a λ-term can have only one normal form, this gives us a contradiction. This means that such an F cannot exist. □

To make such functions possible, one needs an encoding. Two well-known encodings that realize this idea are Mogensen's encoding and Barendregt's encoding. Although both represent lambda terms internally, they do so in fundamentally different ways, which leads to different possibilities for recursion and self-interpretation.

An encoding is an internal representation of lambda terms by lambda terms. More precisely, it assigns to each term M a code $\ulcorner M \urcorner$ that is itself in normal form. This allows terms to be treated as data inside the lambda calculus. An encoding is particularly useful when it supports a self-interpreter and a recursion scheme, since these properties make it possible to define functions on the encoded syntax itself. In other words, we want an encoding to have the following 2 properties.

- There exists a self-interpreter $E \in \Lambda$ such that for all $M \in \Lambda^0$

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

- There exists a recursion scheme, such that given the right lambda terms $A_1, A_2, A_3 \in \Lambda$, there exists an $H \in \Lambda$, such that we can create recursive functions.

The latter is relatively vague since we did not define a recursion scheme yet. For both the encodings that we study in this thesis, there is a different recursion scheme, as can be seen in Theorem 3.1.1 for the Mogensen encoding and in Theorem 3.2.1 for the Barendregt encoding.

In the remainder of this chapter, the term "encoding" will refer to a representation following this definition.

As mentioned before, not every encoding has the same expressive power. In particular, some encodings can interpret only closed lambda terms, while others can also handle open terms. This difference is important, because free variables play a central role in many lambda terms and in the encodings that are studied in this thesis. For this reason, it is useful to make a distinction between weak encodings and strong encodings.

Definition 15. *A **weak encoding** is an encoding for which there exists a self-interpreter $E_0 \in \Lambda$ such that for all $M \in \Lambda^0$*

$$E_0{}^{\ulcorner}M^{\urcorner} =_\beta M,$$

where Λ^0 represents all closed lambda terms.

Definition 16. *A **strong encoding** is an encoding for which there exists a self-interpreter $E \in \Lambda$ such that for all $M \in \Lambda$*

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

This indicates that a weak encoding is sufficient when only closed terms are considered, but it does not necessarily preserve or reconstruct free variables. A strong encoding is more demanding, it must work for all lambda terms, including those with free variables. Hence, every strong encoding is weak, but not every weak encoding is strong.

This distinction will be important in the comparison between Mogensen's encoding and Barendregt's encoding, since they differ in how they handle free variables and self-interpreation.

We now turn to the first concrete encoding considered in this thesis. Mogensen's encoding represents lambda terms directly by their syntactic structure, which makes it a natural starting point for studying self-interpretation and recursion schemes.

3.1 Mogensen's encoding

The encoding is often referred to as the Mogensen-Scott encoding [10], since it represents terms by a Scott-style case analysis over the three constructors of the lambda calculus: variables, applications, and abstractions. Each of the three is mapped to a corresponding constructor in the encoding.

However, before we can define these three constructors in the encoding, we need a new function U_i^k.

Definition 17. *Given $M_1, \ldots, M_k$, define*

$$\langle M_1, \ldots, M_k \rangle := \lambda z.zM_1 \ldots M_k.$$

Define U_i^k, with $1 \leq i \leq k$ by

$$U_i^k := \lambda x_1 \ldots x_k.x_i.$$

Then we get

$$U_i^k M_1 \ldots M_k =_\beta M_i;$$
$$\langle M_1, \ldots, M_k \rangle U_i^k =_\beta U_i^k M_1 \ldots M_k = M_i.$$

To put it simply, U_i^k retrieves the i'th element of a list of size k. It can be easily shown with an example how this works.

Example 6. *Take*

$$U_1^3 \equiv \lambda xyz.x,$$

which gives us that

$$\begin{aligned}\langle M_1, M_2, M_3\rangle U_1^3 &=_\beta U_1^3 M_1 M_2 M_3\\ &=_\beta M_1.\end{aligned}$$

Now that we got U_i^k, we can define the encoding.

Definition 18. *The encoding of lambda terms inside λ-calculus is defined as follows [10].*

$$\begin{aligned}\ulcorner x \urcorner &:= \mathsf{Var}\, x,\\ \ulcorner MN \urcorner &:= \mathsf{App} \ulcorner M \urcorner \ulcorner N \urcorner,\\ \ulcorner \lambda x.M \urcorner &:= \mathsf{Abs}\, (\lambda x. \ulcorner M \urcorner).\end{aligned}$$

Where,

$$\begin{aligned}\mathsf{Var} &:= \lambda xe.eU_1^3 xe;\\ \mathsf{App} &:= \lambda xye.eU_2^3 xye;\\ \mathsf{Abs} &:= \lambda xe.eU_3^3 xe.\end{aligned}$$

Here $\ulcorner M \urcorner$ means the encoding of M.

With these constructors, we can define recursive functions such as self-interpreters, counters, or normal-form checkers. To achieve this, we first need a recursion scheme. As explained before, each encoding needs to have a recursion scheme. For Mogensen, we define this as follows.

Theorem 3.1.1. *Given $A_1, A_2, A_3 \in \Lambda$ there is an $H \in \Lambda$ such that*

$$\begin{aligned}H \ulcorner x \urcorner &=_\beta A_1\, x\, H;\\ H \ulcorner MN \urcorner &=_\beta A_2 \ulcorner M \urcorner \ulcorner N \urcorner H;\\ H \ulcorner \lambda x.M_0 \urcorner &=_\beta A_3 (\lambda x. \ulcorner M_0 \urcorner) H.\end{aligned}$$

Proof. Take $H = \langle\langle B_1, B_2, B_3\rangle\rangle$ with

$$\begin{aligned}B_1 &:= \lambda xz.A_1 x\langle z\rangle;\\ B_2 &:= \lambda xyz.A_2 xy\langle z\rangle;\\ B_3 &:= \lambda xz.A_3 x\langle z\rangle.\end{aligned}$$

Now we check for each case.

Var case:

$$\begin{aligned}
H(\mathsf{Var}\, x) &=_\beta \langle\langle B_1, B_2, B_3\rangle\rangle(\mathsf{Var}\, x)\\
&=_\beta (\lambda z.z\langle B_1, B_2, B_3\rangle)(\mathsf{Var}\, x)\\
&=_\beta (\mathsf{Var}\, x)\langle B_1, B_2, B_3\rangle\\
&=_\beta (\lambda e.e\mathbf{U_1^3}xe)\langle B_1, B_2, B_3\rangle\\
&=_\beta (\mathbf{U_1^3}\, B_1\, B_2\, B_3)\, x\, \langle B_1, B_2, B_3\rangle\\
&=_\beta B_1\, x\, \langle B_1, B_2, B_3\rangle\\
&=_\beta (\lambda x\, z.A_1 x\langle z\rangle)\, x\, \langle B_1, B_2, B_3\rangle\\
&=_\beta A_1\, x\, \langle\langle B_1, B_2, B_3\rangle\rangle\\
&=_\beta A_1\, x\, H.
\end{aligned}$$

App case:

$$\begin{aligned}
H(\mathsf{App}\, x\, y) &=_\beta \langle\langle B_1, B_2, B_3\rangle\rangle(\mathsf{App}\, x\, y)\\
&=_\beta (\lambda z.z\langle B_1, B_2, B_3\rangle)(\mathsf{App}\, x\, y)\\
&=_\beta (\mathsf{App}\, x\, y)\langle B_1, B_2, B_3\rangle\\
&=_\beta (\lambda e.e\mathbf{U_2^3}xye)\langle B_1, B_2, B_3\rangle\\
&=_\beta (\mathbf{U_2^3}\, B_1\, B_2\, B_3)\, x\, y\, \langle B_1, B_2, B_3\rangle\\
&=_\beta B_2\, x\, y\, \langle B_1, B_2, B_3\rangle\\
&=_\beta (\lambda x\, y\, z.A_2 xy\langle z\rangle)\, x\, y\, \langle B_1, B_2, B_3\rangle\\
&=_\beta A_2\, x\, y\, \langle\langle B_1, B_2, B_3\rangle\rangle\\
&=_\beta A_2\, x\, y\, H.
\end{aligned}$$

Abs case:

$$\begin{aligned}
H(\mathsf{Abs}\, x) &=_\beta \langle\langle B_1, B_2, B_3\rangle\rangle(\mathsf{Abs}\, x)\\
&=_\beta (\lambda z.z\langle B_1, B_2, B_3\rangle)(\mathsf{Abs}\, x)\\
&=_\beta (\mathsf{Abs}\, x)\langle B_1, B_2, B_3\rangle\\
&=_\beta (\lambda e.e\mathbf{U_3^3}xe)\langle B_1, B_2, B_3\rangle\\
&=_\beta (\mathbf{U_3^3}\, B_1\, B_2\, B_3)\, x\, \langle B_1, B_2, B_3\rangle\\
&=_\beta B_3\, x\, \langle B_1, B_2, B_3\rangle\\
&=_\beta (\lambda x\, z.A_3 x\langle z\rangle)\, x\, \langle B_1, B_2, B_3\rangle\\
&=_\beta A_3\, x\, \langle\langle B_1, B_2, B_3\rangle\rangle\\
&=_\beta A_3\, x\, H.
\end{aligned}$$

□

This yields a recursion scheme, with this, we can now construct recursive functions. For any given recursive function, it suffices to specify suitable A_1, A_2 and A_3 for the three cases. As a first example, we can use this scheme to define the self-interpreter. In the Mogensen encoding, this takes the following form.

Lemma 3.1.2. *[10] There exists a self-interpreter $E \in \Lambda$ such that for all $M \in \Lambda$*

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

We define,

$$\begin{aligned}
E(\mathsf{Var}\, x) &=_\beta x;\\
E(\mathsf{App}\, x\, y) &=_\beta E\, x\, (E\, y);\\
E(\mathsf{Abs}\, M) &=_\beta \lambda y.E\, (M\, y).
\end{aligned}$$

Proof. [7] Given the following E.

$$E \equiv \langle\langle K, S, C\rangle\rangle.$$

We can then fill this in in the different cases.
Var case:

$$\begin{aligned}
E(\mathsf{Var}\, x) &=_\beta \langle\langle K, S, C\rangle\rangle(\mathsf{Var}\, x) \\
&=_\beta (z.z\langle K, S, C\rangle)(\mathsf{Var}\, x) \\
&=_\beta (\mathsf{Var}\, x)\langle K, S, C\rangle \\
&=_\beta (\lambda e.eU_1^3 xe)\langle K, S, C\rangle \\
&=_\beta \langle K, S, C\rangle U_1^3 x\langle K, S, C\rangle \\
&=_\beta Kx\langle K, S, C\rangle \\
&=_\beta x.
\end{aligned}$$

App case:

$$\begin{aligned}
E(\mathsf{App}\ulcorner M\urcorner\ulcorner N\urcorner) &=_\beta \langle\langle K, S, C\rangle\rangle(\mathsf{App}\ulcorner M\urcorner\ulcorner N\urcorner) \\
&=_\beta (\mathsf{App}\ulcorner M\urcorner\ulcorner N\urcorner)\langle K, S, C\rangle \\
&=_\beta (\lambda e.eU_2^3\ulcorner M\urcorner\ulcorner N\urcorner e)\langle K, S, C\rangle \\
&=_\beta \langle K, S, C\rangle U_2^3\ulcorner M\urcorner\ulcorner N\urcorner\langle K, S, C\rangle \\
&=_\beta S\ulcorner M\urcorner\ulcorner N\urcorner\langle K, S, C\rangle \\
&=_\beta \ulcorner M\urcorner\langle K, S, C\rangle(\ulcorner N\urcorner\langle K, S, C\rangle) \\
&=_\beta^{\mathbf{IH}} M(N) \\
&=_\beta MN.
\end{aligned}$$

Abs case:

$$\begin{aligned}
E(\mathsf{Abs}\,(\lambda x.\ulcorner M\urcorner)) &=_\beta \langle\langle K, S, C\rangle\rangle(\mathsf{Abs}\,(\lambda x.\ulcorner M\urcorner)) \\
&=_\beta (\mathsf{Abs}\,(\lambda x.\ulcorner M\urcorner))\langle K, S, C\rangle \\
&=_\beta (\lambda e.eU_3^3(\lambda x.\ulcorner M\urcorner)e)\langle K, S, C\rangle \\
&=_\beta \langle K, S, C\rangle U_3^3(\lambda x.\ulcorner M\urcorner)\langle K, S, C\rangle \\
&=_\beta C(\lambda x.\ulcorner M\urcorner)\langle K, S, C\rangle \\
&=_\beta (\lambda xyz.xzy)(\lambda x.\ulcorner M\urcorner)\langle K, S, C\rangle \\
&=_\beta \lambda z.(\lambda x.\ulcorner M\urcorner)z\langle K, S, C\rangle \\
&=_\beta \lambda x.\ulcorner M\urcorner\langle K, S, C\rangle \\
&=_\beta^{\mathbf{IH}} \lambda x.M.
\end{aligned}$$

□

This proves that the Mogensen encoding supports self-interpretation in a very direct way and works for all lambda terms, including the open lambda terms. This answers problem 3a and 3b of Table 1 for the Mogensen encoding. The self-interpreter is important because it demonstrates that the encoded syntax can be read back and evaluated inside the lambda calculus itself. Since the encoding works for all lambda terms, including those with free variables, this also confirms that the Mogensen encoding is a strong encoding.

However, this is not the only approach. Another encoding is the one of Barendregt.

3.2 Barendregt's encoding

Barendregt's encoding differs fundamentally from Mogensen's and is closer to the classical Kleene-style approach to self-interpretation [8]. Lambda terms are represented as a unique numerical code via Gödel-style numbering, rather than recursive function structures.

Definition 19. *The encoding of lambda terms in Barendregt's encoding is defined as [3]*

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

where $\langle -, -\rangle$ is some encoding of pairs of numbers as a single number, this can, for example, be the Cantor pairing function [12].

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

Finally, we define $\ulcorner M \urcorner \equiv \#M$ where $\#M$ is a Church numeral. This means we can also write the Barendregt encoding as follows.

$$\begin{aligned} \ulcorner v^{(i)} \urcorner &:= \langle 0, i\rangle, \\ \ulcorner MN \urcorner &:= \langle 1, \langle \ulcorner M \urcorner, \ulcorner N \urcorner\rangle\rangle, \\ \ulcorner \lambda x.M \urcorner &:= \langle 2, \langle \ulcorner x \urcorner, \ulcorner M \urcorner\rangle\rangle. \end{aligned}$$

With an example, it will become more clear what such an encoding looks like.

Example 7. *Let $\#y \equiv \langle 0, 2\rangle$. The encoding of $\lambda y.y$ in Barendregt will be:*

$$\begin{aligned} \ulcorner \lambda y.y \urcorner &=_\beta \langle 2, \langle \#y, \#y\rangle\rangle \\ &=_\beta \langle 2, \langle \langle 0, 2\rangle, \langle 0, 2\rangle\rangle\rangle \\ &=_\beta \langle 2, \langle 6, 6\rangle \\ &=_\beta \langle 2, 84\rangle \\ &=_\beta \overline{3825}. \end{aligned}$$

Note that we use $\overline{n}$ instead of n to actually have a Church numeral instead of a natural number. We can also immediately see some differences. Here every variable is a different natural number. This means that we can see the difference between variables. Furthermore, if you look closely at the abstraction case, you can see that the variable is also encoded, whereas this was not the case with Mogensen. These small differences have a big effect in the possibilities with these encodings. These differences also mean that there is a different recursion scheme. Before making this recursion scheme, we need a function to retrieve data from the pairs.

Definition 20. *We define $P_1, P_2 \in \Lambda$ as follows [11].*

$$\begin{aligned} P_1\langle n, m\rangle &=_\beta n; \\ P_2\langle n, m\rangle &=_\beta m; \end{aligned}$$

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

To actually use these pair functions, we need to have a relation.

Definition 21.

$$\begin{aligned} \textsc{IsVar}\ (m) &\equiv P_1 m == 0; \\ \textsc{IsApp}\ (m) &\equiv P_1 m == 1; \\ \textsc{IsAbs}\ (m) &\equiv P_1 m == 2; \end{aligned}$$

where $==$ is defined as two natural numbers being equal.

Next we want to have a recursion scheme for the Barendregt encoding. This gives us the following theorem.

Theorem 3.2.1. *Given $A_1, A_2, A_3 \in \Lambda$ there is an $H \in \Lambda$ such that*

$$\begin{aligned} H\ulcorner x\urcorner &=_\beta A_1\ulcorner x\urcorner H; \\ H\ulcorner MN\urcorner &=_\beta A_2\ulcorner M\urcorner\ulcorner N\urcorner H; \\ H\ulcorner \lambda x.M_0\urcorner &=_\beta A_3\ulcorner x\urcorner\ulcorner M_0\urcorner H. \end{aligned}$$

Barendregt's encoding allows us to retrieve the abstraction x and the body M separately.

Proof. We can define H as follows

$$\begin{aligned} H \equiv Y(\lambda hm.&\mathsf{If}(\textsc{IsVar}\ m) \\ &\mathsf{then}\ (A_1 m h) \\ &\mathsf{else}\ (\\ &\quad \mathsf{if}\ (\textsc{IsApp}\ m) \\ &\quad \mathsf{then}\ (A_2(P_1(P_2 m))(P_2(P_2 m))h) \\ &\quad \mathsf{else}\ (A_3(P_1(P_2 m))(P_2(P_2 m))h) \\ &)). \end{aligned}$$

□

This scheme can now be used to make new functions. For these functions, we just need to give the right A_1, A_2, A_3.

From here we can also make a self-interpreter for the Barendregt encoding. Although this is a weak self-interpreter, it is still powerful. The following self-interpreter, made by de Bruin [3] is how Barendregt presented it.

Lemma 3.2.2. *There exists a self-interpreter E_0 such that*

$$E_0\ulcorner M\urcorner =_\beta M$$

for all $M \in \Lambda^0$ in Barendregt's encoding.

Proof. [3] There is a term E_0 such that

$$\begin{aligned} E_0\ulcorner x\urcorner F &=_\beta F\ulcorner x\urcorner; \\ E_0\ulcorner MN\urcorner F &=_\beta F(E_0\ulcorner M\urcorner F)(E_0\ulcorner N\urcorner F); \\ E_0\ulcorner \lambda x.M\urcorner F &=_\beta \lambda x.(E_0\ulcorner M\urcorner F_{\ulcorner x\urcorner \to x}); \end{aligned}$$

where $F_{\ulcorner x\urcorner \to x} \equiv F'x$, with

$$\begin{aligned} F'x\ulcorner x\urcorner &=_\beta x; \\ F'x\ulcorner y\urcorner &=_\beta \ulcorner y\urcorner, \text{if } x \not\equiv y. \end{aligned}$$

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

$$E_0\ulcorner M\urcorner F =_\beta M[x_1 := F\ulcorner x_1\urcorner, \ldots, x_n := F\ulcorner x_n\urcorner]$$

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

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

Indeed, for closed,M it now follows that

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

□

This is the self-interpreter in the Barendregt encoding. This tells us that in the Barendregt encoding we can decode closed lambda terms, however we cannot do the same for open lambda terms. This gives us an answer to problems 3a and 3b of Table 1.

The current self-interpreter might be difficult to understand. For that reason, we try to give a clearer explanation.

3.2.1 A deeper look into the self-interpreter of Barendregt

The original construction is not particularly easy to read, mainly because the role of the auxiliary function F' is not immediately clear. For that reason, this section presents a revised and more explicit version of the self-interpreter, with the aim of making its structure and operation easier to understand.

The key idea is that the interpreter is built in two parts. First, a recursive function S traverses the encoded term and processes its structure. Second, an auxiliary function H keeps track of bound variables and determines how variable occurrences should be interpreted. Together, these functions make it possible to interpret encoded lambda terms in a way that is both more transparent and easier to follow.

We first explain the intuition behind the construction, then define the functions formally, and finally show how the self-interpreter follows using this function.

Proposition 1. *There exists a self-interpreter E, in lambda calculus using Barendregt's encoding, for all $M \in \Lambda^0$ such that*

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

Definition 22. *We define E as follows.*

$$E \equiv \lambda m.SmI$$

Proof. Now we need to find an S such that it holds for the following lemma.

Lemma 3.2.3. *There exists some lambda term S such that for all $M \in \Lambda^0$ we have*

$$S\ulcorner M \urcorner F =_\beta M[x_1 := F\ulcorner x_1 \urcorner, ..., x_n := F\ulcorner x_n \urcorner].$$

Proof. Now to construct this S, it suffices to define a λ-term satisfying the following cases:

$$\begin{aligned}
S\ulcorner x \urcorner F &=_\beta F\ulcorner x \urcorner, \\
S\ulcorner MN \urcorner F &=_\beta (S\ulcorner M \urcorner F)(S\ulcorner N \urcorner F), \\
S\ulcorner \lambda x.M \urcorner F &=_\beta \lambda x.(S\ulcorner M \urcorner (H_F\ulcorner x \urcorner x)), \\
&\text{where,} \\
H_F\ulcorner x \urcorner z\ulcorner x \urcorner &=_\beta z, \\
H_F\ulcorner x \urcorner z\ulcorner y \urcorner &=_\beta F\ulcorner y \urcorner, \text{ if } \ulcorner x \urcorner \not\equiv \ulcorner y \urcorner.
\end{aligned}$$

□

The function H_F keeps track of the currently bound variable while the interpreter descends through the abstractions. When a variable occurrence is encountered, H_F compares its code with the last bound variable. If the codes match, the variable is returned directly, otherwise the variable is interpreted using the external function F. This means that we iterate through the list of bound variables and check whether any of them match the given code.

Now consider the term $\lambda xy.yx$. Its interpretation shows how the helper function H_F iterates through the list of bound variables. Starting from the outer abstraction we store x in H_F. Subsequently, we do the same for y. When we get to the body, we can, using H_F, see that both variables match with one of the stored binders.

Example 8.

$$\begin{aligned}
E\ulcorner\lambda xy.yx\urcorner &=_\beta S\ulcorner\lambda xy.yx\urcorner I \\
&=_\beta \lambda x.(S\ulcorner\lambda y.yx\urcorner(H_I\ulcorner x\urcorner x)) \\
&=_\beta \lambda xy.(S\ulcorner yx\urcorner(H_{H_I\ulcorner x\urcorner x}\ulcorner y\urcorner y)) \\
&=_\beta \lambda xy.(S\ulcorner y\urcorner(H_{H_I\ulcorner x\urcorner x}\ulcorner y\urcorner y))(S\ulcorner x\urcorner(H_{H_I\ulcorner x\urcorner x}\ulcorner y\urcorner y)) \\
&=_\beta \lambda xy.(H_{H_I\ulcorner x\urcorner x}\ulcorner y\urcorner y\ulcorner y\urcorner)(H_{H_I\ulcorner x\urcorner x}\ulcorner y\urcorner y\ulcorner x\urcorner) \\
&=_\beta \lambda xy.y(H_I\ulcorner x\urcorner x\ulcorner x\urcorner) \\
&=_\beta \lambda xy.yx.
\end{aligned}$$

As you can see in the definition of E, we chose that the auxiliary function F is chosen to be the identity function I. This is sufficient because closed terms contain no free variables, this means no additional environment needs to be added in this F. With this choice, the construction also simplifies considerably. This results in having the following property.

$$E\ulcorner M\urcorner =_\beta S\ulcorner M\urcorner I =_\beta M$$

□

This now yields the desired self-interpreter. We now formally define the functions S and H_F.

Definition 23. *We can satisfy the proof by defining $S \in \Lambda$ by a recursive case distinction on the code of the input term.*

$$\begin{aligned}
S \equiv Y(\lambda s\, m\, f.&\text{IF } (P_1 m == 0) \\
&\text{THEN } (f\, m) \\
&\text{ELSE } (\\
&\quad \text{IF } (P_1 m == 1) \\
&\quad \text{THEN } (s(P_1(P_2 m))f)(s(P_2(P_2 m))f) \\
&\quad \text{ELSE } (\lambda\, z.s\,(P_2(P_2 m))(H_f\,(P_1(P_2 m))\,z))) \\
&))
\end{aligned}$$

We can define H as follows.

$$H_f \equiv \lambda zxc.\text{IF } (\text{EQ } c\, z) \text{ THEN } x \text{ ELSE } f\, c$$

Which satisfies,

$$\begin{aligned}
&H_F\ulcorner x\urcorner z\ulcorner x\urcorner =_\beta x, \\
&H_F\ulcorner x\urcorner z\ulcorner y\urcorner =_\beta F\ulcorner y\urcorner, \qquad \textit{if } \ulcorner x\urcorner \not\equiv \ulcorner y\urcorner.
\end{aligned}$$

This completes the revised construction of the self-interpreter in the Barendregt encoding.

3.3 Helper Predicates

For later chapters, we define functions to make case distinctions on variables, applications, and abstractions. These work for both the Mogensen encoding and the Barendregt encoding.

Definition 24. *We define* IsAbs *as follows.*

$$\begin{aligned} \textsc{IsAbs}\ \ulcorner x \urcorner &=_\beta \mathsf{False} \\ \textsc{IsAbs}\ \ulcorner MN \urcorner &=_\beta \mathsf{False} \\ \textsc{IsAbs}\ \ulcorner \lambda x.M_0 \urcorner &=_\beta \mathsf{True} \end{aligned}$$

Definition 25. *We define* IsVar *as follows.*

$$\begin{aligned} \textsc{IsVar}\ \ulcorner x \urcorner &=_\beta \mathsf{True} \\ \textsc{IsVar}\ \ulcorner MN \urcorner &=_\beta \mathsf{False} \\ \textsc{IsVar}\ \ulcorner \lambda x.M_0 \urcorner &=_\beta \mathsf{False} \end{aligned}$$

Definition 26. *We define* IsApp *as follows.*

$$\begin{aligned} \textsc{IsApp}\ \ulcorner x \urcorner &=_\beta \mathsf{False} \\ \textsc{IsApp}\ \ulcorner MN \urcorner &=_\beta \mathsf{True} \\ \textsc{IsApp}\ \ulcorner \lambda x.M_0 \urcorner &=_\beta \mathsf{False} \end{aligned}$$

Remark. In the remainder of this thesis, we will freely use previously defined functions inside later constructions. Whenever we define a new recursive function for either encoding, it is understood that it is obtained by applying the corresponding recursion scheme with suitable terms A_1, A_2, and A_3.

Chapter 4

Analyzing encodings in the Lambda Calculus

4.1 Introduction

In this section, we study both the Mogensen and Barendregt encodings. We aim to define a function that computes the normal form of any lambda term. To do so, we first need to investigate how to determine whether a term is in normal form and whether it is a redex, since these notions are necessary before reduction can be carried out. Once these properties have been established, we will construct a function that computes the normal form.

4.2 Counting free variables

To answer problem 1 of Table 1, we show that counting the number of different free variables in a lambda term is possible in the Barendregt encoding but impossible in the Mogensen encoding.

Lemma 4.2.1. *There exists no function F in the Mogensen encoding such that*

$$F\ulcorner M\urcorner =_\beta \overline{n},$$

where $\overline{n}$ is the number of different free variables in M.

Proof. Suppose such a function F does exist. Then applying F to $\ulcorner xy\urcorner$ results in $\overline{2}$, since both x and y are free. However, because of the way that Mogensen encodes variables, we can get the following.

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

However, we could also reduce this differently.

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

This, however, is a contradiction, since the same term would reduce to both $\overline{1}$ and $\overline{2}$. Therefore, no such F can exist in the Mogensen encoding. □

For Barendregt, things are different since he uses a different way of encoding variables.

Lemma 4.2.2. *There exists a function F in the Barendregt encoding such that*

$$F\ulcorner M\urcorner =_\beta \overline{n},$$

where $\overline{n}$ is the number of different free variables in M.

Proof. The idea here is to use a helper function L. This function will create a list of all the free variables. When this list is returned to F, we can simply count the number of distinct variables in this list. To create this list, we need to keep track of the bound variables. In the Barendregt encoding, this is fairly simple since an abstraction is defined as follows.

$$\#(\lambda x.M) := \langle 2, \langle \#x, \#M \rangle \rangle$$

This means that we can simply retrieve the code of x in this case and add it to the list. Then we go through the term and check for each variable if it is in the list. In the case it is not, we will put it in a second list, which keeps track of all the free variables. The full proof is presented in the thesis of Nathan van Beusekom [11]. □

We have now shown that in the Mogensen encoding it is not possible to count the number of different free variables, while in the encoding of Barendregt, this is possible. This now gives us the results for part 1 in Table 1.

4.3 Distinguishing open from closed λ-terms

Next we want to address problem 5. The problem is here is that we want to, given an encoded lambda term, return whether that term is open. For Barendregt, this is quite trivial. As shown in section 4.2 we determined that there exists a function that can count the number of distinct free variables. To make this function, we used a helper function that returned a list with all free variables. To distinguish an open lambda term from a closed one, we simply check whether this list is empty or not. If the list is empty, we know the lambda term is closed, if not, it is open.

For Mogensen, things are more difficult. Looking at Section 4.2, you might conclude that it will be very difficult to find a function F such that it returns whether or not M is an open lambda term, since we cannot count the number of different free variables. It remains an open problem whether there exists a lambda term that can distinguish open from closed lambda terms in the Mogensen encoding. We conjecture that no such function is definable.

4.4 Computing the normal form

In this section, we address Problem 2 as described in Table 1, namely the construction of a function that computes the normal form of a given lambda term. To achieve this, we use a leftmost reduction strategy. This means that it is necessary to first analyze the components that are used in such a reduction process.

In particular, leftmost reduction relies on several key subprocedures:

- the contraction of a redex,
- the identification of a redex within a lambda term,
- and determining whether a lambda term is in normal form.

The following three subsections address these subproblems. Once these have been established, we proceed to construct a function that computes the normal form of a lambda term.

4.4.1 Reducing a redex

Reducing a redex with encodings works differently compared to standard lambda terms. This is because an encoded lambda term is already in normal form. This means that we cannot reduce an encoded lambda term the ordinary way, since a lambda term that is in normal form cannot be reduced. Meaning it is not possible to simply say $\ulcorner(\lambda x.M)N\urcorner =_\beta \ulcorner M[x := N]\urcorner$. Because of that we need to create a reduction function.

Lemma 4.4.1. *For all $M, N \in \Lambda$, we have*

$$\begin{aligned} H\ulcorner(\lambda x.M)N\urcorner &=_\beta \ulcorner M[x := N]\urcorner, \\ H\ulcorner M\urcorner &=_\beta \ulcorner M\urcorner \qquad\qquad \textit{otherwise.} \end{aligned}$$

In the Barendregt encoding this is quite straightforward. Take, for example $(\lambda x_i.x_i)N$.

$$\ulcorner(\lambda x_i.x_i)N\urcorner \equiv \langle 1, \langle\langle 2, \langle \ulcorner x_i\urcorner, \ulcorner x_i\urcorner\rangle\rangle, \ulcorner N\urcorner\rangle$$

Here we actually see the bound variable $\ulcorner x_i\urcorner$ as a Church numeral and we can change every occurrence of this numeral in x_i by N. Now we can easily make such a function in the Barendregt encoding. We define H as follows.

Definition 27.

$$\begin{aligned} H\ulcorner x\urcorner &=_\beta \ulcorner x\urcorner, \\ H\ulcorner xN\urcorner &=_\beta \ulcorner xN\urcorner, \\ H\ulcorner (MN)P\urcorner &=_\beta \ulcorner (MN)P\urcorner, \\ H\ulcorner (\lambda x.M)N\urcorner &=_\beta S\ulcorner x\urcorner\ulcorner M\urcorner\ulcorner N\urcorner, \\ H\ulcorner \lambda x.M\urcorner &=_\beta \ulcorner \lambda x.M\urcorner. \end{aligned}$$

Where S is defined as follows on the structure of M.

$$\begin{aligned} S\ulcorner x_i\urcorner\ulcorner x_j\urcorner\ulcorner N\urcorner &=_\beta \ulcorner N\urcorner \qquad \textit{If } x_i \equiv x_j \\ &=_\beta \ulcorner x_j\urcorner \qquad \textit{Otherwise,} \\ S\ulcorner x\urcorner\ulcorner PQ\urcorner\ulcorner N\urcorner &=_\beta \langle 1, \langle S\ulcorner x\urcorner\ulcorner P\urcorner\ulcorner N\urcorner, S\ulcorner x\urcorner\ulcorner Q\urcorner\ulcorner N\urcorner\rangle\rangle, \\ S\ulcorner x\urcorner\ulcorner \lambda y.M_0\urcorner\ulcorner N\urcorner &=_\beta \langle 2, \langle \ulcorner y\urcorner, S\ulcorner x\urcorner\ulcorner M_0\urcorner\ulcorner N\urcorner\rangle\rangle. \end{aligned}$$

S recursively goes through the lambda term M until it finds all occurrences of x and replaces them by N. This gives us the following lemma for S in the Barendregt encoding.

Lemma 4.4.2. *For all $x, M, N \in \Lambda$, we have that*

$$S\ulcorner x\urcorner\ulcorner M\urcorner\ulcorner N\urcorner =_\beta \ulcorner M[x := N]\urcorner.$$

Proof. We give a proof by case distinction on M.

$M \equiv x$

$$\begin{aligned} S\ulcorner x\urcorner\ulcorner x\urcorner\ulcorner N\urcorner &=_\beta \ulcorner N\urcorner \\ &\equiv \ulcorner x[x := N]\urcorner. \end{aligned}$$

$M \equiv y_i$

$$\begin{aligned} S\ulcorner x\urcorner\ulcorner y_i\urcorner\ulcorner N\urcorner &=_\beta \ulcorner y_i\urcorner \\ &\equiv \ulcorner y_i[x := N]\urcorner. \end{aligned}$$

$M \equiv PQ$

$$\begin{aligned} S\ulcorner x\urcorner\ulcorner PQ\urcorner\ulcorner N\urcorner &=_\beta \langle 1, \langle S\ulcorner x\urcorner\ulcorner P\urcorner\ulcorner N\urcorner, S\ulcorner x\urcorner\ulcorner Q\urcorner\ulcorner N\urcorner\rangle\rangle \\ &=_\beta^{\mathbf{IH}} \langle 1, \langle \ulcorner P[x := N]\urcorner, \ulcorner Q[x := N]\urcorner\rangle\rangle \\ &\equiv \ulcorner PQ[x := N]\urcorner. \end{aligned}$$

$M \equiv \lambda y.M_0$

$$\begin{aligned} S\ulcorner x\urcorner\ulcorner \lambda y.M_0\urcorner\ulcorner N\urcorner &=_\beta \langle 2, \langle \ulcorner y\urcorner, S\ulcorner x\urcorner\ulcorner M_0\urcorner\ulcorner N\urcorner\rangle\rangle \\ &=_\beta^{\mathbf{IH}} \langle 2, \langle \ulcorner y\urcorner, \ulcorner M_0[x := N]\urcorner\rangle\rangle \\ &\equiv \ulcorner \lambda y.M_0[x := N]\urcorner \\ &\equiv \ulcorner (\lambda y.M_0)[x := N]\urcorner. \end{aligned}$$

□

Now we prove Lemma 4.4.1 for the Barendregt encoding.

Proof. We do the proof by a case distinction on M. Note that in this proof we use Lemma 4.4.2 in each case.

$M \equiv x$

$$\begin{aligned} H(\ulcorner(\lambda x.x)N\urcorner) &=_\beta S\ulcorner x\urcorner\ulcorner x\urcorner\ulcorner N\urcorner \\ &=_\beta \ulcorner x[x := N]\urcorner. \end{aligned}$$

$M \equiv y_i$

$$\begin{aligned} H(\ulcorner(\lambda x.y_i)N\urcorner) &=_\beta S\ulcorner x\urcorner\ulcorner y_i\urcorner\ulcorner N\urcorner \\ &=_\beta \ulcorner y_i[x := N]\urcorner. \end{aligned}$$

$M \equiv PQ$

$$\begin{aligned} H(\ulcorner(\lambda x.PQ)N\urcorner) &=_\beta S\ulcorner x\urcorner\ulcorner PQ\urcorner\ulcorner N\urcorner \\ &=_\beta \ulcorner PQ[x := N]\urcorner. \end{aligned}$$

$M \equiv \lambda y.M_0$

$$\begin{aligned} H(\ulcorner(\lambda x.(\lambda y.M_0))N\urcorner) &=_\beta S\ulcorner x\urcorner\ulcorner \lambda y.M_0\urcorner\ulcorner N\urcorner \\ &=_\beta \ulcorner \lambda y.M[x := N]\urcorner. \end{aligned}$$

Note that by looking at the definition of H, it is trivial to say that for any other lambda term we have the property that $H\ulcorner M\urcorner =_\beta \ulcorner M\urcorner$. □

In the Mogensen encoding, it is more difficult to find such a function. We can easily define a function such that

$$H^{\ulcorner}(\lambda x.M)N^{\urcorner} =_\beta (\lambda x.{}^{\ulcorner}M^{\urcorner}){}^{\ulcorner}N^{\urcorner}$$

However this is not good enough, because $(\lambda x.{}^{\ulcorner}M^{\urcorner}){}^{\ulcorner}N^{\urcorner}$ is not beta-equal to ${}^{\ulcorner}M[x := N]^{\urcorner}$, which we can clearly see by the following example.

Example 9. *We take $(\lambda x.M)N$ where $M \equiv x$ and $N \equiv y$.*

$$\begin{aligned} H^{\ulcorner}(\lambda x.x)y^{\urcorner} &\equiv (\lambda x.{}^{\ulcorner}x^{\urcorner}){}^{\ulcorner}y^{\urcorner} \\ &=_\beta (\lambda x.\mathsf{Var}\, x)\mathsf{Var}\, y \\ &=_\beta \mathsf{Var}(\mathsf{Var}\, y) \end{aligned}$$

The problem here is that we have one Var to many. We want to return $\mathsf{Var}\, y$ instead of $\mathsf{Var}(\mathsf{Var}\, y)$. To actually reduce a redex in the Mogensen encoding, we define the following function.

Definition 28. *Take H to be,*

$$\begin{aligned} H^{\ulcorner}x^{\urcorner} &=_\beta {}^{\ulcorner}x^{\urcorner}, \\ H^{\ulcorner}xN^{\urcorner} &=_\beta {}^{\ulcorner}xN^{\urcorner}, \\ H^{\ulcorner}(MN)P^{\urcorner} &=_\beta {}^{\ulcorner}(MN)P^{\urcorner}, \\ H^{\ulcorner}(\lambda x.M)N^{\urcorner} &=_\beta S((\lambda x.{}^{\ulcorner}M^{\urcorner}\sigma){}^{\ulcorner}N^{\urcorner}), \\ H^{\ulcorner}\lambda x.M^{\urcorner} &=_\beta {}^{\ulcorner}\lambda x.M^{\urcorner}. \end{aligned}$$

Here σ is the substitution of any free variable y_i that we came across when recursively going through the lambda term. Later, when we define the function for the leftmost reduction, we see how this is constructed. For now we can assume, $\sigma := [y_1 := \mathsf{Var}\, y_1, y_2 := \mathsf{Var}\, y_2 \ldots, y_n := \mathsf{Var}\, y_n]$. We define S as follows.

$$\begin{aligned} S^{\ulcorner}x^{\urcorner} &=_\beta x, \\ S^{\ulcorner}MN^{\urcorner} &=_\beta \mathsf{App}\,(S^{\ulcorner}M^{\urcorner})\,(S^{\ulcorner}N^{\urcorner}), \\ S^{\ulcorner}\lambda x.M_0{}^{\urcorner} &=_\beta \mathsf{Abs}\,(\lambda y.(\lambda x.{}^{\ulcorner}M^{\urcorner}){}^{\ulcorner}y^{\urcorner}). \end{aligned}$$

This gives us the following lemma for S.

Lemma 4.4.3. *For all $M, N \in \Lambda$ with FV(M) $\subseteq \{x, y_1, \ldots, y_n\}$, we have*

$$S((\lambda x.{}^{\ulcorner}M^{\urcorner}\sigma){}^{\ulcorner}N^{\urcorner}) =_\beta {}^{\ulcorner}M[x := N]^{\urcorner}.$$

To understand why the combination of S and H work, and why we need σ, we look at 2 examples.

Example 10. *Take $(\lambda x.x)y$.*

$$\begin{aligned} H^{\ulcorner}(\lambda x.x)y^{\urcorner} &=_\beta S((\lambda x.{}^{\ulcorner}x^{\urcorner}\sigma){}^{\ulcorner}y^{\urcorner}) \\ &\equiv S((\lambda x.(\mathsf{Var}\, x)\sigma){}^{\ulcorner}y^{\urcorner}) \\ &=_\beta S((\mathsf{Var}\, x)\sigma[x := \mathsf{Var}\, y]) \\ &=_\beta S((\mathsf{Var}\, x)[x := \mathsf{Var}\, y]) \\ &=_\beta S(\mathsf{Var}(\mathsf{Var}\, y)) \\ &=_\beta \mathsf{Var}\, y. \end{aligned}$$

Here you can see that by first unfolding the encoding, we can apply the actual substitution. We then get back to our original problem, where we have one too many Var constructors. Here we use our newly constructed function S to remove one layer.

Now a new problem arises, whenever M is a free variable, we will not have the problem of having an extra layer. However, our function S still strips that layer. Because of that we need a substitution σ. Take a look at the following example.

Example 11. *Take $(\lambda x.y_i)y$. Here y_i is any free variable where $y_i \not\equiv x$. Without our substitution σ we would get the following.*

$$\begin{aligned} H^{\ulcorner}(\lambda x.y_i)y^{\urcorner} &=_\beta S((\lambda x.{}^{\ulcorner}y_i{}^{\urcorner}){}^{\ulcorner}y^{\urcorner}) \\ &\equiv S((\lambda x.(\mathsf{Var}\, y_i)){}^{\ulcorner}y^{\urcorner}) \\ &=_\beta S((\mathsf{Var}\, y_i)[x := \mathsf{Var}\, y]) \\ &=_\beta S(\mathsf{Var}\, y_i) \\ &=_\beta y_i. \end{aligned}$$

However, we want to return the encoding of the free variable. So we introduce $\sigma := [y_1 := \mathsf{Var}\, y_1, y_2 := \mathsf{Var}\, y_2, \ldots, y_n := \mathsf{Var}\, y_n]$. Now take a look at the following example where we use this.

Example 12. *Again, take $(\lambda x.y_i)y$. y_i is any free variable where $y_i \not\equiv x$. This time we actually use the definition of H as described above, where we use σ.*

$$\begin{aligned} H^{\ulcorner}(\lambda x.y_i)y^{\urcorner} &=_\beta S((\lambda x.{}^{\ulcorner}y_i{}^{\urcorner}\sigma){}^{\ulcorner}y^{\urcorner}) \\ &\equiv S((\lambda x.(\mathsf{Var}\, y_i)\sigma){}^{\ulcorner}y^{\urcorner}) \\ &=_\beta S((\mathsf{Var}\, y_i)\sigma[x := \mathsf{Var}\, y]) \\ &=_\beta S(\mathsf{Var}(\mathsf{Var}\, y_i)) \\ &=_\beta \mathsf{Var}\, y_i. \end{aligned}$$

We still need to give a formal proof for Lemma 4.4.1 for the Mogensen encoding and for Lemma 4.4.3.

Proof. We give a proof by case distinction on M.

$M \equiv x$

$$\begin{aligned} S((\lambda x.{}^{\ulcorner}x{}^{\urcorner}\sigma){}^{\ulcorner}N^{\urcorner}) &=_\beta S({}^{\ulcorner}x{}^{\urcorner}\sigma[x := {}^{\ulcorner}N^{\urcorner}]) \\ &=_\beta S({}^{\ulcorner}x{}^{\urcorner}[x := {}^{\ulcorner}N^{\urcorner}]) \\ &=_\beta S((\mathsf{Var}\, x)[x := {}^{\ulcorner}N^{\urcorner}]) \\ &=_\beta S(\mathsf{Var}\, {}^{\ulcorner}N^{\urcorner}) \\ &=_\beta {}^{\ulcorner}N^{\urcorner} \\ &=_\beta {}^{\ulcorner}x[x := N]{}^{\urcorner}. \end{aligned}$$

This works because $\{y_1, \ldots, y_n\} \notin FV(x)$.

$M \equiv y_i$

$$\begin{aligned} S((\lambda x.{}^{\ulcorner}y_i{}^{\urcorner}\sigma){}^{\ulcorner}N^{\urcorner}) &=_\beta S({}^{\ulcorner}y_i{}^{\urcorner}\sigma[x := {}^{\ulcorner}N^{\urcorner}]) \\ &=_\beta S({}^{\ulcorner}y_i{}^{\urcorner}[y_i := \mathsf{Var}\, y_i]) \\ &=_\beta S((\mathsf{Var}\, y_i)[y_i := \mathsf{Var}\, y_1]) \\ &=_\beta S(\mathsf{Var}\, (\mathsf{Var}\, y_i)) \\ &=_\beta \mathsf{Var}\, y_i \\ &=_\beta {}^{\ulcorner}y_i{}^{\urcorner} \\ &=_\beta {}^{\ulcorner}y_i[x := N]{}^{\urcorner}. \end{aligned}$$

This works because $x \notin FV(y_i)$, as $x \notin \{y_1, \ldots, y_n\}$.

$M \equiv PQ$

$$\begin{aligned}
S((\lambda x.\ulcorner PQ\urcorner\sigma)\ulcorner N\urcorner) &=_\beta S(\ulcorner PQ\urcorner\sigma[x := \ulcorner N\urcorner]) \\
&=_\beta S((\mathsf{App}\ulcorner P\urcorner\ulcorner Q\urcorner)\sigma[x := \ulcorner N\urcorner]) \\
&=_\beta S(\mathsf{App}\,(\ulcorner P\urcorner\sigma[x := \ulcorner N\urcorner])\,(\ulcorner Q\urcorner\sigma[x := \ulcorner N\urcorner])) \\
&=_\beta \mathsf{App}\,(S(\ulcorner P\urcorner\sigma[x := \ulcorner N\urcorner]))\,(S(\ulcorner Q\urcorner\sigma[x := \ulcorner N\urcorner])) \\
&=_\beta \mathsf{App}\,(S((\lambda x.\ulcorner P\urcorner\sigma)\ulcorner N\urcorner))\,(S((\lambda x.\ulcorner Q\urcorner\sigma)\ulcorner N\urcorner)) \\
&=^{\mathbf{IH}}_\beta \mathsf{App}\ulcorner P[x := N]\urcorner\ulcorner Q[x := N]\urcorner \\
&=_\beta \ulcorner PQ[x := N]\urcorner.
\end{aligned}$$

$M \equiv \lambda y.M_0$

$$\begin{aligned}
S((\lambda x.\ulcorner \lambda y.M_0\urcorner\sigma)\ulcorner N\urcorner &=_\beta S((\lambda x.\mathsf{Abs}\,(\lambda y.\ulcorner M_0\urcorner)\sigma)\ulcorner N\urcorner) \\
&=_\beta S(\mathsf{Abs}\,(\lambda y.\ulcorner M_0\urcorner)\sigma[x := \ulcorner N\urcorner]) \\
&=_\beta S(\mathsf{Abs}\,(\lambda y.\ulcorner M_0\urcorner\sigma[x := \ulcorner N\urcorner])) \\
&=_\beta \mathsf{Abs}\,(\lambda z.(S(\lambda y.\ulcorner M_0\urcorner\sigma[x := \ulcorner N\urcorner])(\mathsf{Var}\,z))) \\
&=_\beta \mathsf{Abs}\,(\lambda z.(S(\ulcorner M_0\urcorner\sigma[x := \ulcorner N\urcorner][y := \mathsf{Var}\,z]))) \\
&=_\beta \mathsf{Abs}\,(\lambda z.(S(\lambda x.\ulcorner M_0\urcorner\sigma[y := \mathsf{Var}\,z])\ulcorner N\urcorner)) \\
&=_\beta \mathsf{Abs}\,(\lambda y.(S(\lambda x.\ulcorner M_0\urcorner\sigma[y := \mathsf{Var}\,y])\ulcorner N\urcorner)) \\
&=_\beta \mathsf{Abs}\,(\lambda y.(S(\lambda x.\ulcorner M_0\urcorner\sigma_0)\ulcorner N\urcorner)) \\
&=^{\mathbf{IH}}_\beta \mathsf{Abs}\,(\lambda y.\ulcorner M_0[x := N]\urcorner) \\
&=_\beta \ulcorner \lambda y.M_0[x := N]\urcorner.
\end{aligned}$$

Where $\sigma_0 = [y_1 := \mathsf{Var}\,y_1, \ldots, y_n := \mathsf{Var}\,y_n, y := \mathsf{Var}\,y]$ which also works for the proof.

□

Next, we give a formal proof for H.

Proof. We make a proof by a case distinction on M. Note that in this proof we use Lemma 4.4.3 in each case.

$M \equiv x$

$$\begin{aligned}
H(\ulcorner(\lambda x.x)N\urcorner) &=_\beta S((\lambda x.(\mathsf{Var}\,x)\sigma)\ulcorner N\urcorner) \\
&=_\beta \ulcorner x[x := N]\urcorner.
\end{aligned}$$

$M \equiv y_i$

$$\begin{aligned}
H(\ulcorner(\lambda x.y_i)N\urcorner) &=_\beta S((\lambda x.(\mathsf{Var}\,y_i)\sigma)\ulcorner N\urcorner) \\
&=_\beta \ulcorner y_i[x := N]\urcorner.
\end{aligned}$$

$M \equiv PQ$

$$\begin{aligned}
H(\ulcorner(\lambda x.PQ)N\urcorner) &=_\beta S((\lambda x.\ulcorner PQ\urcorner\sigma)\ulcorner N\urcorner) \\
&=_\beta \ulcorner PQ[x := N]\urcorner.
\end{aligned}$$

$M \equiv \lambda y.M_0$

$$\begin{aligned}
H(\ulcorner(\lambda x.(\lambda y.M_0))N\urcorner &=_\beta S((\lambda x.\ulcorner \lambda y.M_0\urcorner\sigma)\ulcorner N\urcorner) \\
&=_\beta \ulcorner \lambda y.M[x := N]\urcorner.
\end{aligned}$$

Note that by looking at the definition of H it is again trivial to say that for any other lambda term we have the property that $H\ulcorner M\urcorner =_\beta \ulcorner M\urcorner$. □

This now shows that we can create a function that reduces a redex in both the Barendregt encoding and the Mogensen encoding.

4.4.2 Redex detection

Before we can reduce a redex, we first need to find it. For that we make a function that returns whether a lambda term is a redex.

Lemma 4.4.4. *There exists a function* IsRedex *such that*

$$\begin{aligned}\textsc{IsRedex}\ \ulcorner M\urcorner &=_\beta \mathsf{True} \text{ if } M \text{ is a redex}\\ &=_\beta \mathsf{False} \text{ if } M \text{ is not a redex}\end{aligned}$$

The idea is simple. We use a nested case analysis on the structure of M.

$$\begin{aligned}\textsc{IsRedex}\ \ulcorner x\urcorner &=_\beta \mathsf{False},\\ \textsc{IsRedex}\ \ulcorner(\lambda x.M)N\urcorner &=_\beta \mathsf{True},\\ \textsc{IsRedex}\ \ulcorner MN\urcorner &=_\beta \mathsf{False},\\ \textsc{IsRedex}\ \ulcorner\lambda x.M\urcorner &=_\beta \mathsf{False}.\end{aligned}$$

This function has no recursion. This means that this is relatively easy to implement. This can be done in both the Barendregt encoding and the Mogensen encoding. Let's remind ourselves that we have to find a A_1, A_2 and A_3 for our recursion scheme even though this function does not use recursion.

Proof. Take A_1, A_2 and A_3 to be,

$$\begin{aligned}A_1 &\equiv \lambda xh.\mathsf{False},\\ A_2 &\equiv \lambda mnh.\mathsf{If}\ \textsc{IsAbs}(m)\\ &\qquad\quad \mathsf{then\ True}\\ &\qquad\quad \mathsf{else\ False},\\ A_3 &\equiv \lambda xmh.\mathsf{False}.\end{aligned}$$

Note that we use a helper function IsAbs which will return whether M is an abstraction. □

Now that we know it is possible to determine whether M is a redex, it is also straightforward to define a function that detects whether M contains a redex. We do this by recursively going through a lambda term and checking if the term is a redex. We can even count the number of redexes.

Lemma 4.4.5. *There exists a function* RedexCount*, in lambda calculus using either Barendregt's or Mogensen's encoding, such that*

$$\textsc{RedexCount}\ \ulcorner M\urcorner =_\beta \overline{n}$$

where $\overline{n}$ is the number of redexes in M.

Proof. We define RedexCount as follows.

$$\begin{aligned}
\textsc{RedexCount}\ \ulcorner x \urcorner &=_\beta \overline{0},\\
\textsc{RedexCount}\ \ulcorner MN \urcorner &=_\beta \mathsf{If}\ \textsc{IsRedex}\ \ulcorner MN \urcorner\\
&\qquad \mathsf{then}\ \overline{1} + \textsc{RedexCount}\ \ulcorner M \urcorner + \textsc{RedexCount}\ \ulcorner N \urcorner\\
&\qquad \mathsf{else}\ \textsc{RedexCount}\ \ulcorner M \urcorner + \textsc{RedexCount}\ \ulcorner N \urcorner,\\
\textsc{RedexCount}\ \ulcorner \lambda x.M_0 \urcorner &=_\beta \textsc{RedexCount}\ \ulcorner M_0 \urcorner.
\end{aligned}$$

□

4.4.3 Checking normal form

The last step before computing the normal form, is that we need a function that returns whether a lambda term is in normal form. This gives us the following lemma.

Lemma 4.4.6.

$$\begin{aligned}
\textsc{IsNF}\ \ulcorner M \urcorner &=_\beta \mathsf{True}\ \textit{if } M \textit{ is in normal form}\\
&=_\beta \mathsf{False}\ \textit{if } M \textit{ is not in normal form.}
\end{aligned}$$

Proof. We prove by induction on the structure of the term. In the application case, we consider 2 cases. One where we come across a redex. This means that we use a nested case analysis on the structure of M.

$$\begin{aligned}
\textsc{IsNF}\ \ulcorner x \urcorner &=_\beta \mathsf{True},\\
\textsc{IsNF}\ \ulcorner (\lambda x.M)N \urcorner &=_\beta \mathsf{False},\\
\textsc{IsNF}\ \ulcorner MN \urcorner &=_\beta \textsc{AND}\ (\textsc{IsNF}\ \ulcorner M \urcorner)(\textsc{IsNF}\ \ulcorner N \urcorner),\\
\textsc{IsNF}\ \ulcorner \lambda x.M \urcorner &=_\beta \textsc{IsNF}\ \ulcorner M \urcorner.
\end{aligned}$$

We then define

$$\textsc{IsNF}\ \equiv Y(\lambda hm.mA_1A_2A_3).$$

Take A_1, A_2 and A_3 to be,

$$\begin{aligned}
A_1 &\equiv \lambda x.true,\\
A_2 &\equiv \lambda mn.\textsc{If}\ \neg\textsc{IsAbs}\ m\\
&\qquad\quad \textsc{Then}\ \textsc{AND}\ (hm)(hn)\\
&\qquad\quad \textsc{Else}\ false,\\
A_3 &\equiv \lambda m.hm.
\end{aligned}$$

□

4.4.4 Computing the normal form

Now that we know how to find a redex, how to reduce it, and how to determine whether a lambda term is in normal form, we can construct our function that computes the normal form. This gives us the following lemma.

Lemma 4.4.7. *There exists a lambda term G such that for all $M \in \Lambda$ we have,*

$$\begin{aligned} G\ulcorner M\urcorner &=_\beta \ulcorner nf(M)\urcorner && \text{if } M \text{ has a nf} \\ &=_\beta \infty && \text{if } M \text{ has no nf.} \end{aligned}$$

Definition 29. *We define G for all $M \in \Lambda$ as follows.*

$$\begin{aligned} G\ulcorner M\urcorner \equiv\ & \textsc{If isNF } \ulcorner M\urcorner \\ & \textsc{Then } \ulcorner M\urcorner \\ & \textsc{Else } G(L\ulcorner M\urcorner). \end{aligned}$$

Here L is a function that does one leftmost reduction step. This now clearly computes the normal form if we can find such a function L. As explained before, the hard part is actually doing a substitution. Now that we have a function for that property, we can make a function that does one leftmost reduction step, namely L, which should have the following property.

$$M \to^l_\beta M' \Leftrightarrow L\ulcorner M\urcorner =_\beta \ulcorner M'\urcorner.$$

In the Mogensen encoding, we can define this function as follows.

Definition 30.

$$\begin{aligned} L\ulcorner x\urcorner &=_\beta \ulcorner x\urcorner, \\ L\ulcorner \lambda x.M\urcorner &=_\beta \mathsf{Abs}\,(\lambda x.(\lambda x.L\ulcorner M\urcorner)\mathsf{Var}\,x) \\ &=_\beta \mathsf{Abs}\,(\lambda x.L\ulcorner M\urcorner[x := \mathsf{Var}\,x]), \\ L\ulcorner MN\urcorner &=_\beta \textsc{If isAbs } \ulcorner M\urcorner \\ &\quad \textsc{Then } H\ulcorner MN\urcorner \\ &\quad \textsc{Else if isNF } \ulcorner M\urcorner \\ &\qquad \textsc{Then App } \ulcorner M\urcorner (L\ulcorner N\urcorner) \\ &\qquad \mathsf{App}\,(L\ulcorner M\urcorner)\ulcorner N\urcorner. \end{aligned}$$

Here we can see the construction of σ in the abstraction case. For the Barendregt encoding, we can accomplish the same thing.

Definition 31.

$$\begin{aligned} L\ulcorner x\urcorner &=_\beta \ulcorner x\urcorner, \\ L\ulcorner \lambda x.M\urcorner &=_\beta \langle 2, \langle \ulcorner x\urcorner, L\ulcorner M\urcorner\rangle\rangle, \\ L\ulcorner MN\urcorner &=_\beta \textsc{If isAbs } \ulcorner M\urcorner \\ &\quad \textsc{Then } H\ulcorner MN\urcorner \\ &\quad \textsc{Else if isNF } \ulcorner M\urcorner \\ &\qquad \textsc{Then } \langle 1, \langle \ulcorner M\urcorner, L\ulcorner N\urcorner\rangle\rangle \\ &\qquad \textsc{Else } \langle 1, \langle L\ulcorner M\urcorner, \ulcorner N\urcorner\rangle\rangle. \end{aligned}$$

Note that we did not use the IsRedex function. However, we did use the logic behind that function, which is the IsAbs function on the first term of the application. This now gives us the following lemma.

Lemma 4.4.8. *For all $M, M' \in \Lambda$, we have*

$$M \to^l_\beta M' \Rightarrow L\ulcorner M\urcorner =_\beta \ulcorner M'\urcorner,$$

and

$$L\ulcorner M\urcorner =_\beta \ulcorner M'\urcorner \Rightarrow M \to^l_\beta M' \text{ or } M \in NF \text{ and } M \equiv M'.$$

Note that this lemma is less strong than what was previously stated. This is because when $M \in \text{NF}$, L still returns that same M, however it obviously does not hold that $M \to_l^\beta M$. We first give the proof for the Mogensen encoding.

Proof. Starting with the proof of

$$M \to_\beta^l M' \Rightarrow L\ulcorner M \urcorner =_\beta \ulcorner M' \urcorner.$$

We will do a proof by induction on M. A distinction will be made on the location of the leftmost redex.

I. $M \equiv (\lambda x.M_0)N$. By the definition of the leftmost reduction we know that

$$(\lambda x.M_0)N \to_\beta^l M_0[x := N].$$

Now applying L on M and using Lemma 4.4.1 we get

$$\begin{aligned} L\ulcorner(\lambda x.M_0)N\urcorner &=_\beta H\ulcorner(\lambda x.M_0)N\urcorner \\ &=_\beta \ulcorner M_0[x := N]\urcorner. \end{aligned}$$

II. $M \equiv PQ$ where there is a redex in P. Here we know that

$$PQ \to_\beta^l P'Q, \qquad P \to_\beta^l P'.$$

Given the inductive hypothesis that

$$P \to_\beta^l P' \Rightarrow L\ulcorner P\urcorner =_\beta \ulcorner P'\urcorner,$$

we get

$$\begin{aligned} L\ulcorner PQ\urcorner &=_\beta \mathsf{App}\,(L\ulcorner P\urcorner)\ulcorner Q\urcorner \\ &=_\beta^{\mathbf{IH}} \mathsf{App}\ulcorner P'\urcorner\ulcorner Q\urcorner \\ &=_\beta \ulcorner P'Q\urcorner. \end{aligned}$$

III. $M \equiv PQ$ where $P \in NF$ and there is a redex in Q. Here we know that

$$PQ \to_\beta^l PQ', \qquad Q \to_\beta^l Q',$$

using the inductive hypothesis we get

$$\begin{aligned} L\ulcorner PQ\urcorner &=_\beta \mathsf{App}\ulcorner P\urcorner(L\ulcorner Q\urcorner) \\ &=_\beta^{\mathbf{IH}} \mathsf{App}\ulcorner P\urcorner\ulcorner Q'\urcorner \\ &=_\beta \ulcorner PQ'\urcorner. \end{aligned}$$

IV. $M \equiv \lambda x.M_0$. We know that

$$\lambda x.M_0 \to_\beta^l \lambda x.M_0',$$

together with the inductive hypothesis we get,

$$\begin{aligned} L\ulcorner\lambda x.M_0\urcorner &=_\beta \mathsf{Abs}\,(\lambda x.L\ulcorner M_0\urcorner[x := \mathsf{Var}\,x]) \\ &=_\beta^{\mathbf{IH}} \mathsf{Abs}\,(\lambda x.\ulcorner M_0'\urcorner) \\ &=_\beta \ulcorner\lambda x.M_0'\urcorner. \end{aligned}$$

This completes the proof for the first part. Now let's prove the other way,

$$L\ulcorner M\urcorner =_\beta \ulcorner M'\urcorner \Rightarrow M \to_\beta^l M' \text{ or } M \in NF \text{ and } M \equiv M'.$$

We will prove by an induction on M. We have the inductive hypothesis: For all subterms N of M, we have

$$L\ulcorner N\urcorner =_\beta \ulcorner N'\urcorner \Rightarrow N \to_\beta^l N'.$$

$M \equiv x$ Starting with the base case. We have that $L\ulcorner x\urcorner =_\beta \ulcorner x\urcorner$. Any reduction $L\ulcorner x\urcorner =_\beta \ulcorner M'\urcorner$ must therefore satisfy $\ulcorner M'\urcorner =_\beta \ulcorner x\urcorner$ hence $M' \equiv x$. Then $x \in$ NF and $x \equiv x$ both hold trivially.

$M \equiv \lambda x.M_0$ Next we have the first inductive case. Suppose $L\ulcorner \lambda x.M_0\urcorner =_\beta \ulcorner M'\urcorner$. By definition of L this reduction must be of the form

$$\begin{aligned} L\ulcorner \lambda x.M_0\urcorner &=_\beta \mathsf{Abs}\,(\lambda x.L\ulcorner M_0\urcorner[x := \mathsf{Var}\,x]) \\ &=_\beta \mathsf{Abs}\,(\lambda x.\ulcorner M_0'\urcorner). \end{aligned}$$

The inductive hypothesis now gives us that $M_0 \to_\beta^l M_0'$. This, together with using the definition of leftmost reduction we get that $\lambda x.M_0 \to_\beta^l \lambda x.M_0'$.

$M \equiv PQ$ Again we assume that $L\ulcorner PQ\urcorner =_\beta \ulcorner M'\urcorner$. Then by definition of L this can be multiple things depending on the case.

- Starting with the redex being in P. In this case, we get

$$\begin{aligned} L\ulcorner PQ\urcorner &=_\beta \mathsf{App}\,(L\ulcorner P\urcorner)\ulcorner Q\urcorner \\ &=_\beta^{\mathbf{IH}} \mathsf{App}\ulcorner P'\urcorner\ulcorner Q\urcorner, \end{aligned}$$

  which gives us that $P \to_\beta^l P'$. This now gives us that $PQ \to_\beta^l P'Q$.
- The next case is that P is in normal form and that the leftmost index is in Q. Here we get

$$\begin{aligned} L\ulcorner PQ\urcorner &=_\beta \mathsf{App}\ulcorner P\urcorner(L\ulcorner Q\urcorner) \\ &=_\beta^{\mathbf{IH}} \mathsf{App}\ulcorner P\urcorner\ulcorner Q'\urcorner, \end{aligned}$$

  which gives us that $Q \to_\beta^l Q'$. This now gives us that $PQ \to_\beta^l PQ'$.
- The last case is that the leftmost redex is the outer redex. Using Lemma 4.4.1, we get

$$\begin{aligned} L\ulcorner(\lambda x.M_0)N\urcorner &=_\beta H\ulcorner(\lambda x.M_0)N\urcorner \\ &=_\beta \ulcorner M_0[x := N]\urcorner. \end{aligned}$$

  By IH on $M_0[x := N]$, we get that $M_0[x := N] \to_\beta^l M'$, and since $(\lambda x.M_0)N \to_\beta^l M_0[x := N]$, we can conclude that $(\lambda x.M_0)N \to_\beta^l M'$.

□

This concludes the proof of Lemma 4.4.8 which proves that L is a valid leftmost reduction function in the Mogensen encoding. Next we prove the same lemma for the Barendregt encoding.

Proof. Again starting with the proof of

$$M \to_\beta^l M' \Rightarrow L\ulcorner M\urcorner =_\beta \ulcorner M'\urcorner.$$

We will do a proof by induction on M. A distinction will be made on the location of the leftmost redex.

I. $M \equiv (\lambda x.M_0)N$. By the definition of the leftmost reduction we know that

$$(\lambda x.M_0)N \to^l_\beta M_0[x := N].$$

Now applying L on M and using Lemma 4.4.1 we get

$$\begin{aligned} L\ulcorner(\lambda x.M_0)N\urcorner &=_\beta H\ulcorner(\lambda x.M_0)N\urcorner \\ &=_\beta \ulcorner M_0[x := N]\urcorner. \end{aligned}$$

II. $M \equiv PQ$ where there is a redex in P. Here we know that

$$PQ \to^l_\beta P'Q, \qquad P \to^l_\beta P'.$$

Given the inductive hypothesis that

$$P \to^l_\beta P' \Rightarrow L\ulcorner P\urcorner =_\beta \ulcorner P'\urcorner,$$

we get

$$\begin{aligned} L\ulcorner PQ\urcorner &=_\beta \langle 1, \langle L\ulcorner P\urcorner, \ulcorner Q\urcorner\rangle\rangle \\ &=^{\mathbf{IH}}_\beta \langle 1, \langle \ulcorner P'\urcorner, \ulcorner Q\urcorner\rangle\rangle \\ &=_\beta \ulcorner P'Q\urcorner. \end{aligned}$$

III. $M \equiv PQ$ where $P \in NF$ and there is a redex in Q. Here we know that

$$PQ \to^l_\beta PQ', \qquad Q \to^l_\beta Q',$$

using the inductive hypothesis we get

$$\begin{aligned} L\ulcorner PQ\urcorner &=_\beta \langle 1, \langle \ulcorner P\urcorner, L\ulcorner Q\urcorner\rangle\rangle \\ &=^{\mathbf{IH}}_\beta \langle 1, \langle \ulcorner P\urcorner, \ulcorner Q'\urcorner\rangle\rangle \\ &=_\beta \ulcorner PQ'\urcorner. \end{aligned}$$

IV. $M \equiv \lambda x.M_0$. We know that

$$\lambda x.M_0 \to^l_\beta \lambda x.M_0',$$

together with the inductive hypothesis we get.

$$\begin{aligned} L\ulcorner \lambda x.M_0\urcorner &=_\beta \langle 2, \langle \ulcorner x\urcorner, L\ulcorner M_0\urcorner\rangle\rangle \\ &=^{\mathbf{IH}}_\beta \langle 2, \langle \ulcorner x\urcorner, \ulcorner M_0'\urcorner\rangle\rangle \\ &=_\beta \ulcorner \lambda x.M_0'\urcorner. \end{aligned}$$

This completes the proof for the first part. Now let's prove the other way,

$$L\ulcorner M\urcorner =_\beta \ulcorner M'\urcorner \Rightarrow M \to^l_\beta M' \text{ or } M \in NF \text{ and } M \equiv M'.$$

We will prove by an induction on M. We have the inductive hypothesis: For all subterms N of M, we have

$$L\ulcorner N\urcorner =_\beta \ulcorner N'\urcorner \Rightarrow N \to^l_\beta N'.$$

$M \equiv x$ Starting with the base case. We have that $L\ulcorner x\urcorner =_\beta \ulcorner x\urcorner$. Any reduction $L\ulcorner x\urcorner =_\beta \ulcorner M'\urcorner$ must therefore satisfy $\ulcorner M'\urcorner =_\beta \ulcorner x\urcorner$ hence $M' \equiv x$. Then $x \in$ NF and $x \equiv x$ both hold trivially.

$M \equiv \lambda x.M_0$ Next we have the first inductive case. Suppose $L\ulcorner \lambda x.M_0 \urcorner =_\beta \ulcorner M' \urcorner$. By definition of L this reduction must be of the form

$$\begin{aligned} L\ulcorner \lambda x.M_0 \urcorner &=_\beta \langle 2, \ulcorner x \urcorner, L\ulcorner M_0 \urcorner \rangle \rangle \\ &=_\beta \langle 2, \ulcorner x \urcorner, \ulcorner M_0' \urcorner \rangle \rangle. \end{aligned}$$

The inductive hypothesis now gives us that $M_0 \to_\beta^l M_0'$. This, together with using the definition of leftmost reduction we get that $\lambda x.M_0 \to_\beta^l \lambda x.M_0'$.

$M \equiv PQ$ Again we assume that $L\ulcorner PQ \urcorner =_\beta \ulcorner M' \urcorner$. Then by definition of L this can be multiple things depending on the case.

- Starting with the redex being in P. In this case, we get

$$\begin{aligned} L\ulcorner PQ \urcorner &=_\beta \langle 1, \langle L\ulcorner P \urcorner, \ulcorner Q \urcorner \rangle \rangle \\ &=_\beta^{\mathbf{IH}} \langle 1, \langle \ulcorner P' \urcorner, \ulcorner Q \urcorner \rangle \rangle, \end{aligned}$$

  which gives us that $P \to_\beta^l P'$. This now gives us that $PQ \to_\beta^l P'Q$.
- The next case is that P is in normal form and that the leftmost redex is in Q. Here we get

$$\begin{aligned} L\ulcorner PQ \urcorner &=_\beta \langle 1, \langle \ulcorner P \urcorner, L\ulcorner Q \urcorner \rangle \rangle \\ &=_\beta^{\mathbf{IH}} \langle 1, \langle \ulcorner P \urcorner, \ulcorner Q' \urcorner \rangle \rangle, \end{aligned}$$

  which gives us that $Q \to_\beta^l Q'$. This now gives us that $PQ \to_\beta^l PQ'$.
- The last case is that the leftmost redex is the outer redex. Using Lemma 4.4.1 we get,

$$\begin{aligned} L\ulcorner (\lambda x.M_0)N \urcorner &=_\beta H\ulcorner (\lambda x.M_0)N \urcorner \\ &=_\beta \ulcorner M_0[x := N] \urcorner. \end{aligned}$$

  By IH on $M_0[x := N]$, we get that $M_0[x := N] \to_\beta^l M'$, and since $(\lambda x.M_0)N \to_\beta^l M_0[x := N]$, we can conclude that $(\lambda x.M_0)N \to_\beta^l M'$.

□

This now tells us that it is indeed possible to construct a function that does one leftmost reduction in both the Mogensen encoding and the Barendregt encoding. The only step remaining is to proof correctness of G.

Lemma 4.4.9. *If $M \to_l M'$, then $G\ulcorner M \urcorner =_\beta G\ulcorner M' \urcorner$.*

Using the previous Lemma 4.4.8, we can make a proof by induction on M, again using the different cases of the location of the leftmost redex.

I. $M \equiv (\lambda x.M_0)N$. Suppose $(\lambda x.M_0)N \to_l M'$, then $M' \equiv M_0[x := N]$ and

$$\begin{aligned} G\ulcorner (\lambda x.M_0)N \urcorner &=_\beta G(L\ulcorner (\lambda x.M_0)N \urcorner) \\ &=_\beta G\ulcorner M_0[x := N] \urcorner \\ &=_\beta \ulcorner M_0[x := N] \urcorner \\ &=_\beta \ulcorner M' \urcorner. \end{aligned}$$

II. $M \equiv PQ$ where there is a redex in P. Suppose $PQ \to_l M'$, then $M' \equiv P'Q$ and

$$\begin{aligned} G\ulcorner PQ \urcorner &=_\beta G(L\ulcorner PQ \urcorner) \\ &=_\beta G\ulcorner P'Q \urcorner \\ &=_\beta \ulcorner P'Q \urcorner \\ &\equiv \ulcorner M' \urcorner. \end{aligned}$$

III. $M \equiv PQ$ where $P \in NF$ and there is a redex in Q. Suppose $PQ \to_l M'$, then $M' \equiv PQ'$ and

$$\begin{aligned} G\ulcorner PQ\urcorner &=_\beta G(L\ulcorner PQ\urcorner) \\ &=_\beta G\ulcorner PQ'\urcorner \\ &=_\beta \ulcorner PQ'\urcorner \\ &\equiv \ulcorner M'\urcorner. \end{aligned}$$

IV. $M \equiv \lambda x.M_0$ Suppose $\lambda x.M_0 \to_l M'$, then $M' \equiv \lambda x.M_0'$ and

$$\begin{aligned} G\ulcorner \lambda x.M_0\urcorner &=_\beta G(L\ulcorner \lambda x.M_0\urcorner) \\ &=_\beta G\ulcorner \lambda x.M_0'\urcorner \\ &=_\beta \ulcorner \lambda x.M_0'\urcorner \\ &\equiv \ulcorner M'\urcorner. \end{aligned}$$

Lemma 4.4.10. *If $M \in NF$, then $G\ulcorner M\urcorner =_\beta \ulcorner M\urcorner$.*

Proof. This lemma follows from the definition of G and Lemma 4.4.6. If M is in normal form, then $\ulcorner M\urcorner$ is returned. □

Lemma 4.4.11. *If M has nf N, then $G\ulcorner M\urcorner =_\beta \ulcorner N\urcorner$.*

Proof. From the fact that N is the normal form of M, we know that $M \to_l^\beta M_1 \to_l^\beta \cdots \to_l^\beta N$. Using Lemma 4.4.9 we know that $G\ulcorner M\urcorner =_\beta G\ulcorner M_1\urcorner =_\beta G\ulcorner N\urcorner$. Then using Lemma 4.4.10 and the fact that $N \in$ NF, we conclude that $G\ulcorner N\urcorner =_\beta \ulcorner N\urcorner$. Putting this together, we see that $G\ulcorner M\urcorner =_\beta \ulcorner N\urcorner$. □

Lemma 4.4.12. *If $G\ulcorner M\urcorner =_\beta \ulcorner N\urcorner$, then $N \in NF$.*

Proof. This lemma also follows from the definition of G. This is because the only way to get out of G, is by having an input that is in normal form. This means that N has to be in normal form. □

Together these proofs show us that in the case that M has a normal form, that G will find that normal form. Now we still need to prove that if M does not have a normal form, G will reduce infinitely many times.

Lemma 4.4.13. *If M has no normal form, then*

$$G\ulcorner M\urcorner \to^\beta G(L\ulcorner M\urcorner) \to^\beta G\ulcorner M_1\urcorner \to^\beta G(L\ulcorner M_1\urcorner) \to^\beta \dots$$

where $M \to_l^\beta M_1 \to_l^\beta M_2 \to_l^\beta \dots$.

Proof. We will make a proof by contradiction. Suppose there is some N for which

$$G\ulcorner M\urcorner =_\beta \ulcorner N\urcorner.$$

From Lemma 4.4.12 we get that $N \in$ NF. This gives us that after n calls of L, we get that $L\ulcorner M_n\urcorner =_\beta \ulcorner N\urcorner$. But also $L\ulcorner M_{n-1}\urcorner =_\beta \ulcorner M_n\urcorner$, $L\ulcorner M_{n-2}\urcorner =_\beta \ulcorner M_{n-1}\urcorner$ and so on. Now from Lemma 4.4.9 we get

$$M \to_l M_1 \to_l \cdots \to_l M_{n-1} \to_l M_n \to_l N.$$

But then by the definition of the leftmost reduction this means that $M =_\beta N$. However, we know that M has no normal form and that $N \in$ NF. This is a contradiction, which means that there cannot be such an N for which $G\ulcorner M\urcorner =_\beta \ulcorner N\urcorner$ if M does not have a normal form. This tells us that G will reduce infinitely because if we look at the definition of G, we can see that it only has 2 reduction options. It either returns another G or $\ulcorner N\urcorner$. Since we just showed that this second option is not possible, we know it has to reduce infinitely. □

This extensive proof now shows that Lemma 4.4.7 indeed holds, which means that it is indeed possible to find the normal form of any lambda term M in both the Barendregt and the Mogensen encoding.

4.4.5 Explanation of the solution

In this section we explain in more detail why this works. To achieve that, we first need to understand that one way to find a normal form is by using leftmost reduction. The leftmost reduction will guarantee finding the normal form if one exists. For our function G, we want to keep repeating these leftmost reductions until the normal form is found.

This is precisely what our function G does, it keeps calling the leftmost reduction function L until M is in normal form by checking for normal form in each iteration. Assuming we have a function L that returns one leftmost reduction step, G is a function that returns the normal form.

To make a function L that returns one leftmost reduction step, we need to understand how this works. To use this leftmost reduction we need to keep in mind that there are a few possibilities of what M can be. That is why we will be looking at the shape of M.

I. $(\lambda x.M_0)N$;

II. PQ where P is not an abstraction and there is a redex in P;

III. PQ where $P \in NF$, P is not an abstraction, and there is a redex in Q;

IV. $\lambda x.M_0$.

Looking closely at L we can see that each of these cases is implemented.

We can see that I. is the case where $\textsc{isAbs}\ \ulcorner \lambda x.M_0 \urcorner$ holds, so we return $H \ulcorner MN \urcorner$. For now we can assume that H is a function that will do the actual reduction step ($M \to M'$) by using the substitution function S. We apply the outer redex, which is precisely what we want.

Example 13. *Take M to be $(\lambda x.x)y$ so that we are in case I. Then using Lemma 4.4.1 we get.*

$$\begin{aligned} G\ulcorner(\lambda x.x)y\urcorner &=_\beta G(L\ulcorner(\lambda x.x)y\urcorner) \\ &=_\beta G(H\ulcorner(\lambda x.x)y\urcorner) \\ &=_\beta G(\ulcorner y\urcorner) \\ &=_\beta \ulcorner y\urcorner. \end{aligned}$$

In case II., we know that $\textsc{isAbs}\ \ulcorner P \urcorner$ does not hold because if it did hold, we would be in case I. So next we check if $\textsc{isNF}\ \ulcorner P \urcorner$ which is not the case because there is a redex in P. So we return $\mathsf{App}\,(L\ulcorner P\urcorner)\ulcorner N\urcorner$. This is again exactly what you want because the leftmost redex is in P, so you will now apply L on P.

Example 14. *Take M to be $((\lambda x.x)y)z$ so that we are in case II. By reusing Example 13 we get,*

$$\begin{aligned} G\ulcorner((\lambda x.x)y)z\urcorner &=_\beta G(L\ulcorner((\lambda x.x)y)z\urcorner) \\ &=_\beta G(\mathsf{App}\,(L\ulcorner(\lambda x.x)y\urcorner)\ulcorner z\urcorner) \\ &=_\beta G(\mathsf{App}\ulcorner y\urcorner\ulcorner z\urcorner) \\ &=_\beta G(\ulcorner yz\urcorner) \\ &=_\beta \ulcorner yz\urcorner. \end{aligned}$$

For case III., the same logic applies; however, now we know that $P \in NF$ which means that we do not run L on P but on Q this time because the leftmost redex is in Q. You can quickly see that an example here would be very similar to case II. Note again that there is not a redex on the top level because we then would be in case I.

Lastly, we have case IV., which is taken care of by the abstraction case of L. Here we look inside the abstraction because we know that the leftmost redex, if present, is inside M_0. So that is what we do, we apply L on M_0.

Example 15. *Now we take M to be $\lambda z.(\lambda x.x)y$ to have an outer abstraction for case IV.*

$$\begin{aligned} G\ulcorner\lambda z.(\lambda x.x)y\urcorner &=_\beta G(L\ulcorner\lambda z.(\lambda x.x)y\urcorner) \\ &=_\beta G(\mathsf{Abs}\,(\lambda z.L\ulcorner(\lambda x.x)y\urcorner[z := \mathsf{Var}\,z]) \\ &=_\beta G(\mathsf{Abs}\,(\lambda z.\ulcorner y\urcorner)) \\ &=_\beta G(\ulcorner\lambda z.y\urcorner) \\ &=_\beta \ulcorner\lambda z.y\urcorner. \end{aligned}$$

It should be clear now that the leftmost reduction function L returns one leftmost reduction step, assuming that H is correct.

H is actually a very simple function. We return M itself if it is not a redex, and if it is, we give it to the substitution function S where we have the following property.

Corollary 4.4.13.1. *For all $(\lambda x.M)N \in \Lambda^0$, we have*

$$S((\lambda x.\ulcorner M\urcorner)\ulcorner N\urcorner) =_\beta \ulcorner M[x := N]\urcorner.$$

Chapter 5

Related Work

Self-interpretation and encodings in the lambda calculus have been studied for a long time. Barendregt's paper is the classical starting point for this line of work. It develops the theory of the lambda calculus in depth and provides the background for later encodings and self-interpretation [2]. His later paper on self-interpretation gives a concise discussion of the idea that a lambda term can be interpreted internally in the lambda calculus. This is an important reference point for subsequent research on encodings and self-interpretation [3]. Together, these works show that self-interpretation is possible in the lambda calculus for closed lambda terms in the Barendregt encoding [2][3].

Mogensen's work is a major next step in the lambda calculus. In his paper *Efficient Self-interpretations in Lambda Calculus*, he presents a new encoding, as presented in this thesis, and uses it to construct a self-interpreter, a self-reducer, and a proof of the fixed-point theorem for the Mogensen encoding [10]. This paper and the one discussed from Barendregt are the basis of the construction of this thesis.

Berarducci and Böhm take a different but related approach in their work on self-interpretation having a normal form [4]. Their goal is not just to interpret lambda terms internally, but to do so in a way where one can obtain a self-interpreter that itself has a normal form. This is closely related to the question of how much structure can be preserved when terms are represented using an encoding.

Böhm, Piperno, and Guerrini continue this direction by studying how functions can be defined in the lambda calculus using normal forms [5]. Their work focuses on defining recursive functions. This is particularly relevant because it connects to the idea of lambda definability, and it uses a structured case analysis over encoded terms. Their approach builds the recursive functions directly from lambda-definable components, which is conceptually close to the way that encodings work [5].

Kleene's paper *λ-definability and recursiveness* is one of the classic works connecting lambda calculus with recursion theory [8]. It shows that lambda-definable functions and recursive functions are deeply related, which makes it important earlier work to later work on fixed points and encodings in the lambda calculus.

Geuvers's lecture slides provide a useful overview of self-interpretation in the lambda calculus and explains both the power and limitations of these techniques, which is clearly closely related to this thesis [6][7]. These slides are helpful since they make the underlying ideas used easier to understand. Similarly, van Beusekom's thesis on encodings in the lambda calculus discusses both the Barendregt and the Mogensen encoding together with the basics about lambda calculus. He shows a way to create a function that returns whether a lambda term is in normal form. Next to that he shows that in the Barendregt encoding, it is possible to count the different free variables,

whereas this is not possible in the Mogensen encoding [11]. This makes it a useful reference point for understanding how the theory about encodings works.

A related angle comes from Kozen's paper on Church-Rosser's theorem, which is not about encodings directly; however, it is relevant to the normalization functions created in this thesis [9]. Results about confluence and normal forms matter since the encodings depend on the fact that reduction behaves well enough to reason about a reduction strategy. In the same way, Abelson and Sussman's *Structure and Interpretation of Computer Programs* is not a lambda calculus paper; however, it is relevant in the broader sense that computation can be understood through application and abstraction [1]. It is therefore useful as a general background reference.

Taken together, these works show a clear progression. Barendregt provides the theoretical foundation for self-interpretation [3], Mogensen gives an efficient and compact encoding that works for open lambda terms as well [10]. Later lectures and theses revisit these constructions, clarify their limitations, and show more possibilities with encodings in the lambda calculus [7][11].

Chapter 6

Conclusions

In this thesis, we compared Mogensen's encoding and Barendregt's encoding. The central idea was how the different representations of syntax influence what can be defined internally, in particular regarding free variables, normalization, and self-interpretation. The results indicate that the two encodings support different kinds of reasoning about lambda terms, and that these differences are directly caused by the way that variables and abstractions are represented.

A first important conclusion is that the two encodings provide different information about variables. In Barendregt's encoding, variables are represented explicitly by numerical codes, meaning that each variable is assigned their own natural number. Abstractions also store the code of the bound variable. Because of this, functions can be defined that inspect variables and keep track of bound and free variables during recursion. This makes it possible to count the number of distinct free variables in a term and, as a direct consequence, to decide whether a term is open or closed. In Mogensen's encoding, this information is not available in the same explicit form. Since abstractions are represented as $\mathsf{Abs}\,(\lambda x.\ulcorner M \urcorner)$, the distinction between free and bound occurrences cannot be recovered in a direct way. This explains why counting free variables is impossible in the Mogensen encoding, and why distinguishing open from closed is conjectured to be impossible.

A second conclusion is about self-interpretation. Mogensen's encoding is a strong encoding, meaning that its self-interpreter works for all lambda terms, including open terms. This means that encoded terms preserve enough structure to reconstruct the original syntax, even if there are free variables in the term. Barendregt's encoding, by contrast, yields only a weak self-interpreter. It correctly decodes closed terms, but does not provide a decoder for open terms. This difference is fundamental.

The main technical contribution of this thesis is the normalization result for Mogensen's encoding. For both encodings, functions were defined for redex detection, redex counting, and normal form testing. With these components, a leftmost reduction function was constructed. In the Barendregt encoding, this construction is relatively straightforward, because substitution can be defined directly on the numerical representation of the variables. In the Mogensen encoding, the situation is considerably more subtle. Here, substitution cannot be obtained by a direct replacement in the encoding because one must replace encoded variables $\ulcorner x \urcorner$, rather than ordinary variables x. We resolved this by introducing a substitution function S, and a systematic method for handling free variables through the substitution σ. This yields in a function G that computes the normal form of any lambda term whenever a normal form exists and reduces infinitely otherwise. It works by repeatedly calling, L which does exactly one leftmost reduction step. This establishes that normalization is possible in both encodings.

A further contribution is the reformulation of the self-interpreter for the Barendregt encoding. While Barendregt's original presentation gives a correct self-interpreter for a closed term, the role of

the auxiliary environment F is not immediately clear. In this thesis, that construction is rewritten using the function H_F, making it more explicit how bound variables are tracked during the process of recursively going through the encoded term. This does not change the expressive power of the encoding, but it does improve the clarity of the interpreter and it makes the construction easier to follow.

The results can be summarized as follows.

Problem	**Barendregt encoding**	**Mogensen encoding**
Counting # of different free variables	Possible: there is a function that counts the free variables of a term	Impossible: no function can count free variables for all encoded terms
Computing the normal form	Easy: a normalization function is straightforward to define	Possible: there is a normalization function, but its construction is non-trivial (main result of this thesis)
Decoding open λ-terms	Impossible: open terms cannot be decoded from the self-interpreter	Possible: all lambda terms can be decoded from their encoding, this includes open terms
Decoding closed λ-terms	Possible: closed can be decoded using the self-interpreter	Possible: the fact that open terms can be decoded implies that closed terms can also be decoded
Distinguishing open from closed λ-terms	Possible: there is a function that decides whether a term is closed	Open problem: currently unknown; we conjecture that no such function exists

Table 6.1: Capabilities of Barendregt and Mogensen's encodings on five problems.

These results indicate that either encoding has its own pros and cons. Barendregt's encoding is stronger when explicit reasoning about variables is required, because the encoding directly gives you the variables and binders; however, there is no self-interpreter for open lambda terms. Mogensen's encoding is stronger when the goal is to decode an open lambda term. The price of this, however, is that some seemingly simple functions, such as counting free variables, become undefinable.

Several directions for future work remain. The most immediate open problem is whether open and closed terms can be distinguished in Mogensen's encoding. A proof of impossibility would give an even better difference between the 2 encodings, while a positive result would reveal additional strengths in Mogensen's encoding. A second direction would be to study whether the normalization construction for both encodings can be simplified. Although the function G established in this thesis is correct, the construction is technically involved, and a more simplified formulation could improve the understanding. Finally, it remains interesting to investigate other properties of these encodings.

Bibliography

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[2] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

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

[4] Alessandro Berarducci and Corrado Böhm. A self-interpreter of lambda calculus having a normal form. In Egon Börger, Gerhard Jäger, Hans Kleine Büning, Simone Martini, and Michael M. Richter, editors, *Computer Science Logic, 6th Workshop, CSL '92, San Miniato, Italy, September 28 - October 2, 1992, Selected Papers*, Lecture Notes in Computer Science, pages 85–99. Springer, 1992.

[5] Corrado Böhm, Adolfo Piperno, and Stefano Guerrini. Lambda-definition of function(al)s by normal forms. In Donald Sannella, editor, *Programming Languages and Systems - ESOP'94, 5th European Symposium on Programming, Edinburgh, UK, April 11-13, 1994, Proceedings*, Lecture Notes in Computer Science, pages 135–149. Springer, 1994.

[6] Herman Geuvers. The power and limitations of self-interpretation. Lecture slides, fall 2015. Huygens College, `https://www.cs.ru.nl/~herman/onderwijs/2015Reflection/lecture7.pdf`.

[7] Herman Geuvers. Lecture 2: Self interpretation in the lambda-calculus. Lecture slides, winter 2016. 21st Estonian Winter School in Computer Science, `https://cs.ioc.ee/ewscs/2016/geuvers/geuvers-slides-lecture2.pdf`.

[8] Stephen C. Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353, 1936.

[9] Dexter Kozen. Church-Rosser Made Easy. *Fundam. Informaticae*, 103(1-4):129–136, 2010.

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

[11] Nathan van Beusekom. Properties of codings in lambda calculus, 2018. Bachelor thesis Computing Science, Radboud University, `https://www.cs.ru.nl/bachelors-theses/2018/Nathan_van_Beusekom___4571592___Properties_of_codings_in_lambda-calculus.pdf`.

[12] Wikipedia contributors. Pairing function. `https://en.wikipedia.org/wiki/Pairing_function`, 2026. Accessed 2026-06-06.