Radboud University Nijmegen

Faculty of Science

A Study In Meta

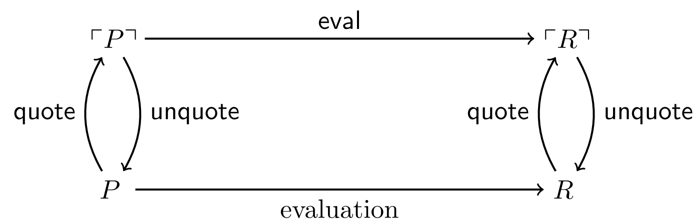
A Deep Dive Into Self-Interpreters for Lambda Calculus

Thesis Msc Mathemathical Foundations of Computer Science

Supervisor:
Herman Geuvers

Author:
Luuk Verkleij

Second reader:
Freek Wiedijk



August 2021

Contents

1 Introduction **3**

2 Preliminaries **5**
2.1 Untyped Lambda Calculus . . . 5
2.1.1 Evaluation . . . 6
2.1.2 Church Encoding . . . 8
2.2 Typed Lambda Calculus . . . 9
2.2.1 Church Encoding . . . 11
2.2.2 Normalization . . . 12
2.3 System F_ω . . . 12
2.3.1 Adding Polymorphism . . . 14
2.3.2 Adding Type Operators . . . 15
2.3.3 Normalization . . . 16
2.3.4 Church Encoding . . . 17

3 Self Interpreters **19**

4 Self-Interpreters For Untyped Lambda Calculus **21**
4.0.1 Quote as a term . . . 22
4.0.2 Recursive function definition scheme . . . 24
4.0.3 Self-Evaluator . . . 27
4.1 A Trivial Self-Recognizer . . . 30
4.2 Mogensen Self-Interpreter . . . 34
4.2.1 Function Definition Scheme . . . 36
4.2.2 double quote property . . . 37
4.3 Closed Term Self-Interpreters . . . 39
4.3.1 Closed-Term Function Definition Schemes . . . 40
4.4 Barendregt Self-Interpreter . . . 42
4.4.1 Function Definition Scheme . . . 42

5 Self-Interpreters For Typed Calculi **44**

6 Self-interpreters For Normalizing Calculi **48**
6.1 Computability Theory and Breaking The Self-Recognizer Normalization Barrier . . . 49
6.2 Bauer's Normalization Barrier . . . 51
6.3 Complexity of $\Box\tau$ . . . 53
6.4 Self-Evaluator . . . 57

7 System F_ω Self-Recognizers **58**
7.1 Trivial Self-Recognizer . . . 58
7.2 Brown-Palsberg Self-Recognizer . . . 63
7.2.1 Representing Types . . . 63
7.2.1.1 Properties & Proofs . . . 63
7.2.2 Coding Terms Lower-Order terms . . . 68
7.2.2.1 Properties & Proofs . . . 69
7.2.3 Coding Higher-Order Terms . . . 74
7.2.3.1 Constructor Abstraction Term . . . 74
7.2.3.2 Constructor Application Term . . . 77
7.2.4 The Final encoding Function . . . 78

7.2.4.1 Proofs 80
7.3 Function Definition Scheme 85
7.4 Applications 87
7.4.1 Term Type Tester 87
7.4.2 Normal Form Checker 89
7.4.3 Term Type Counter 94
7.5 Restrictions 98
7.5.0.1 Type Recognition 98

8 Discussion **100**

9 Related Work **101**

10 Conclusion **102**

1 Introduction

Self-interpreters, or sometimes called metacircular interpreters, are interpreters for the languages written in itself. They are omnipresent in the world of programming, for example Javascript[19], Python[32] and Haskell[26] have self-interpreters. Haskell is especially interesting, since it is, as a functional programming language, closely related to lambda calculus [2].

Lambda calculus was invented by Alonzo Church in 1936 and, like Turing machines, is a formal system designed to formalize the informal notion of effective calculable [14]. In the 85 year that followed it has been instrumental for theoretical computer science. Untyped lambda calculus has been a foundation for functional programming languages, whereas typed normalizing calculi have spawned proof assistants, like Coq[28]. However, even after 85 years of research, there are still plenty of questions in need to be answered. One of them arises with the intersection of strongly normalizing calculi and self-interpreters.

It is possible to distinguish between two self-interpreters, a self-recognizer which can interpreted the language in itself and a self-evaluator, which can interpreted language's evaluation in itself. The first self-recognizer for lambda-calculus was from Kleene [21] in 1936. The first typed self-recognizer came from Rendel, Ostermann and Hofer, as they presented a self-recognizer for F^*_ω [30]. The first self-recognizer for a strongly normalizing barrier was defined by Brown and Palsberg, which resulted in disproving a popular conjecture that is called the Normalization Barrier Conjecture. The normalization barrier is according to conventional wisdom that a self-interpreter for a strongly normalizing lambda calculus is impossible [10, 29]. The overturning of conventional wisdom have led to some questions. While a self-recognizer is possible in a strongly normalized lambda calculus, Brown and Palsberg stated that it is still an open question if a self-evaluator is possible[11]. Bauer notes that with the definition as understood it is possible to have a self-interpreter with weak structural properties. He questions if the definition of self-interpreter needs to be expanded such that strong structural properties are guaranteed[9].

In this thesis we try to get a clearer picture about two questions:

- Is there a definition for a self-interpreter that leads to strong structural properties
- Is it possible to have a self-evaluator for a strongly normalized lambda calculus.

We do this by making a review of the self-interpreters for lambda calculus. The result is that self-interpreters, that supports function definition schemes, have strong structural properties. We also find that a self-evaluator for a strongly normalizing lambda calculus is possible.

We start the thesis with Section 2 where we present a summary of the lambda calulculi used in this paper. In Section 3 we investigate what self-interpretation means in general. In Section 4 we look at 3 different self-interpreters for untyped lambda calculus and investigate their properties. In Section 5 we investigate what will change with a self-interpreter when we add types to lambda calculus. In Section 6 we investigate what the Normalization Barrier Conjecture is and why and what it means to break the normalization barrier. At the end of this section we answer an open question from Brown and Palsberg. In Section 7 we will investigate the first two self-interpreters that broke through the normalization barrier defined by Brown and Palsberg[10]. We end with

Section 8, where we discuss the results, Section 9, where we discuss related work, and as last Section 10 where we summarize and come to a conclusion.

2 Preliminaries

2.1 Untyped Lambda Calculus

Here we will summarize the key properties we use in this paper. For readers interested in a tutorial we refer to other sources [6, 33]. A compact summary of the untyped lambda calculus can be found in Definition 2.1.

Definition 2.1. Untyped Lambda Calculus
Grammar

$$\langle \mathsf{term} \rangle := \langle \mathsf{var} \rangle \ \mid (\lambda \, \langle \mathsf{var} \rangle \ . \ \langle \mathsf{term} \rangle) \mid (\langle \mathsf{term} \rangle \ \ \langle \mathsf{term} \rangle)$$

Λ is the set of all terms.
Λ_c is the set of all closed terms.

α-equivalence

$$\frac{}{M =_\alpha M} \qquad \frac{}{\lambda x \ . \ M =_\alpha \lambda y \ . \ M[x := y]} \ y \notin \mathbf{FV}(M)$$

$$\frac{M =_\alpha N}{N =_\alpha M} \qquad \frac{M =_\alpha N \quad N =_\alpha L}{M =_\alpha L}$$

β-reduction

$$\frac{}{(\lambda x \ . \ M) \ N \to_\beta M[x := N]} \qquad \frac{M \to_\beta N}{\lambda x \ . \ M \to_\beta \lambda x \ . \ N}$$

$$\frac{M \to_\beta N}{Z \ M \to_\beta Z \ N} \qquad \frac{M \to_\beta N}{M \ Z \to_\beta N \ Z}$$

$M \twoheadrightarrow_\beta N$ if M reduces to N in zero or more steps
$M \twoheadrightarrow_\beta^+ N$ if M reduces to N in one or more steps.

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

Fixed-Point Combinator

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

Untyped lambda calculus is defined using the following grammar

$$\langle \mathsf{term} \rangle := \langle \mathsf{var} \rangle \ \mid (\lambda \, \langle \mathsf{var} \rangle \ . \ \langle \mathsf{term} \rangle) \mid (\langle \mathsf{term} \rangle \ \ \langle \mathsf{term} \rangle)$$

The three cases we call variable, abstraction and application. Variables are written in lower case $a, b, c...$, whereas a term is written in a upper case $M, N, P, Q, ...$ the set

of all terms in notated with Λ. As a convention we say that applications are left associative, so we write $((M\ N)\ P)$ as $M\ N\ P$. Abstraction is right associative, we write $(\lambda x\ .\ (\lambda y\ .\ (\lambda z\ .\ x)))$ as $\lambda x\ .\ \lambda y\ .\ \lambda z\ .\ x$ and to reduce clutter we often write it as $\lambda x\ y\ z\ .\ x$.

When a term has a variable, we distinguish between *free* and *bound* variables. When we have an abstraction before the variable with the same name we say that it is bound, like x in the term $\lambda x\ .\ x$. When there is no abstraction before the variable with the same variable name it is free, like the variable x in the terms x or $\lambda y\ .\ x$. When all variables are bound we say that the term is *closed* and the set of closed terms we denote with Λ_c.

When variables are bound we find that we have terms that are the same except for renaming, like the terms $\lambda x\ .\ x$ and $\lambda y\ .\ y$. We can make use of α-*conversion* to turn these two terms into the same term. α-Conversion is defined as follows:

$$\lambda x\ .\ M \rightarrow_\alpha \lambda y\ .\ M[x := y]$$

We say that two terms M_1 and M_2 are α-*equivalent* to each other, written as $M_1 =_\alpha M_2$, if the two terms are equal to each other by applying zero or more α-conversions. Formally we define α-equivalence as follows:

Definition 2.2. (Definition of the Relation $=_\alpha$)

$$\frac{}{M =_\alpha M} \qquad \frac{}{\lambda x\ .\ M =_\alpha \lambda y\ .\ M[x := y]}\ y \notin \mathbf{FV}(M)$$

$$\frac{M =_\alpha N}{N =_\alpha M} \qquad \frac{M =_\alpha N \quad N =_\alpha L}{M =_\alpha L}$$

Alpha-equivalence does not affect evaluation, therefore if we want to avoid dealing with variable renaming we may want to remove α-conversion. This is especially useful when implementing lambda calculus. This can be done by making use of the *De Bruijn representation* [16], defined as follows:

$$\langle \mathsf{term} \rangle := n \mid \lambda \langle \mathsf{term} \rangle \mid \langle \mathsf{term} \rangle\ \langle \mathsf{term} \rangle$$

Where n is a number and point towards which abstraction it is bound to. For example "$\lambda x\ y\ .\ x$" is "$\lambda\ \lambda\ 2$" in De Bruijn representation.

2.1.1 Evaluation

For evaluation the lambda calculus uses the β-reduction rule. A β-reduction is transforming a term of the form $(\lambda x\ .\ M)\ N$, called a *redex*, by substituting variable x with the term N. The term we get in result who write as $M[x := N]$ is called a *redux*. When a term does not have a redex, we say the term is in *normal form*. When M does has a redex we can reduce this in a other name named N which we write as $M \rightarrow_\beta N$. The $\rightarrow_\beta$ relation is defined in Definition 2.3.

Definition 2.3. Definition of the relation $\rightarrow_\beta$

$$\frac{}{(\lambda x\ .\ M)\ N \rightarrow_\beta M[x := N]} \qquad \frac{M \rightarrow_\beta N}{\lambda x\ .\ M \rightarrow_\beta \lambda x\ .\ N}$$

$$\frac{M \to_\beta N}{Z\ M \to_\beta Z\ N} \qquad \frac{M \to_\beta N}{M\ Z \to_\beta N\ Z}$$

Sometimes we have to do a α-conversion before β-reduction. For example when we consider $(\lambda x\ .\ x\ y)\ y$. Then if we don't do a α-conversion first, we get $(\lambda x\ y\ .\ x\ y)\ y \to_\beta \lambda y\ .\ y\ y$, whereas it should reduce to something α-equivalent to $\lambda x\ .\ x\ y$. In short, when a beta-reduction binds a free variable, then we need to do a α-conversion first. Often we want to say that a term M reduces to N in zero or more steps. For this we write $\twoheadrightarrow_\beta$ and we formally define as follows:

Definition 2.4. Definition of the relation $\twoheadrightarrow_\beta$

$$\frac{M =_\alpha N}{M \twoheadrightarrow_\beta N} \qquad \frac{M \to_\beta N}{M \twoheadrightarrow_\beta N} \qquad \frac{M \to_\beta N \quad N \twoheadrightarrow_\beta L}{M \twoheadrightarrow_\beta L}$$

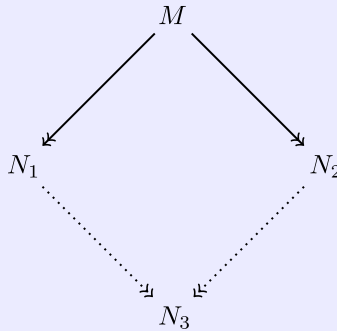
We write $M \twoheadrightarrow_\beta^+ N$ if M reduces into N in one or more steps. When $M \twoheadrightarrow_\beta N$ we say that M and N are β-equivalent, written as $M =_\beta N$ and $N =_\beta M$.

We can get a normal form of a term by repeatedly applying β-reduction. Not all terms do have a normal form however. Consider the term $(\lambda x\ .\ x\ x)(\lambda x\ .\ x\ x)$, then we have

$$(\lambda x\ .\ x\ x)(\lambda x\ .\ x\ x) \to_\beta (\lambda x\ .\ x\ x)(\lambda x\ .\ x\ x) \to_\beta (\lambda x\ .\ x\ x)(\lambda x\ .\ x\ x) \to_\beta \ldots$$

What we do have is that if a term has a normal then it only have one. A property that is called confluence or the Church-Rosser property, proven by Church and Rosser in 1936 [15]. Here we present the proof as given by Barendregt [8].

Theorem 2.5. *(Confluence / Church-Rosser Property)*
If $M \twoheadrightarrow_\beta N_1$ and $M \twoheadrightarrow_\beta N_2$ then there exists some N_3 such that $N_1 \twoheadrightarrow_\beta N_3$ and $N_2 \twoheadrightarrow_\beta N_3$, or in diagram form:



Proof. Prove of this property can be found in the literature, the standard prove is found in Barendregt's book "The Lambda Calculus" [8]. □

From this we find that if M has a normal form, then that is the only one. Therefore when we can talk about the normal form, instead of a normal form.

Corollary 2.6. *Let M be a term. If M has a normal form, then it is unique up to α-equivalence.*

Proof. Let N_1 and N_2 be two distinct normal forms of M. We have $N_1 =_\beta N_2$, therefore there is a term Z such that $N_1 \twoheadrightarrow_\beta Z$ and $N_2 \twoheadrightarrow_\beta Z$. Since both N_1 and N_2 are in normal form and therefore do not have a redex, then it follows from the definition of $\twoheadrightarrow_\beta$ that $N_1 =_\alpha Z =_\alpha N_2$. □

We have another important property for the lambda calculus, we namely have a fixed-point theorem.

Theorem 2.7. *For every term M there is a term X such that $M\ X =_\beta X$.*

Proof.
Let $N = \lambda x\ .\ M\ (x\ x)$ and $X = N\ N$, then we have

$$X = N\ N =_\beta \lambda x\ .\ M\ (x\ x)\ N =_\beta M\ (N\ N) = M\ X$$

□

From this we can define a fixed-point combinator, i.e. a function that returns some fixed-point of its argument.

Corollary 2.8. *Let Y be defined as follows: $Y := \lambda f\ .\ (\lambda x\ .\ f\ (x\ x))\ (\lambda x\ .\ f\ (x\ x))$ Then Y is a fixed-point combinator, i.e. for every term M $M\ (Y\ M) =_\beta Y\ M$.*

This will come in handy, since we now can define a term by recursion by using the fixed-point combinator.

2.1.2 Church Encoding

In lambda calculus we can also define datatypes, like the booleans, natural numbers and tuples and operations on them. These are named after Church as he defined it first. Here we show the definition and operations, but not proof the correctness of them.

Definition 2.9. (Church Booleans)

$$\mathsf{true} := \lambda t\ f\ .\ t$$
$$\mathsf{false} := \lambda t\ f\ .\ f$$

Definition 2.10. (Church Booleans Operations)

$$\begin{aligned} \mathsf{and} &:= \lambda p\ q\ .\ p\ q\ p \\ \mathsf{or} &:= \lambda p\ q\ .\ p\ p\ q \\ \mathsf{not} &:= \lambda p\ .\ p\ \mathsf{false}\ \mathsf{true} \\ \mathsf{if} &:= \lambda p\ t\ f\ .\ p\ t\ f \end{aligned}$$

Note that with this encoding of booleans, booleans themselves already form a if statement. Therefore the term if is not needed, however it can be useful clarification.

To define the church numerals, we will be making use of power notation which we define as follows

$$M^1 N := MN \text{ and } M^{n+1} N := M(M^n N)$$

Definition 2.11. (Church Numerals)

$$\overline{0} := \lambda f\ x\ .\ x$$
$$\overline{n} := \lambda f\ x\ .\ f^n x$$

Definition 2.12. (Operations on Church Numerals)

$$\begin{array}{lllll} \text{Succesor Function:} & \mathsf{suc} & := & \lambda n\ .\ \lambda f\ .\ \lambda x\ .\ f\ (n\ f\ x) \\ \text{Addition:} & \mathsf{add} & := & \lambda m\ .\ \lambda n\ .\ n\ \mathsf{suc}\ m \\ \text{Multiplication:} & \mathsf{mult} & := & \lambda m\ .\ \lambda n\ .\ \lambda f\ .\ m\ (n\ f) \\ \text{Exponentiation:} & \mathsf{exp} & := & \lambda m\ .\ \lambda n\ .\ n\ m \end{array}$$

Definition 2.13. (Church Pairs)

$$\begin{array}{llll} \text{Pairing Function:} & \mathsf{pair} & := & \lambda x\ y\ z\ .\ z\ x\ y \\ \text{First Projection:} & \mathsf{fst} & := & \lambda p\ .\ p\ \mathsf{true} \\ \text{Second Projection:} & \mathsf{snd} & := & \lambda p\ .\ p\ \mathsf{false} \end{array}$$

2.2 Typed Lambda Calculus

Until now we have looked at lambda calculus without types. There are also are typed variants. In this section we will summarize the important properties of simply typed lambda calculus, or $\lambda_{\rightarrow}$, for this paper. For a tutorial we point the reader to Barendregt's "Introduction to Lambda Calculus" [6]. A summary of simply typed lambda calculus can be found in Definition 2.14.

Definition 2.14. Simply Typed Lambda Calculus / $\lambda_{\rightarrow}$
Grammar

$$\begin{array}{rl} \langle\mathsf{type}\rangle := & \epsilon\ |\ \langle\mathsf{typevar}\rangle\ \ |\ \langle\mathsf{type}\rangle \rightarrow \langle\mathsf{type}\rangle \\ \langle\mathsf{preterm}\rangle := & \langle\mathsf{termvar}\rangle\ \ |\ \lambda\,\langle\mathsf{var}\rangle : \langle\mathsf{type}\rangle\ \ .\ \langle\mathsf{term}\rangle\ \ |\ \langle\mathsf{preterm}\rangle\ \ \langle\mathsf{preterm}\rangle \end{array}$$

Type Derivation Rules
A *context* is defined as $\Gamma = \{x_1 : \tau_1,\ .\ , x_n : \tau_n\}$.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma \ . \ M) : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M \ N : \sigma}$$

Terms
A pre-term M is a term of $\lambda_{\to}$ if there exists a context Γ and a type τ such that $\Gamma \vdash M : \tau$
Λ^{τ} denotes all terms of type τ.
Λ_c^{τ} denotes all closed terms of type τ.
If $\Lambda_c^{\tau} = \emptyset$ then we say τ is not *inhabited*.

We will be introducing *simply typed lambda calculus*, also written as $\lambda \to$, which is the foundation of every typed lambda calculus. We will be defining types as follows:

$$\langle \mathsf{type} \rangle := \epsilon \ | \ \langle \mathsf{typevar} \rangle \ | \ \langle \mathsf{type} \rangle \to \langle \mathsf{type} \rangle$$

Here $\langle \mathsf{typevar} \rangle$ is of the form of $\{\alpha, \beta, \gamma, ...\}$ and act similarly to variables in terms. We can consider types again as open and as closed. Here we take ϵ as our base type, and we consider ϵ the empty type. We will see that it is possible for types to not have closed terms and we define that ϵ is such a type.

Now for the definition of terms, we will assume that every term has a type as we defined before.

$$\langle \mathsf{preterm} \rangle := \langle \mathsf{termvar} \rangle \ | \ \lambda \, \langle \mathsf{var} \rangle : \langle \mathsf{type} \rangle \ . \ \langle \mathsf{term} \rangle \ | \ \langle \mathsf{preterm} \rangle \ \langle \mathsf{preterm} \rangle$$

We call this pre-term because not every pre-term we can define makes sense if we enforce typing. For example if we have the pre-term $\lambda x : \sigma \ . \ x$, type $\tau \neq \sigma$ and a term $M : \tau$, then we want to disallow the pre-term $(\lambda x : \sigma \ . \ x) \ M$. To do this we introduce *derivation rules*.

Definition 2.15. Type Judgement

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma \ . \ M) : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M \ N : \sigma}$$

Γ is called a *context*, which is a set $\{x_1 : \sigma_1, \ldots x_n : \sigma_n\}$. When we can derive $M : \sigma$ from a context Γ we have a type judgement, written as $\Gamma \vdash M : \sigma$.

Now we define a term of $\lambda_{\to}$ to be a pre-term M with a type judgement $\Gamma \vdash M : \sigma$. Note that for all closed terms we have a type judgement of the form $\emptyset \vdash M : \sigma$, also written as $\vdash M : \sigma$. The set of all terms of type σ is written as Λ^{σ} and the set of all closed terms of type σ is written as Λ_c^{σ}.

By introducing types we get a interesting side effect, not every type has a closed term. An example for this is ϵ by definition but also the type type $(\epsilon \to \epsilon) \to \epsilon$ as no closed terms. When there is a type σ and a term M such that $\vdash M : \sigma$, we say that σ is *inhabited*. Note that while ϵ is not inhabited, $\epsilon \to \epsilon$ is inhabited by the term $\lambda x : \epsilon \ . \ x$ as follows from this type derivation:

$$\frac{\dfrac{x : \epsilon \in \{x : \epsilon\}}{\{x : \epsilon\} \vdash x : \epsilon}}{\vdash (\lambda x : \epsilon \ . \ x) : \epsilon \to \epsilon}$$

2.2.1 Church Encoding

In $\lambda_{\rightarrow}$ we can also define datatypes, like the booleans, natural numbers and tuples and the operations on them. Here we again show no proof of the correctness of definitions.

Definition 2.16. (Church Booleans)

$$\begin{aligned} \mathsf{true} &:= \lambda t\ f : \tau\ .\ t \\ \mathsf{false} &:= \lambda t\ f : \tau\ .\ f \end{aligned}$$

Let $\mathsf{Bool}_\tau := \tau \to \tau \to \tau$ then

Definition 2.17. (Church Booleans Operations)

$$\begin{aligned} \mathsf{and} &:= \lambda p\ q : \mathsf{Bool}_\tau\ .\ \lambda t\ f : \tau\ .\ p\ (q\ t\ f)\ f \\ \mathsf{or} &:= \lambda p\ q : \mathsf{Bool}_\tau\ .\ \lambda t\ f : \tau\ .\ p\ t\ (q\ t\ f) \\ \mathsf{not} &:= \lambda p : \mathsf{Bool}_\tau\ .\ \lambda t\ f : \tau\ .\ p\ f\ t \\ \mathsf{if} &:= \lambda p : \mathsf{Bool}_\tau\ .\ \lambda t\ f : \tau\ .\ p\ t\ f \end{aligned}$$

Note that with types we can no longer take the short cuts we did with untyped lambda calculus and also note that the term if only works for terms of type τ.

Let τ be some type, then we define the church numerals (under type τ) as follows:

Definition 2.18. (Church Numerals)

$$\begin{aligned} \overline{0} &:= \lambda f : \tau \to \tau\ .\ x : \tau\ .\ x \\ \overline{n} &:= \lambda f : \tau \to \tau\ .\ x : \tau\ .\ f^n x \end{aligned}$$

Now let the type $\mathsf{Nat}_\tau := (\tau \to \tau) \to \tau \to \tau$.

Definition 2.19. (Operations on Church Numerals)

$$\begin{aligned} \mathsf{suc} &:= \lambda n : \mathsf{Nat}_\tau\ .\ \lambda f : \tau \to \tau\ .\ \lambda x : \tau\ .\ f\ (n\ f\ x) \\ \mathsf{add} &:= \lambda m : \mathsf{Nat}_\tau\ .\ \lambda n : \mathsf{Nat}_\tau\ .\ \lambda f : \tau \to \tau\ .\ \lambda x : \tau\ .\ m\ f\ (n\ f\ x) \\ \mathsf{mult} &:= \lambda m : \mathsf{Nat}_\tau\ .\ \lambda n : \mathsf{Nat}_\tau\ .\ \lambda f : \tau \to \tau\ .\ \lambda x : \tau\ .\ m\ (n\ f)\ x \end{aligned}$$

Compared to untyped lambda calculus we do not have exponentiation, as it is impossible to define. In fact the type Nat_τ under β-conversion are exactly the extended polynomials [40].

Definition 2.20. (Church Pairs)

$$\begin{aligned}
\mathsf{pair}_\tau &:= \lambda x\ y : \tau\ .\ z : \tau \to \tau \to \tau\ .\ z\ x\ y \\
\mathsf{fst}_\tau &:= \lambda p\ .\ p\ \mathsf{true}_\tau \\
\mathsf{snd}_\tau &:= \lambda p\ .\ p\ \mathsf{false}_\tau
\end{aligned}$$

2.2.2 Normalization

We have seen that not all terms that are definable in untyped lambda calculus are definable when we introduce types. What is also not definable is the fixed-point combinator. This however hints at a property that $\lambda_\to$ has, namely every term has a normal form. A property we will call *normalization*. We can consider this property analogous to the "total" property from computability theory. Except in lambda calculus we can consider two different types of normalization. Since β-reduction rules are non-deterministic it will happen that there are multiple possible β-reductions we can choose from. We call the order in which we have chosen the β-reductions a path. So we can consider a term to be normalizing either if there exists a finite path to the normal form or a if every path will lead in finite steps to a normal form.

Definition 2.21. A calculus is *Weakly Normalizing* when for every term there is at least one finite reduction path ending in a normal form.

Definition 2.22. A calculus is *Strongly Normalizing* when there is no term with an infinite reduction path.

We find that $\lambda_\to$ is strongly normalizing.

Lemma 2.23. *$\lambda_\to$ is strongly normalizing.*

Proof.
Multiple proofs of this can be found from the literature. The classical proof is given by Tait in "Intensional interpretations of functionals of finite type" [37]. Inspired by the Barendregt–Geuvers–Klop conjecture, Morten Heine Sørensen infers a proof of strong normalization from weak normalization in "Strong Normalization from Weak Normalization in Typed λ-Calculi"[35]. □

2.3 System F_ω

In this section we will we will summarize the important properties of System F_ω, or λ_ω , for this paper. For a compact version we refer to Definition 2.24. For a tutorial of System F_ω and other calculi in the Barendregt's lambda cube we refer the reader to Barendregt's "Lambda Calculi with Types" [3, §5].

Definition 2.24. System F_ω / λ_ω
Grammar

$$
\begin{aligned}
\langle \mathsf{kind} \rangle &:= * \mid \langle (\langle \mathsf{kind} \rangle) \to \langle \mathsf{kind} \rangle \\
\langle \mathsf{ctor} \rangle &:= \langle \mathsf{typevar} \rangle \mid \langle \mathsf{ctor} \rangle \to \langle \mathsf{ctor} \rangle \mid \Pi \langle \mathsf{typevar} \rangle : \langle \mathsf{kind} \rangle \ . \ \langle \mathsf{ctor} \rangle \\
&\mid \lambda \langle \mathsf{typevar} \rangle : \langle \mathsf{kind} \rangle \ . \ \langle \mathsf{ctor} \rangle \mid \langle \mathsf{ctor} \rangle \ \langle \mathsf{ctor} \rangle \\
\langle \mathsf{preterm} \rangle &:= \langle \mathsf{termvar} \rangle \mid \lambda \langle \mathsf{termvar} \rangle : \langle \mathsf{type} \rangle \ . \ \langle \mathsf{term} \rangle \mid \langle \mathsf{preterm} \rangle \ \langle \mathsf{preterm} \rangle \\
&\mid (\lambda \langle \mathsf{typevar} \rangle : \langle \mathsf{kind} \rangle \ . \ \langle \mathsf{preterm} \rangle) \mid (\langle \mathsf{preterm} \rangle \ \langle \mathsf{ctor} \rangle)
\end{aligned}
$$

A *context* is defined as $\Gamma = \{x_1 : \tau_1, \ . \ , x_n : \tau_n\} \cup \{\alpha_1 : \kappa_1, \ . \ , \alpha_m : \kappa_m\}$.

Kind Derivation Rules

$$
\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \tau \to \sigma} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \Pi\alpha : \kappa \ . \ \tau}
$$

$$
\frac{\Gamma, \alpha : \kappa_1 \vdash C : \kappa_2}{\Gamma \vdash (\lambda\alpha : \kappa_1 \ . \ C) : \kappa_1 \to \kappa_2} \qquad \frac{\Gamma \vdash C_1 : \kappa_2 \to \kappa_1 \qquad \Gamma \vdash C_2 : \kappa_2}{\Gamma \vdash C_1 \ C_2 : \kappa_1}
$$

Type β-Reduction

$$
\frac{}{(\lambda\alpha \ . \ C_1) \ C_2 \to_\beta C_1[\alpha := C_2]} \qquad \frac{C_1 \to_\beta C_2}{\lambda\alpha : \kappa \ . \ C_1 \to_\beta \lambda\alpha : \kappa \ . \ C_2}
$$

$$
\frac{C_1 \to_\beta C_2}{C_3 \ C_1 \to_\beta C_3 \ C_2} \qquad \frac{C_1 \to_\beta C_2}{C_1 \ C_3 \to_\beta C_2 \ C_3}
$$

$$
\frac{\tau_1 \to_\beta \sigma_1 \qquad \tau_2 \to_\beta \sigma_2}{\tau_1 \to \tau_2 \to_\beta \sigma_1 \to \sigma_2} \qquad \frac{\tau \to_\beta \sigma}{\Pi\alpha : \kappa \ . \ \tau \to_\beta \Pi\alpha : \kappa \ . \ \sigma}
$$

Type Derivation Rules

$$
\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \vdash \sigma : * \qquad \Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma \ . \ M) : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M \ N : \sigma}
$$

$$
\frac{\Gamma, \alpha : \kappa \vdash M : \tau}{\Gamma \vdash (\lambda\alpha : \kappa \ . \ M) : (\Pi\alpha : \kappa \ . \ \tau)} \qquad \frac{\Gamma \vdash M : (\Pi\alpha : \kappa \ . \ \tau) \qquad \Gamma \vdash C : \kappa}{\Gamma \vdash M \ C : \tau[\alpha := C]}
$$

$$
\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash \tau =_\beta \sigma \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash M : \sigma}
$$

Term β-reduction

$$
\frac{}{(\lambda x : \tau \ . \ M) \ N \to_\beta M[x := N]} \qquad \frac{}{(\lambda\alpha : \kappa \ . \ M) \ C \to_\beta M[\alpha := C]}
$$

$$\frac{M \to_\beta N}{L\ M \to_\beta L\ N} \qquad \frac{M \to_\beta N}{M\ L \to_\beta N\ L} \qquad \frac{M \to_\beta N}{M\ C \to_\beta N\ C}$$

$$\frac{M \to_\beta N}{\lambda x : \tau\ .\ M \to_\beta \lambda x : \tau\ .\ N} \qquad \frac{M \to_\beta N}{\lambda \alpha : \kappa\ .\ M \to_\beta \lambda \alpha : \kappa\ .\ N}$$

System F_ω, also named λ_ω , is one of the calculi on the Barendregt lambda cube[4], as shown in Figure 1. It extends $\lambda_\to$ with polymorphism and type operators. Here we will only introduce System F_ω and not the rest of Barendregt's cube.

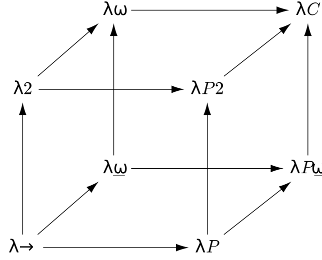


Figure 1: Barendregt's Lambda Cube[4]

2.3.1 Adding Polymorphism

To get to System F_ω, we will first introduce polymorphism to a typed lambda calculus. Polymorphism can also be understood as making terms be depended on typed. For example in our simply typed world, for every type we have a different identity term, namely the term $\lambda x : \tau\ .\ x$. All of these are the same term, just under a different type. What we could do introduce a new application rule involving types, i.e. some rule $M\ \tau$, such that we can have a single identity term. This is what we do when we introduce polymorphism.

First we introduce what is essentially a type for types, we will be named a kind. A kind is symbolised with $*$ and every type is of kind $*$. Now we can introduce a abstraction rule, $\lambda\alpha : *\ .\ M$ and with that an application rule $M\ \tau$. Now for type checking we will want that our new application term can only accept types. For this we have to introduce a new type which accept a type variable and gives back a type. This will be written as $\Pi\alpha : *\ .\ \tau$. This results in grammar for our calculus as in Definition 2.3.1.

Definition 2.25. (Grammar for Typed Calculus with Polymorphism)

$$\begin{aligned}
\langle \mathsf{kind} \rangle &:= * \\
\langle \mathsf{type} \rangle &:= \ \langle \mathsf{typevar} \rangle \ \mid \ \langle \mathsf{type} \rangle \to \langle \mathsf{type} \rangle \ \mid \Pi \, \langle \mathsf{typevar} \rangle : \langle \mathsf{kind} \rangle \\
&\quad \mid \ \langle \mathsf{type} \rangle \ \ \langle \mathsf{preterm} \rangle \\
\langle \mathsf{preterm} \rangle &:= \langle \mathsf{termvar} \rangle \ \mid \lambda \, \langle \mathsf{termvar} \rangle : \langle \mathsf{type} \rangle \ \ . \ \langle \mathsf{term} \rangle \ \mid \ \langle \mathsf{preterm} \rangle \ \ \langle \mathsf{preterm} \rangle \\
&\quad \mid (\lambda \, \langle \mathsf{typevar} \rangle : \langle \mathsf{kind} \rangle \ \ . \ \langle \mathsf{type} \rangle) \mid (\langle \mathsf{preterm} \rangle \ \ \langle \mathsf{type} \rangle)
\end{aligned}$$

Now that we have a way to bind and make use of type variables, we have to also introduce α-equality for types. For example the types $\Pi\alpha : *\ .\ \alpha$ looks a lot like $\Pi\beta : *\beta$ and in fact we want them to be the same. We can adept the α-equivalence definition given in 2.2 to terms.

Definition 2.26. (α-Equivalence For Types)

$$\frac{}{\tau =_\alpha \tau} \qquad \frac{}{\Pi\alpha : * \ . \ \tau =_\alpha \lambda\beta \ . \ \tau[\alpha := \beta]} \ \alpha \notin \mathbf{FV}(\tau)$$

$$\frac{\tau =_\alpha \sigma}{\sigma =_\alpha \tau} \qquad \frac{\tau =_\alpha \sigma \quad \sigma =_\alpha \nu}{\tau =_\alpha \nu}$$

We have also introduced new pre-terms, so to make use of these we require an extension to type derivation rules, α-equality for terms and β-reduction.

Definition 2.27. (Extensions for Polymorphism)
α-Equivalence

$$\frac{\tau =_\alpha \sigma}{\lambda x : \tau \ . \ M =_\alpha \lambda y : \sigma \ . \ M[x := y]} \ y \notin \mathbf{FV}(M)$$

$$\frac{M =_\alpha N}{\lambda\alpha : * \ . \ M =_\alpha \lambda\alpha : * \ . \ N} \qquad \frac{M =_\alpha N \quad \tau =_\alpha \sigma}{M \ \tau \ =_\alpha \ N \ \sigma}$$

Type Derivation Rules

$$\frac{\Gamma, \alpha : * \vdash M : \tau}{\Gamma \vdash (\lambda\alpha : * \ . \ M) : (\Pi\alpha : * \ . \ \tau)} \qquad \frac{\Gamma \vdash M : (\Pi\alpha : * \ . \ \tau)}{\Gamma \vdash M \ \sigma : \tau[\alpha := \sigma]}$$

β-Reduction

$$\frac{}{(\lambda\alpha : * \ . \ M) \ \sigma \to_\beta M[\alpha := \sigma]} \qquad \frac{M \to_\beta N}{\lambda\alpha : * \ . \ M \to_\beta \lambda\alpha : * \ . \ N}$$

$$\frac{M \to_\beta N}{M \ \tau \to_\beta N \ \tau}$$

2.3.2 Adding Type Operators

Secondly we will be introducing type operators to a typed lambda calculus. Type operators can also be understood as making types be depended on types. For example in our simply typed world, for every type we have a different pair type. However all of them are of the form of $\tau \to \tau \to (\tau \to \tau \to \tau) \to \tau$. Similarly to the identity term in the polymorphic example, we may want to have a single "type" Pair that accepts a type τ and returns Pair_τ such that we can have a single pairing type. Essentially, we would like to have functions for types. For this we introduce type operators.

Since we now have kinds for types, introduced when we added polymorphism, we can extend our type system with rules of $\lambda_\to$. If we represent kinds with κ, then we can add a rule to our kinds $\kappa_1 \to \kappa_2$. Then now we can add abstraction and application to our type system. We from now on will call types and functions on types together

constructors, denoted with C and abbreviate with ctor. This results in the following new grammar:

$$\begin{aligned}\langle\textsf{kind}\rangle &:= * \mid \langle(\langle\textsf{kind}\rangle)\rangle \rightarrow \langle\textsf{kind}\rangle \\ \langle\textsf{ctor}\rangle &:= \langle\textsf{typevar}\rangle \mid \langle\textsf{ctor}\rangle \rightarrow \langle\textsf{ctor}\rangle \mid \Pi\,\langle\textsf{typevar}\rangle : \langle\textsf{kind}\rangle\ .\ \langle\textsf{ctor}\rangle \\ &\quad \mid \lambda\,\langle\textsf{typevar}\rangle : \langle\textsf{kind}\rangle\ .\ \langle\textsf{ctor}\rangle \mid \langle\textsf{ctor}\rangle\ \langle\textsf{ctor}\rangle\end{aligned}$$

Note that with the introduction of constructors, we can also make use of this with the polymorphism. Therefore we also change the pre-terms polymorphic cases.

$$\begin{aligned}\langle\textsf{preterm}\rangle &:= \langle\textsf{termvar}\rangle \mid \lambda\,\langle\textsf{termvar}\rangle : \langle\textsf{type}\rangle\ .\ \langle\textsf{term}\rangle \mid \langle\textsf{preterm}\rangle\ \langle\textsf{preterm}\rangle \\ &\quad \mid (\lambda\,\langle\textsf{typevar}\rangle : \langle\textsf{kind}\rangle\ .\ \langle\textsf{preterm}\rangle) \mid (\langle\textsf{preterm}\rangle\ \langle\textsf{ctor}\rangle)\end{aligned}$$

Now that we have introduced rules from $\lambda_{\rightarrow}$ to our typing system, we also need derivation rules and reduction rules. Note that we have a clear distinction between types, denoted with τ and type and constructors, denoted with C, as we still require terms to be of a type, which is always of kind $*$.

Kind Derivation Rules

$$\frac{\alpha : \kappa \in \Gamma}{\Gamma \vdash \alpha : \kappa} \qquad \frac{\Gamma \vdash \tau : * \qquad \Gamma \vdash \sigma : *}{\Gamma \vdash \tau \rightarrow \sigma} \qquad \frac{\Gamma, \alpha : \kappa \vdash \tau : *}{\Gamma \vdash \Pi\alpha : \kappa\ .\ \tau}$$

$$\frac{\Gamma, \alpha : \kappa_1 \vdash C : \kappa_2}{\Gamma \vdash (\lambda\alpha : \kappa_1\ .\ C) : \kappa_1 \rightarrow \kappa_2} \qquad \frac{\Gamma \vdash C_1 : \kappa_2 \rightarrow \kappa_1 \qquad \Gamma \vdash C_2 : \kappa_2}{\Gamma \vdash C_1\ C_2 : \kappa_1}$$

Type β-Reduction

$$\frac{}{(\lambda\alpha\ .\ C_1)\ C_2 \rightarrow_\beta C_1[\alpha := C_2]} \qquad \frac{C_1 \rightarrow_\beta C_2}{\lambda\alpha : \kappa\ .\ C_1 \rightarrow_\beta \lambda\alpha : \kappa\ .\ C_2}$$

$$\frac{C_1 \rightarrow_\beta C_2}{C_3\ C_1 \rightarrow_\beta C_3\ C_2} \qquad \frac{C_1 \rightarrow_\beta C_2}{C_1\ C_3 \rightarrow_\beta C_2\ C_3}$$

$$\frac{\tau_1 \rightarrow_\beta \sigma_1 \qquad \tau_2 \rightarrow_\beta \sigma_2}{\tau_1 \rightarrow \tau_2 \rightarrow_\beta \sigma_1 \rightarrow \sigma_2} \qquad \frac{\tau \rightarrow_\beta \sigma}{\Pi\alpha : \kappa\ .\ \tau \rightarrow_\beta \Pi\alpha : \kappa\ .\ \sigma}$$

Putting it all together, we get System F_ω as we can see in Definition 2.24.

2.3.3 Normalization

Instead of proving that System F_ω is strongly normalizing, it's also possible to prove that all calculi from Barendregt's lambda cube are strongly normalizing.

Theorem 2.28. *All calculi in Barendregt's lambda cube are strongly normalizing*

Proof.
Multiple proofs that λ_C is strongly normalizing are available in the literature. We point to Herman Geuvers' proof in "A short and flexible proof of strong normalization for the calculus of constructions" [17]. Since λ_C is an extension of the other calculi in the Barendregt's cube, it follows that they are also strongly normalizing. $\square$

2.3.4 Church Encoding

We can again define a church encoding, but unlike $\lambda_\to$ these will be closer to the Church Booleans from untyped lambda calculus, since we now can chose our type.

Definition 2.29. (Church Booleans)

$$\mathsf{Bool} := \Pi\alpha : * \,.\, \alpha \to \alpha \to \alpha$$

$$\mathsf{true} := \lambda\alpha : * \,.\, \lambda t\ f : \mathsf{Bool}\ \alpha \,.\, t$$
$$\mathsf{false} := \lambda\alpha : * \,.\, \lambda t\ f : \mathsf{Bool}\ \alpha \,.\, f$$

Definition 2.30. (Church Booleans Operations)

$$\begin{aligned}
\mathsf{and} &:= \lambda p\ q : \mathsf{Bool} \,.\, \Pi\alpha : * \,.\, \lambda t\ f : \alpha \,.\, p\ \alpha\ (q\ \alpha\ t\ f)\ f \\
\mathsf{or} &:= \lambda p\ q : \mathsf{Bool} \,.\, \Pi\alpha : * \,.\, \lambda t\ f : \alpha \,.\, p\ \alpha\ t\ (q\ \alpha\ t\ f)\ f \\
\mathsf{not} &:= \lambda p : \mathsf{Bool} \,.\, \Pi : \alpha : * \,.\, \lambda t\ f : \alpha \,.\, p\ \alpha\ f\ t \\
\mathsf{if} &:= \lambda p : \mathsf{Bool} \,.\, \Pi : \alpha : * \,.\, \lambda t\ f : \alpha \,.\, p\ \alpha\ t\ f
\end{aligned}$$

When defining the Church Numerals for System F_ω, we get a nice surprise; we can define the exponentiation again.

Definition 2.31. (Church Numerals)

$$\mathsf{Nat} := \Pi\alpha : * \,.\, (\alpha \to \alpha) \to \alpha \to \alpha$$

$$\overline{0} := \lambda\alpha : * \,.\, \lambda f : \alpha \to \alpha \,.\, x : \alpha \,.\, x$$
$$\overline{n} := \lambda\alpha : * \,.\, \lambda f : \alpha \to \alpha \,.\, x : \alpha \,.\, f^n x$$

Definition 2.32. (Operations on Church Numerals)

$$\begin{aligned}
\mathsf{suc} &:= \lambda n : \mathsf{Nat} \;.\; \lambda \alpha : * \;.\; \lambda f : \alpha \to \alpha \;.\; \lambda x : \alpha \;.\; f\ (n\ \alpha\ f\ x) \\
\mathsf{add} &:= \lambda m\ n : \mathsf{Nat} \;.\; n\ \mathsf{Nat}\ \mathsf{suc}\ m \\
\mathsf{mult} &:= \lambda m\ n : \mathsf{Nat} \;.\; \lambda \alpha : * \;.\; \lambda f : \alpha \to \alpha \;.\; m\ \alpha\ (n\ \alpha\ f) \\
\mathsf{exp} &:= \lambda m\ n : \mathsf{Nat} \;.\; \lambda \alpha : * \;.\; n\ (\alpha \to \alpha)\ (m\ \alpha)
\end{aligned}$$

One of the strengths of polymorphism and type operators is that now we are able to define tuples. These tuples are analogous to the Church pairs from untyped lambda calculus.

Definition 2.33. (Church Tuples)

$$\mathsf{Tuple} := \lambda \alpha\ \beta : * \;.\; \Pi \gamma : * \;.\; (\alpha \to \beta \to \gamma) \to \gamma$$

$$\begin{aligned}
\mathsf{tuple} &:= \lambda \alpha\ \beta : * \;.\; \lambda x : \alpha \;.\; \lambda y : \beta \;.\; \lambda \gamma : * \;.\; \lambda z : \alpha \to \beta \to \gamma \;.\; z\ x\ y \\
\mathsf{tfst} &:= \lambda \alpha\ \beta : * \;.\; \lambda p : \mathsf{Tuple}\ \alpha\ \beta \;.\; p\ \alpha\ (\lambda x : \alpha \;.\; \lambda y : \beta\ x) \\
\mathsf{tsnd} &:= \lambda \alpha\ \beta : * \;.\; \lambda p : \mathsf{Tuple}\ \alpha\ \beta \;.\; p\ \beta\ (\lambda x : \alpha \;.\; \lambda y : \beta\ x)
\end{aligned}$$

And from Church tuples we can define Church pairs.

Definition 2.34. (Church Pairs)

$$\mathsf{Pair} := \lambda \alpha : * \;.\; \mathsf{Tuple}\ \alpha\ \alpha$$

$$\begin{aligned}
\mathsf{pair} &:= \lambda \alpha : * \;.\; \lambda x\ y : \alpha \;.\; \mathsf{tuple}\ \alpha\ \alpha\ x\ y \\
\mathsf{fst} &:= \lambda \alpha : * \;.\; \lambda p : \mathsf{Pair}\ \alpha \;.\; \mathsf{tfst}\ \alpha\ \alpha\ p \\
\mathsf{snd} &:= \lambda \alpha : * \;.\; \lambda p : \mathsf{Pair}\ \alpha \;.\; \mathsf{tsnd}\ \alpha\ \alpha\ p
\end{aligned}$$

3 Self Interpreters

The informal definition of a self-interpreter is an interpreter for the language written in the language itself.[11][7][24] However we have to note that the precise definition of self-interpreter seems to be contested or at least unclear. Therefore it is important that first the definition of self-interpretation, used in this paper, will be presented and the motivation for this definition.

The definition of self-interpreter is unclear, as was shown when Matt Brown and Jens Palsberg released their paper "Breaking Through the Normalization Barrier: A Self-Interpreter for F-omega".[10] In this paper they broke through conventional wisdom that a self-interpreter is impossible for total languages and this sparked multiple comments about the definition of self-interpreter. For example Ben Lynn commented on this "there is no official agreement on the definition of representation or self-interpretation, or even how we should name these concepts."[23] and Andrej Bauer commented "The construction is trivial, which makes one wonder whether something is wrong with the notion of self-interpreter used by Brown and Palsberg." [9]. This uncertainty of the definition is also made clear by the paper itself, since the first thing Brown and Palsberg did is clarify that they use Barendregt's [7] notation of self-interpreter and a year later Brown and Palsberg came back on their assertion that they had made a self-interpreter. Instead in their paper "Typed Self-Evaluation via Intensional Type Functions" they renamed it to a self-recognizer.[11]

Before we give the notion of self-interpretation, we have to note where the terminology comes from. The concept of self-interpretation comes from popular programming languages, like Lisp, Python, Haskell, Javascript and many more. In these programming languages one has a self-interpreter in the form of an evaluate or execute function. In these functions we can enter a program of the program language itself, as a string, and it will return the value that would be obtained through evaluation of the program. Now it is important to remember that when we do this, we enter a string, a value. This is important to note, since that means that when we are talking about self-interpretation, we are talking about a combination of representation and some form of interpretation. In the case of programming languages, the representation is the string. The representation of a program we call an encoding and the encoding function itself we call quote or $\ulcorner - \urcorner$ and we write $\ulcorner P \urcorner$ if we have a program P in its representation form.

Finally we still need to mention the interpreter part of the self-interpreter, which is also where the confusion is. When we interpret a representation of a program, let's say P, we could return two end results, either the evaluation of the program or a representation of the evaluation of the program. There is a difference between those two. Let's say we have a program P, which evaluates into R, and the representation of this program $\ulcorner P \urcorner$. Then to get R we could have a program to turn $\ulcorner P \urcorner$ into P and then evaluate. Such a program we will call unquote and the program together with the representation will be called a self-recognizer. However to turn $\ulcorner P \urcorner$ into the representation of the result of the evaluation of P is less easy. This is because we will find that, in general, there does not exist a program which turns R back into a representation, which means in general we are not able to use unquote. Instead we need a program E that evaluates $\ulcorner P \urcorner$ to $\ulcorner R \urcorner$. The combination of *eval* and the representation we will call a self-evaluator. The two different self-interpretations are shown in Figure 2.

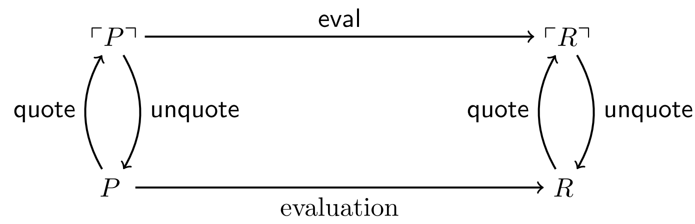


Figure 2: Self-Recognizer and Self-Evaluator. [11]

From Figure 2 we can do some diagram chasing to get some properties of our functions. For the quote and unquote function we note that $\mathsf{unquote} \circ \mathsf{quote}$ is the identity function for programs, and similarly $\mathsf{quote} \circ \mathsf{unquote}$ is the identity function for the range of quote. From this we can already conclude that both unquote and quote have to be injective. We also have $\mathsf{eval} = \mathsf{quote} \circ$ "evaluation" $\circ\, \mathsf{unquote}$ implying that the function eval on $\ulcorner P \urcorner$ halts if and only if P halts.

This definition of having two different self-interpreters, namely a self-recognizer and self-evaluator and the figure showing the difference, has been taken from Matt Brown and Jens Palsberg [11] and will be used as the definition of the self-interpreter for the rest of this paper.

4 Self-Interpreters For Untyped Lambda Calculus

In the previous section we have made an assertion for what a self-interpreter is. We have seen that there are fundamentally two parts, a representation and an interpreter. To apply this to untyped lambda calculus, first we will give a formal definition of a representation, self-recognizer and self-evaluator in untyped lambda calculus and the conclusion we can draw. After that we will discuss some representations, what interpreters they support and the difference between this on closed and open terms.

Why we need representation in the first place can be explained when we assume no representation, which is equivalent to assuming that quote is the identity function. Note that from a glance at Figure 2 we can already see that $\mathsf{quote} = \mathsf{unquote} = \mathsf{eval} = id_\Lambda$, turning them all into an equivalent term. Moreover, without representation we also lose the ability to do information analysis on terms.

Lemma 4.1.
The terms E_1, E_2 and E_3 do no exist such that

$$E_1\ M =_\beta \mathsf{true} \iff M \textit{ is in normal form} \qquad (\textit{for all } M \in \Lambda)$$

$$E_2\ M =_\beta \begin{cases} \overline{1} & \textit{if } M =_\alpha \lambda x\ .\ N_2 \textit{ for some } N_1 \in \Lambda \\ \overline{2} & \textit{if } M =_\alpha N_1\ N_2 \textit{ for some } M_1, N_2 \in \Lambda \\ \overline{3} & \textit{otherwise} \end{cases} \qquad (\textit{for all } M \in \Lambda)$$

$$E_3\ M_1\ M_2 =_\beta \mathsf{true} \iff M_1 =_\alpha M_2 \qquad (\textit{for all } M_1, M_2 \in \Lambda)$$

Proof.
For all cases, let $I = \lambda x\ .\ x$.
Case E_1

$$\mathsf{true} =_\beta E_1\ I\ =_\beta E_1\ (I\ I) =_\beta \mathsf{false} \qquad ↯$$

Case E_2

$$\overline{1} =_\beta E_2\ I =_\beta E_2\ (I\ I) =_\beta \overline{2} \qquad ↯$$

Case E_3

$$\mathsf{true} =_\beta E_3\ I\ I =_\beta E_3\ I\ (I\ I) =_\beta \mathsf{false} \qquad ↯$$

□

Therefore to have a representation of terms seems to be crucial. Now, before we can formally define what a self-interpreter is in untyped lambda calculus, we must define what it means to be a representation of a term. We have claimed that a representation of a terms is a value. Values in the case of (untyped) lambda calculus are term that are in normal form. Now by definition of the self-interpreter, we need some function such that every term is represented by a unique representation. We note that in lambda calculus we have α-equivalent terms, terms that are the same up to renaming. Since α-equivalence can disappear when we for example use De Bruijn representation, we will work modulo α-equivalences. This leads us to the following definition.

Definition 4.2.
An *encoding function* is a function $\ulcorner - \urcorner : \Lambda \to \Lambda$, such that for all $M, N \in \Lambda$ we have

- Injectivity up to alpha equivalence, i.e. $\ulcorner M \urcorner =_\alpha \ulcorner N \urcorner \Rightarrow M =_\alpha N$
- Normality, i.e. $\ulcorner M \urcorner$ is in normal form

The set $\{\ulcorner M \urcorner \mid M \in \Lambda\}$ is called a *representation* or an *encoding*.

Now that we have defined what a representation is, we can derive the definitions for both self-interpreters from Figure 2.

Definition 4.3.
A *self-recognizer* is a pair $(\ulcorner - \urcorner, \mathsf{unquote})$, where $\ulcorner - \urcorner : \Lambda \to \Lambda$ is a encoding function and $\mathsf{unquote} \in \Lambda$ an interpreter, such that the following holds for all $M \in \Lambda$

$$\mathsf{unquote}\ \ulcorner M \urcorner \twoheadrightarrow_\beta M$$

Definition 4.4.
A *self-evaluator* is a pair $(\ulcorner - \urcorner, E)$, where $\ulcorner - \urcorner : \Lambda \to \Lambda$ is a encoding function and $E \in \Lambda$ an interpreter, such that the following holds for all terms $M \in \Lambda$

$$E\ \ulcorner M \urcorner \twoheadrightarrow_\beta \ulcorner \mathbf{nf}(M) \urcorner \quad \text{if } M \text{ has a normal form}$$
$$E\ \ulcorner M \urcorner \text{ has no normal form} \quad \text{otherwise}$$

For the self-recognizer it is possible to take a shortcut. This is since after we apply the recognizer term, an evaluation of the term will happen no matter what. Therefore to have a recognizer it is enough to reduce to something that will reduce to the same thing as the term. Meaning, we only need to have beta-equivalence for our recognizer.

Definition 4.5.
A *weak self-recognizer* is a pair $(\ulcorner - \urcorner, \mathsf{unquote})$, where $\ulcorner - \urcorner : \Lambda \to \Lambda$ is a encoding function and $\mathsf{unquote} \in \Lambda$ an interpreter, such that the following holds for all $M \in \Lambda$:

$$\mathsf{unquote}\ \ulcorner M \urcorner =_\beta M$$

4.0.1 Quote as a term

You may have noticed that the encoding function itself is not defined as lambda term. If it is definable, then we have $\mathsf{eval} := \lambda x\ .\ \mathsf{quote}\ (\mathsf{unquote}\ x)$. Unfortunately quote is not definable in untyped lambda calculus.

Lemma 4.6.
The encoding function is not lambda definable, i.e. there does not exists a $\mathsf{quote} \in \Lambda$ *such that* $\mathsf{quote}\ M =_\beta \ulcorner M \urcorner$ *for all* $M \in \Lambda$.

Proof.

Let $I = \lambda x \,.\, x$. Assume, towards contradiction, that such a quote does exists. Then by definition of the encoding function we have $\ulcorner I\ I \urcorner \neq_\beta \ulcorner I \urcorner$. However, by definition of quote we also have:

$$\ulcorner I\ I \urcorner =_\beta \mathsf{quote}\ (I\ I) =_\beta \mathsf{quote}\ I =_\beta \ulcorner I \urcorner$$

However by definition of $\ulcorner - \urcorner$, we have $\ulcorner I\ I \urcorner \neq_\beta \ulcorner I \urcorner$, which leads us to the contradiction. □

This is unfortunate, as we cannot define the evaluator by just composing the quote and unquote terms together. However note that the quote term as defined in Lemma 4.6 is stronger than we need. For the composition to work we only need a quote term to turn terms into the representation of its normal form, given that it has one. This, though, is likely also impossible.

Conjecture 4.7. *If a encoding function* $\ulcorner - \urcorner$ *and term* unquote *form a self-recognizer then there is no term* nfquote *such that for all terms* M *that have a normal form we have*

$$\mathsf{nfquote}\ M =_\beta \ulcorner \boldsymbol{nf}(M) \urcorner$$

Therefore some form of a general quote term is impossible. This does not rule out some very specific kinds of quote function. In fact, for all the untyped lambda calculus representations we will define, we will have two very specific quote terms.

Definition 4.8. We say an encoding function $\ulcorner - \urcorner$ has the *double quote property* if there are two terms app and dquote for which we have

$$\mathsf{app}\ \ulcorner M \urcorner\ \ulcorner N \urcorner \twoheadrightarrow_\beta \ulcorner M\ N \urcorner$$
$$\mathsf{dquote}\ \ulcorner M \urcorner \twoheadrightarrow_\beta \ulcorner \ulcorner M \urcorner \urcorner$$

What is nice about this property is that we have a fixed-point theorem. This theorem is from Kleene [22] for which Barendregt [8] shows a variation for untyped lambda calculus. Here we show a proof based on one from Mogensen [25].

Lemma 4.9. *(Second Fixed-Point Theorem)*
Let $\ulcorner - \urcorner$ *be a encoding function with the double quote property, then for all terms* F *there exists a term* M *such that*

$$M \twoheadrightarrow_\beta F\ \ulcorner M \urcorner$$

Proof. Let F be some term. Now define M as follows:

$$M := A\ \ulcorner A \urcorner \qquad \text{where } A := \lambda n\,.\, F\ (\mathsf{app}\ n\ (\mathsf{dquote}\ n))$$

Then we have

$$
\begin{aligned}
& M \\
= \;& A \ulcorner A \urcorner \\
\to_\beta \;& F\ (\mathsf{app} \ulcorner A \urcorner\ (\mathsf{dquote} \ulcorner A \urcorner)) && \text{(Definition } A\text{)} \\
\twoheadrightarrow_\beta \;& F\ (\mathsf{app} \ulcorner A \urcorner \ulcorner\ulcorner A \urcorner\urcorner) && \text{(Definition } \mathsf{dquote}\text{)} \\
\twoheadrightarrow_\beta \;& F \ulcorner A \ulcorner A \urcorner\urcorner && \text{(Definition } \mathsf{app}\text{)} \\
= \;& F \ulcorner M \urcorner
\end{aligned}
$$

□

We will see that all our representations will have the double quote property and therefore they all have the second fixed-point theorem. This leads us to following informal conjecture:

Conjecture 4.10. *If a representation is effective it has the double quote property.*

4.0.2 Recursive function definition scheme

We have some properties that representations can have, however until now we have only looked at what we can do with representations, which we can also do without representations. To show that we can indeed do more with representations, we will make use of a function definition scheme, as defined by Geuvers[18].

Definition 4.11. We say a encoding function $\ulcorner - \urcorner$ has a *function definition scheme* if for all H_1, H_2 and H_3 there exists some term H such that we have

$$
\begin{aligned}
H \ulcorner x \urcorner &=_\beta H_1 \ulcorner x \urcorner\ H \\
H \ulcorner M\ N \urcorner &=_\beta H_2 \ulcorner M \urcorner \ulcorner N \urcorner\ H \\
H \ulcorner \lambda x\ .\ M \urcorner &=_\beta H_3\ (\lambda x\ .\ \ulcorner M \urcorner)\ H
\end{aligned}
$$

If a representation has such a function definition scheme, then we can create the first two terms given in Lemma 4.1 for the representation.

Lemma 4.12. *Let* $(\ulcorner - \urcorner, \mathsf{unquote})$ *be a self-recognizer where* $\mathsf{unquote}$ *is a closed term. Let* $\ulcorner - \urcorner$ *support a function definition scheme. Then we have a term that can distinguish a variable, abstraction or application.*

Proof. Let A_1, A_2 and A_3 be as follows

$$
\begin{aligned}
A_1 &:= \lambda x\ y\ .\ \overline{1} \\
A_2 &:= \lambda x\ y\ z\ .\ \overline{2} \\
A_3 &:= \lambda x\ y\ .\ \overline{3}
\end{aligned}
$$

then by the function definition scheme we have a term A for which every M we

have

$$A \ulcorner M \urcorner =_\beta \begin{cases} \overline{3} & \text{if } M =_\alpha \lambda x \ . \ N_2 \text{ for some } N_1 \in \Lambda \\ \overline{2} & \text{if } M =_\alpha N_1 \ N_2 \text{ for some } M_1, N_2 \in \Lambda \\ \overline{1} & \text{otherwise} \end{cases}$$

□

Corollary 4.13. *There are terms* isvar*,* isapp *and* isabs *such that*

$$\begin{aligned} \mathsf{isvar} \ \ulcorner M \urcorner =_\beta \mathsf{true} &\iff M = x \\ \mathsf{isapp} \ \ulcorner M \urcorner =_\beta \mathsf{true} &\iff M = N_1 \ N_2 \\ \mathsf{isabs} \ \ulcorner M \urcorner =_\beta \mathsf{true} &\iff M = \lambda x \ . \ \ N \end{aligned}$$

Proof. Follows from Lemma 4.12. □

Lemma 4.14. *Let* $(\ulcorner - \urcorner, \mathsf{unquote})$ *be a self-recognizer where* unquote *is a closed term. Let* $\ulcorner - \urcorner$ *support a function definition scheme. Then we have a NF checker term, i.e. a term* A *such that*

$$A \ \ulcorner M \urcorner \twoheadrightarrow_\beta \mathsf{true} \iff M \text{ is in normal form}$$

Proof. Let A_1, A_2 and A_3 be as follows

$$\begin{aligned} A_1 &:= \lambda x \ h \ . \ \mathsf{true} \\ A_2 &:= \lambda x \ y \ h \ . \ \mathsf{if} \ (\mathsf{isAbs} \ x) \ \mathsf{then} \ \mathsf{false} \ \mathsf{else} \ (\mathsf{and} \ (h \ x) \ (h \ y)) \\ A_3 &:= \lambda x \ h \ . \ (\lambda z \ . \ h \ (x \ z)) \ I \end{aligned}$$

Case for when M is in normal form
Base case $M = x$

$$A \ \ulcorner x \urcorner =_\beta A_1 \ x \ A \twoheadrightarrow_\beta \mathsf{true}$$

Inductive case $M = (\lambda x \ . \ N_1) \ N_2$

$$\begin{aligned} & A \ \ulcorner N_1 \ N_2 \urcorner \\ =_\beta \ & A_2 \ \ulcorner N_1 \urcorner \ulcorner N_2 \urcorner \ A \\ =_\beta \ & \mathsf{and} \ (A \ \ulcorner N_1 \urcorner) \ (A \ \ulcorner N_2 \urcorner) \\ = \ & \mathsf{true} \qquad\qquad \text{induction and reduction} \end{aligned}$$

Inductive case $M = (\lambda x \ . \ N_1) \ N_2$

$$
\begin{aligned}
& A \ulcorner \lambda x \,.\, N \urcorner \\
=_\beta\; & A_3\ (\lambda x \,.\, \ulcorner N \urcorner)\ A \\
=_\beta\; & (\lambda z \,.\, A((\lambda x \,.\, \ulcorner N \urcorner)\ z))\ I \\
=_\beta\; & (\lambda z \,.\, A \ulcorner N \urcorner [x := z])\ I \\
=_\alpha\; & (\lambda x \,.\, A \ulcorner N \urcorner)\ I \\
=_\beta\; & (\lambda x \,.\, \mathsf{true})\ I && \text{induction} \\
=_\beta\; & \mathsf{true}
\end{aligned}
$$

Case for when M is not in normal form

Base case: $M = (\lambda x \,.\, N_1)\ N_2$

$$
\begin{aligned}
A \ulcorner (\lambda x \,.\, N_1)\ N_2 \urcorner &=_\beta A_2 \ulcorner (\lambda x \,.\, N_1) \urcorner \ulcorner N_2 \urcorner\ A \\
&=_\beta \mathsf{if}\ (\mathsf{isAbs} \ulcorner (\lambda x \,.\, N_1) \urcorner)\ \mathsf{then\ false} \\
&\quad\ \mathsf{else}\ (\mathsf{and}\ (A \ulcorner (\lambda x \,.\, N_1) \urcorner)\ (A \ulcorner N_2 \urcorner)) \\
&=_\beta \mathsf{false}
\end{aligned}
$$

Induction case: $M = N_1\ N_2$

$$
\begin{aligned}
A \ulcorner N_1\ N_2 \urcorner &=_\beta A_2 \ulcorner (\lambda x \,.\, N_1) \urcorner \ulcorner N_2 \urcorner\ A \\
&=_\beta \mathsf{if}\ (\mathsf{isAbs} \ulcorner (\lambda x \,.\, N_1) \urcorner)\ \mathsf{then\ false} \\
&\quad\ \mathsf{else}\ (\mathsf{and}\ (A \ulcorner (\lambda x \,.\, N_1) \urcorner)\ (A \ulcorner N_2 \urcorner)) \\
&=_\beta \mathsf{if}\ (\mathsf{isAbs} \ulcorner (\lambda x \,.\, N_1) \urcorner)\ \mathsf{then\ false\ else\ false} \text{By IH} \\
&=_\beta \mathsf{false}
\end{aligned}
$$

Induction case: $M = \lambda x \,.\, N_1$

$$
\begin{aligned}
A \ulcorner \lambda x \,.\, N_1 \urcorner &=_\beta A_3 \ulcorner \lambda x \,.\, N_1 \urcorner\ A \\
&=_\beta (\lambda z \,.\, A\ (\ulcorner N_1 \urcorner\ z))\ I \\
&=_\beta (\lambda z \,.\, A \ulcorner N_1 \urcorner [x := z])\ I \\
&=_\beta (\lambda z \,.\, \mathsf{false})\ I && \text{By IH} \\
&=_\beta \mathsf{false}
\end{aligned}
$$

□

We find that a representation supporting a function definition scheme could be called strong, since we can already define multiple terms on it for analysis. In fact, when we have a term H such that $H \ulcorner x \urcorner \ =_\beta x$, then we can define a weak recognizer term. It seems that having a function definition scheme is the stronger requirement for a representation. We will investigate this further by presenting a self-recognizer that does not supports a function definition scheme.

Lemma 4.15. *Let* $\ulcorner - \urcorner$ *be a representation with a function definition scheme. Let* unvar *be a term such that*

$$\mathsf{unvar} \ulcorner x \urcorner =_\beta x$$

Then we have a weak recognizer

Proof.
Let unvar be as in the statement. Now let

$$\begin{aligned}\mathsf{unquote}_1 &:= \lambda x\ h\ .\ \mathsf{unvar}\ x\\ \mathsf{unquote}_2 &:= \lambda x\ y\ h\ .\ (h\ x)\ (h\ y)\\ \mathsf{unquote}_3 &:= \lambda x\ h\ .\ \lambda y\ .\ h\ (x\ y)\end{aligned}$$

Base case $M = x$

$$\begin{aligned}\mathsf{unquote}\ \ulcorner x \urcorner &=_\beta \mathsf{unvar}\ x\\ &=_\beta x\end{aligned}$$

Inductive case $M = N_1\ N_2$

$$\begin{aligned}\mathsf{unquote}\ \ulcorner N_1\ N_2 \urcorner &=_\beta (\mathsf{unquote}\ N_1)\ (\mathsf{unquote}\ N_2)\\ &=_\beta N_1\ N_2\end{aligned}$$

Inductive case $M = \lambda x\ .\ N$

$$\begin{aligned}\mathsf{unquote}\ \ulcorner \lambda x\ .\ N \urcorner &=_\beta \lambda z\ .\ \mathsf{unquote}\ N[x := z]\\ &=_{\alpha\beta} \lambda x\ .\ N\end{aligned}$$

□

4.0.3 Self-Evaluator

For the definition of an evaluator we have multiple options. The most straightforward option would be to define a term that reduces a single redex and then continuous applying this until we reach a normal form if there is one by making use of the function definition scheme and the fixed-point operator. There are two problems with going this method. The first one is practical, it is actually not obvious to define a single redex reduction term, and Mogensen believes that it the single redex reduction is more complex to define than his self-evaluator[25]. Secondly we have to deal with evaluation strategies, which may result into terms that have a normal form using the underlying evaluation not having it using the evaluator or visa versa. For example the term $(\lambda x\ .\ y)\ \Omega$ has a normal form if we go for left redex first reduction but not if we go for right redex first. Therefore if a term is actually an evaluator depends upon the underlying evaluation strategy.

Mogensen instead chooses to define an evaluator where the underlying evaluation mechanism handles the reduction. This results in a term which always uses the correct evaluation strategy such as to be considered an evaluator. To define the Mogensen evaluator we first need to assume some properties of the representation.

Definition 4.16. We say a representation is *modular* if there terms var, app and

abs such that

$$\mathsf{var}\ x =_\beta \ulcorner x \urcorner$$
$$\mathsf{app}\ \ulcorner M \urcorner\ \ulcorner N \urcorner =_\beta \ulcorner M\ N \urcorner$$
$$\mathsf{abs}\ \lambda x\ .\ \ulcorner M \urcorner =_\beta \ulcorner \lambda x\ .\ M \urcorner$$

The idea for the Mogensen evaluator is when we get into an application case $\ulcorner M\ N \urcorner$, we want to transform $\ulcorner M \urcorner$ into a function that takes $\ulcorner N \urcorner$ such that we get $\ulcorner M[x := N] \urcorner$. However there are cases where M is not an abstraction term. In that case we want $\ulcorner M \urcorner$ to be transformed into a function that makes an application node when given N.
Secondly if we do not hit an application, we do not want to return a function, but the representation. Mogensen solves these conflicting goals by returning a pair of a function and representation. The resulting evaluator we get is in Definition 4.17.

Definition 4.17. Let $\ulcorner - \urcorner$ be a modular representation with a function definition scheme, with a term *unvar* such that

$$\mathsf{unvar}\ \ulcorner x \urcorner =_\beta x$$

Then the Mogensen evaluator, E, is defined as follows:

$$E := \lambda m\ .\ \mathsf{snd}\ (\widehat{E}\ m)$$

Where $\widehat{E}$ is defined using the function definition scheme making use of the following terms

$$\widehat{E}_{var} := \lambda m\ h\ .\ \mathsf{unvar}\ x$$
$$\widehat{E}_{app} := \lambda m\ n\ h\ .\ (\mathsf{fst}\ (h\ m))\ (h\ n)$$
$$\widehat{E}_{abs} := \lambda m\ h\ .\ (\lambda g\ .\ \mathsf{pair}\ g\ (\mathsf{abs}\ (\lambda w\ .\ \mathsf{snd}\ (g\ (P\ \mathsf{var}\ w)))))\ (\lambda v\ .\ h\ (m\ v))$$

and where P is defined using the function definition scheme making use of the following terms

$$P_{var,abs} := \lambda m\ h\ .\ \mathsf{pair}\ (\lambda v\ .\ h\ (\mathsf{app}\ m\ (\mathsf{snd}\ v)))\ m$$
$$P_{app} := \lambda m\ n\ h\ .\ P_{var,abs}\ (\mathsf{app}\ m\ n)\ h$$

The evaluator works as follows; the final evaluator is the term E. This term uses the given representation and the term $\widehat{E}$ to get a pair of a function and representation, so we return the representation.
The term $\widehat{E}$ is split in three parts. The first part is the variable. Since we assume closed terms we know that this variable has been substituted, so we return this. For the application case we want the function, i.e. the first term in the pair, apply this to the evaluated second term.
Things get interesting when we hit an abstraction, here the function that does the reduction is build. P makes sure that when a application node has a variable in front, it builds an application node whenever it is applied to an argument.

Conjecture 4.18. *Let $\ulcorner - \urcorner$ be a modular representation with a Mogensen evaluator term E, then $(\ulcorner - \urcorner, E)$ forms a self-evaluator*

Proof. The proof ended up to be out of scope for the paper. Mogensen has a proof of his evaluator for the Mogensen representation, which could be adapted into a more general proof.[25] □

4.1 A Trivial Self-Recognizer

Before we go into the representations presented by Mogensen and Barendregt, we will look at a simple novel representation, based on the simple recognizer given by Brown and Palsberg[10]. This representation may be the simplest representation that supports a recognizer, and therefore we will call this representation the trivial self-recognizer.

Thinking about a simple recognizer, the first one we would reach for would be a recognizer that is the indentity term and where the encoding function would be $\ulcorner M \urcorner = M$ or $\ulcorner M \urcorner = \mathbf{nf}(M)$. This, though, is not possible by definition of the encoding functions, since the first example does not have normality, whereas the second is not injective.

To form a self-recognizer, we need a representation with a bit more. To get normality for an encoding function, we can make use of the fact that we can turn any redex irreducible by putting a free variable in front of it. Do this for all redexes and we make a term irreducible. Then at the end we can bind the free variable and then for any term we have an normal and unique representation. Now applying the identity term to a representation of a term would return us the term itself.

Definition 4.19. Let z be some variable not in M, then we define the trivial self-recognizer $(\ulcorner - \urcorner, \mathsf{unquote})$ as follows

$$\begin{aligned} \ulcorner M \urcorner &= \lambda z \ . \ \ulcorner M \urcorner_z && \text{Where } z \text{ is a fresh variable} \\ \ulcorner x \urcorner_z &= x \\ \ulcorner \lambda x \ . \ M \urcorner_z &= \lambda x \ . \ \ulcorner M \urcorner_z \\ \ulcorner M \ N \urcorner_z &= z \ \ulcorner M \urcorner_z \ \ulcorner N \urcorner_z \\ \mathsf{unquote} &= \lambda q \ . \ q \ I \end{aligned}$$

Lemma 4.20. $(\ulcorner - \urcorner, \mathsf{unquote})$ *forms a self-recognizer.*

Proof. First note that we have the following reduction:

$$\begin{aligned} \mathsf{unquote} \ \ulcorner M \urcorner &= (\lambda q \ . \ q \ I) \ulcorner M \urcorner \\ &\to_\beta \ulcorner M \urcorner \ I \\ &= (\lambda z \ . \ \ulcorner M \urcorner_z) \ I \\ &\to_\beta \ulcorner M \urcorner_z [z := I] \end{aligned}$$

Therefore we can finish the proof by proving $\ulcorner M \urcorner_z [z := I] \twoheadrightarrow_\beta M$, which we will do by using induction on the structure of the lambda term M.

Base case: $M = x$

$$\ulcorner x \urcorner_z [z := I] = x$$

Inductive case: $M = \lambda x \ . \ N$

$$\begin{aligned} \ulcorner \lambda x \ . \ N \urcorner_z [z := I] &= \lambda x \ . \ \ulcorner M \urcorner_z [z = I] \\ &\twoheadrightarrow_\beta \lambda x \ . \ M && \text{By IH} \end{aligned}$$

Inductive case: $M = N_1\ N_2$

$$\begin{aligned}\ulcorner N_1\ N_2 \urcorner_z[z := I] &= I\ \ulcorner N_1 \urcorner_z[z := I]\ \ulcorner N_2 \urcorner_z z = I] \\ &\to_\beta \ulcorner N_1 \urcorner_z[z := I]\ \ulcorner N_2 \urcorner_z[z := I] \\ &\twoheadrightarrow_\beta N_1\ N_2 && \text{By IH}\end{aligned}$$

□

Now we have a very simple representation that seems to support a recognizer but not a lot more. However we claimed in Section 4.0.1 that all our untyped lambda calculus representations will have the double quote property. Interestingly enough this simple representation has this as well.

Definition 4.21.

$$\begin{aligned}\mathsf{app} &:= \lambda m\ n\ .\ \lambda x\ .\ x\ (m\ x)\ (n\ x) \\ \mathsf{dquote} &:= \lambda m\ .\ \lambda w\ z\ .\ m\ (\lambda a\ b\ .\ w\ (w\ z\ a)\ b)\end{aligned}$$

Lemma 4.22. *For all terms M and N we have*

$$\mathsf{app}\ \ulcorner M \urcorner\ \ulcorner N \urcorner \twoheadrightarrow_\beta \ulcorner M\ N \urcorner$$

Proof.

$$\begin{aligned}\mathsf{app}\ \ulcorner M \urcorner\ \ulcorner N \urcorner &\twoheadrightarrow_\beta \lambda x\ .\ x\ (\underline{\ulcorner M \urcorner_z\ x})\ (\underline{\ulcorner N \urcorner_z\ x}) \\ &\twoheadrightarrow_\beta \lambda x\ .\ x\ (\ulcorner M \urcorner_z[z := x])\ (\ulcorner N \urcorner_z[z := x]) \\ &=_\alpha \lambda z\ .\ z\ \ulcorner M \urcorner_z\ \ulcorner N \urcorner_z \\ &= \ulcorner M\ N \urcorner\end{aligned}$$

□

Lemma 4.23. *For all terms M we have*

$$\mathsf{dquote}\ \ulcorner M \urcorner =_\beta \ulcorner\ulcorner M \urcorner\urcorner$$

Proof.
Let $D := \lambda a\ b\ .\ w\ (w\ z\ a)\ b$, then we have

$$\mathsf{dquote}\ \ulcorner M \urcorner \twoheadrightarrow_\beta \lambda w\ z\ .\ \ulcorner M \urcorner_z[z := D]$$

Therefore to complete the proof, we require that

$$\ulcorner M \urcorner_z[z := (\lambda a\ b\ .\ w\ (w\ z\ a)\ b)] =_\beta \ulcorner\ulcorner M \urcorner_z \urcorner_w \tag{1}$$

Which we prove using induction on the structure of the term.
Base case

$$\ulcorner x \urcorner_z[z := D] =_\beta x =_\beta \ulcorner\ulcorner x \urcorner_z \urcorner_w$$

Inductive case

$$\begin{aligned}\ulcorner \lambda x\ .\ M \urcorner_z[z := D] &=_\beta \lambda x\ .\ \ulcorner M \urcorner_z[z := D] \\ &=_\beta \lambda x\ .\ \ulcorner\ulcorner M \urcorner_z \urcorner_w && \text{By IH} \\ &= \ulcorner\ulcorner \lambda x\ .\ M \urcorner_z \urcorner_w\end{aligned}$$

Inductive case

$$\begin{aligned}\ulcorner M\ N \urcorner_z[z := D] &=_\beta D\ \ulcorner M \urcorner_{\mathsf{id}}[z := D]\ \ulcorner N \urcorner_z[z := D] \\ &=_\beta w\ (w\ z\ \ulcorner M \urcorner_z[\mathsf{id} := D])\ \ulcorner N \urcorner_z[z := D]) \\ &=_\beta w\ (w\ z\ \ulcorner\ulcorner M \urcorner_z \urcorner_w)\ (\ulcorner\ulcorner N \urcorner_z \urcorner_w) && \text{By IH} \\ &=_\beta w\ (w\ z\ \ulcorner\ulcorner M \urcorner_z \urcorner_w)\ (\ulcorner\ulcorner N \urcorner_z \urcorner_w) \\ &= w\ \ulcorner z\ \ulcorner M \urcorner_z \urcorner_w\ \ulcorner\ulcorner N \urcorner_z \urcorner_w \\ &= \ulcorner z\ \ulcorner M \urcorner_z\ \ulcorner N \urcorner_z \urcorner_w \\ &= \ulcorner\ulcorner M\ N \urcorner_z \urcorner_w\end{aligned}$$

□

Corollary 4.24. *The representation $\ulcorner M \urcorner$, as in Definition 4.19, has the double quote property.*

While having a trivial self-recognizer is informative, the question remains if we can do even more with it and what its limitations are. We find that indeed this representation is not as powerful as later representations we will look at. For example it is unable to check if an encoded term is an abstraction, application or variable, which also implies that it does not have a function definition scheme.

Lemma 4.25. *The trivial self-recognizer does not support a term* isAbs *such that for all terms M we have*

$$\mathsf{isAbs}\ \ulcorner M \urcorner =_\beta \mathsf{true} \iff M = \lambda x\ .\ N \text{ for some term } N$$

Proof.
Towards contradiction, let's assume there is a term isAbs such as in the statement, then we get the following equivalence

$$\begin{aligned}\mathsf{true} &=_\beta \mathsf{isAbs}\ \ulcorner \lambda x\ .\ x \urcorner \\ &= \mathsf{isAbs}\ \lambda z\ .\ \lambda x\ .\ x \\ &=_\beta (\mathsf{isAbs}\ \lambda z\ .\ y)[y := \lambda x\ .\ x] \\ &= (\mathsf{isAbs}\ \lambda z\ .\ \ulcorner y \urcorner_z)[y := \lambda x\ .\ x] \\ &= (\mathsf{isAbs}\ \ulcorner y \urcorner)[y := \lambda x\ .\ x] \\ &=_\beta \mathsf{false}[y := \lambda x\ .\ x] \\ &=_\beta \mathsf{false} && ↯\end{aligned}$$

□

Corollary 4.26. *The trivial self-recognizer does not support a function definition scheme.*

4.2 Mogensen Self-Interpreter

Now that we have looked at a weak representation, we are interested in stronger representations. In this section we will present the representation defined by Mogensen [25] and will look at the strength of this representation. We will find that it support both a recognizer and an evaluator. Moreover it will support a function definition scheme, from which we can conclude that it too has the double quote property.

Definition 4.27. The Mogensen encoding function $\ulcorner - \urcorner_m$ is defined as follows [25]

$$\begin{aligned} \ulcorner x \urcorner_m &= \lambda e \, . \, e \; U_1^3 \; x \; e \\ \ulcorner M \; N \urcorner_m &= \lambda e \, . \, e \; U_2^3 \; \ulcorner M \urcorner_m \; \ulcorner N \urcorner_m \; e \\ \ulcorner \lambda x . M \urcorner_m &= \lambda e \, . \, e \; U_3^3 \; (\lambda x \, . \, \ulcorner M \urcorner_m) \; e \end{aligned}$$

Theorem 4.28. $\ulcorner - \urcorner_m$ *is a encoding function*

Proof. To prove that Mogensen code is a encoding function, we have to prove that it has normality and is injective up to alpha equivalence. Normality can be proven by induction on the terms.

Base case
For all variables x, we have that $\ulcorner x \urcorner_m = \lambda e \, . \, e \; U_1^3 \; x \; e$ has no beta redexes, therefore it is in normal form.

Application case
For all terms M, N, we have that $\ulcorner M \; N \urcorner_m = \lambda e \, . \, e \; U_2^3 \; \mathfrak{m}(M) \; \ulcorner N \urcorner_m \; e$. By induction hypothesis we have that $\ulcorner M \urcorner_m$ and $\ulcorner N \urcorner_m$ are in normal form, therefore $\lambda e \, . \, e \; U_2^3 \; \ulcorner M \urcorner_m \; \ulcorner N \urcorner_m \; e$ has no beta redexes and therefore is also in normal form.

Abstraction case
For all terms M, we have that $\ulcorner \lambda x \, . \, M \urcorner_m = \lambda e \, . \, e \; U_3^3 \; (\lambda x \, . \, \ulcorner M \urcorner_m) \; e$. By induction hypothesis we have that $\ulcorner M \urcorner_m$ is in normal form, therefore there are no beta redexes $\lambda x . \ulcorner M \urcorner_m$ and so there are none in $\lambda e \, . \, e \; U_3^3 \; (\lambda x . \ulcorner M \urcorner_m) \; e$, and therefore it is in normal form.

Injectivity we can prove by case distinction.
Let M, N be some terms such that $M \neq_\alpha N$. Towards contradiction, assume $\ulcorner M \urcorner_m =_\alpha \ulcorner N \urcorner_m$. If M is some variable, then $\ulcorner M \urcorner_m = \lambda e \, . \, e \; U_1^3 \; x \; e =_\alpha \ulcorner N \urcorner_m$.
So therefore $N =_\alpha x$, however since x is free, we have $N =_\alpha M$ ↯.
The argument is similar for the other two cases. $\square$

Now that we have defined the Mogensen definition and proven that it indeed fits our definition for an representation, we can go on to prove the properties of it. The first thing we find is that this representation forms a self-recognizer.

Definition 4.29. The recognizer[25] for the Mogensen representation is defined as follows

$$\mathsf{unquote}_m := \lambda a \, . \, a \; (\lambda b \, . \, b \; K \; S \; C)$$

Where

$$K := \lambda xy\ .\ x,$$
$$S := \lambda xyz\ .\ xz(yz),$$
$$C := \lambda xyz\ .\ xzy$$

Theorem 4.30. *The pair* $(\ulcorner - \urcorner_m, \mathsf{unquote}_m)$ *forms a self-recognizer.*

Proof. First we show, by induction on the structure of lambda terms, that $\ulcorner M \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) \twoheadrightarrow_\beta M$.

Base case

$$\begin{aligned}
\ulcorner x \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) &\rightarrow_\beta (\lambda b\ .\ b\ K\ S\ C)\ U_1^3\ x\ (\lambda b\ .\ b\ K\ S\ C) \\
&\rightarrow_\beta (U_1^3\ K\ S\ C)\ x\ (\lambda b\ .\ b\ K\ S\ C) \\
&\twoheadrightarrow_\beta K\ x\ (\lambda b\ .\ b\ K\ S\ C) \\
&\twoheadrightarrow_\beta x
\end{aligned}$$

Application case

$$\begin{aligned}
\ulcorner M\ N \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) &\rightarrow_\beta (\lambda b\ .\ b\ K\ S\ C)\ U_2^3\ \ulcorner M \urcorner_m\ \ulcorner N \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) \\
&\rightarrow_\beta (U_2^3\ K\ S\ C)\ \ulcorner M \urcorner_m\ \ulcorner N \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) \\
&\twoheadrightarrow_\beta S\ \ulcorner M \urcorner_m\ \ulcorner N \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) \\
&\twoheadrightarrow_\beta \ulcorner M \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C)\ (\ulcorner N \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C)) \\
&\twoheadrightarrow_\beta M\ N
\end{aligned}$$

Abstraction case

$$\begin{aligned}
\ulcorner \lambda x\ .\ M \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) &\rightarrow_\beta (\lambda b\ .\ b\ K\ S\ C)\ U_3^3\ (\lambda x\ .\ \ulcorner M \urcorner_m)\ (\lambda b\ .\ b\ K\ S\ C) \\
&\twoheadrightarrow_\beta C\ (\lambda x\ .\ \ulcorner M \urcorner_m)\ (\lambda b\ .\ b\ K\ S\ C) \\
&\twoheadrightarrow_\beta \lambda z\ .\ (\lambda x\ .\ \ulcorner M \urcorner_m)\ z\ (\lambda b\ .\ b\ K\ S\ C) \\
&\rightarrow_\beta \lambda z\ .\ \ulcorner M \urcorner_m[x := z]\ (\lambda b\ .\ b\ K\ S\ C) \\
&=_\alpha \lambda x\ .\ \ulcorner M \urcorner_m\ (\lambda b\ .\ b\ K\ S\ C) \\
&\twoheadrightarrow_\beta \lambda x\ .\ M
\end{aligned}$$

Now with this fact in our pocket, we can take any $M \in \Lambda$ and then we have

$$\mathsf{unquote}_m \ulcorner M \urcorner_m \rightarrow_\beta \ulcorner M \urcorner_m (\lambda b\ .\ b\ K\ S\ C) \twoheadrightarrow_\beta M$$

□

Mogensen does not only present a recognizer for his representation, but also an evaluator, proving that it also forms a self-evaluator, therefore having our first representation that forms a self-interpreter.

Definition 4.31. Evaluator for the Mogensen encoding. [25, §4 Self-reduction]

$$E_m := \lambda m \;.\; R\; m\; (\lambda a\; b \;.\; b)$$

where

$$
\begin{array}{lll}
R & := & Y\; \lambda r\; m \;.\; m\;\; (\lambda x \;.\; x) \\
 & & \qquad\qquad (\lambda m\; n \;.\; (r\; m)\; (\lambda a\; b\; a)\; (r\; n)) \\
 & & \qquad\qquad (\lambda m \;. \\
 & & \qquad\qquad\quad (\lambda g \;.\; \lambda x \;.\;\; x\; g \\
 & & \qquad\qquad\qquad\qquad (\lambda a\; b\; c \;.\;\; c \\
 & & \qquad\qquad\qquad\qquad\qquad (\lambda w\;\; g \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (P\; \lambda a\; b\; c \;.\; a\; w) \\
 & & \qquad\qquad\qquad\qquad\qquad\quad (\lambda a\; b \;.\; b) \\
 & & \qquad\qquad\qquad\qquad\qquad) \\
 & & \qquad\qquad\qquad\qquad) \\
 & & \qquad\qquad\quad) \\
 & & \qquad\qquad\quad \lambda v \;.\; r\; (m\; v) \\
 & & \qquad\qquad)
\end{array}
$$

4.2.1 Function Definition Scheme

Before we prove that Mogensen representation also has the double quote property, we will present that it supports a function definition scheme. Proving this will not only show that it has properties that the trivial self-recognizer does not have, but will also greatly help in defining and proving correctness of the `dquote` term.

Proposition 4.32. *Let $A_1, A_2, A_3 \in \Lambda$. Then there is an $H \in \Lambda$ such that*

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

Proof.
Let $H = \lambda x \;.\; x\; (\lambda y \;.\; y\; B_1\; B_2\; B_3)$ where

$$
\begin{aligned}
B_1 &= \lambda x\; y \;.\; A_1\; (\lambda e \;.\; U_1^3\; x\; e)\; (\lambda a \;.\; a\; y) \\
B_2 &= \lambda x\; y\; z \;.\; A_2\; x\; y\; (\lambda a \;.\; a\; z) \\
B_3 &= \lambda x\; y \;.\; A_3\; x\; (\lambda z \;.\; z\; y)
\end{aligned}
$$

Then H forms the term H as in the statement.

Variable case

$$\begin{aligned} H \ulcorner x \urcorner_m &=_\beta \ulcorner x \urcorner_m \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \ U_1^3 \ x \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta B_1 \ x \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta A_1 \ (\lambda e \ . \ U_1^3 \ x \ e) \ H \\ &= A_1 \ulcorner x \urcorner_m \ H \end{aligned}$$

Application case

$$\begin{aligned} H \ulcorner N_1 \ N_2 \urcorner_m &=_\beta \ulcorner N_1 \ N_2 \urcorner_m \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \ U_2^3 \ulcorner N_1 \urcorner_m \ulcorner N_2 \urcorner_m \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta B_2 \ulcorner N_1 \urcorner_m \ulcorner N_2 \urcorner_m \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta A_2 \ulcorner N_1 \urcorner_m \ulcorner N_2 \urcorner_m \ H \end{aligned}$$

Abstraction case

$$\begin{aligned} H \ulcorner \lambda x \ . \ M \urcorner_m &=_\beta \ulcorner \lambda x \ . \ N \urcorner_m \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \ U_3^3 \ (\lambda x \ . \ulcorner N \urcorner_m) \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta B_3 \ (\lambda x \ . \ulcorner N \urcorner_m) \ (\lambda b \ . \ b \ B_1 \ B_2 \ B_3) \\ &=_\beta A_3 \ (\lambda x \ . \ulcorner N \urcorner_m) \ H \end{aligned}$$

□

4.2.2 double quote property

The double quote property together with the second fixed-point theorem are proven by Mogensen in his paper ”Efficient Self-Interpretation in Lambda Calculus” [25]. Here we present his proof by using the function definition scheme.

Lemma 4.33. *There is a term* dquote *such that*

$$\mathsf{dquote} \ \ulcorner M \urcorner =_\beta \ulcorner \ulcorner M \urcorner \urcorner$$

Proof. First let

$$\begin{aligned} \mathsf{var} &:= \lambda x \ . \ \lambda e \ . \ e \ U_1^3 \ x \ e \\ \mathsf{app} &:= \lambda x \ y \ . \ \lambda e \ . \ e \ U_2^3 \ x \ y \ e \\ \mathsf{abs} &:= \lambda x \ \lambda e \ \ . \ \ e \ (\lambda y \ . \ U_3^3 \ x) \ e \end{aligned}$$

Then we have for these terms the following property:

$$\begin{aligned} &\mathsf{var} \ x \twoheadrightarrow_\beta \ulcorner x \urcorner \\ &\mathsf{app} \ulcorner M \urcorner \ulcorner N \urcorner \twoheadrightarrow_\beta \ulcorner M \ N \urcorner \\ &\mathsf{abs} \ \lambda x \ . \ulcorner M \urcorner \twoheadrightarrow_\beta \ulcorner \lambda x \ . \ M \urcorner \end{aligned}$$

Define $\mathsf{dquote}_1, \mathsf{dquote}_2$ and dquote_3 as follows:

$$\mathsf{dquote}_1 := \lambda x\ h\ .\ \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{var}\ e)\ \ulcorner U_1^3 \urcorner)\ x)\ (\mathsf{var}\ e)$$
$$\mathsf{dquote}_2 := \lambda x\ y\ h\ .\ \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{var}\ e)\ \ulcorner U_2^3 \urcorner)\ (h\ x))\ (h\ y))\ (\mathsf{var}\ e)$$
$$\mathsf{dquote}_3 := \lambda x\ h\ .\ \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{var}\ e)\ \ulcorner U_2^3 \urcorner)(\lambda y\ .\ (h\ (x\ y))))\ (\mathsf{var}\ e)$$

Let dquote be the term we get from the function definition scheme with terms dquote_1, dquote_2 and dquote_3. Then we have

Base case $M = x$

$$\begin{aligned} \mathsf{dquote}\ \ulcorner x \urcorner &=_\beta \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{var}\ e)\ \ulcorner U_1^3 \urcorner)\ \ulcorner x \urcorner)\ (\mathsf{var}\ e) \\ &=_\beta \ulcorner \lambda e\ .\ e\ U_1^3\ .\ \ulcorner x \urcorner\ e \urcorner \\ &= \ulcorner\ulcorner x \urcorner\urcorner \end{aligned}$$

Inductive case $M = N_1\ N_2$

$$\begin{aligned} \mathsf{dquote}\ \ulcorner N_1\ N_2 \urcorner &=_\beta \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ \mathsf{app}\ ((\mathsf{var}\ e)\ \ulcorner U_2^3 \urcorner)\ (\mathsf{dquote}\ \ulcorner N_1 \urcorner)) \\ &\qquad (\mathsf{dquote}\ \ulcorner N_2 \urcorner))\ (\mathsf{var}\ e) \\ &=_\beta \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ \mathsf{app}\ ((\mathsf{var}\ e)\ \ulcorner U_2^3 \urcorner)\ \ulcorner\ulcorner N_1 \urcorner\urcorner)\ \ulcorner\ulcorner N_2 \urcorner\urcorner) \\ &\qquad (\mathsf{var}\ e) \\ &=_\beta \ulcorner \lambda e\ .\ e\ U_2^3\ \ulcorner N_1 \urcorner\ \ulcorner N_2 \urcorner\ e \urcorner \\ &= \ulcorner\ulcorner N_1\ N_2 \urcorner\urcorner \end{aligned}$$

Inductive case $M = \lambda x\ .\ N$

$$\begin{aligned} \mathsf{dquote}\ \ulcorner \lambda z\ .\ N \urcorner &=_\beta \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{var}\ e)\ \ulcorner U_2^3 \urcorner) \\ &\qquad (\lambda y\ .\ (\mathsf{dquote}\ \ulcorner N \urcorner[z := y])))\ (\mathsf{var}\ e) \\ &=_{\alpha\beta} \mathsf{abs}\ \lambda e\ .\ \mathsf{app}\ (\mathsf{app}\ (\mathsf{app}\ (\mathsf{var}\ e)\ \ulcorner U_2^3 \urcorner) \\ &\qquad (\lambda y\ .\ \ulcorner\ulcorner N \urcorner\urcorner)\ (\mathsf{var}\ e) \\ &=_\beta \ulcorner \lambda e\ .\ e\ U_3^3\ (\lambda z\ .\ \ulcorner N \urcorner)\ e \urcorner \\ &= \ulcorner\ulcorner \lambda x\ .\ M \urcorner\urcorner \end{aligned}$$

□

Corollary 4.34. *The Mogensen representation has the double quote property and therefore a second fixed-point theorem.*

4.3 Closed Term Self-Interpreters

For this section we will look at a different type of self-interpreter. The Mogensen one is a nice self-interpreter, however it is very much defined in lambda terms. Historically arithmetization has been important for computability proofs. For example Church used Gödel numbering to represent untyped lambda calculus to show for the first time that there are arithmetic problems that is not solvable [14]. We will see that a very important difference between those representations based on numbers and the representations we have already seen is that numbers are closed-term, i.e. they do not have free variables. Therefore we will generalize these representations to closed-term representations.

Definition 4.35.
A *closed-term encoding function* is a encoding function $\ulcorner - \urcorner$ as in Definition 4.2 such that for all terms M we have $\ulcorner M \urcorner$ is closed.

The set $\{\ulcorner M \urcorner \mid M \in \Lambda\}$ is called a *closed-term representation* or a *closed-term encoding*.

Unfortunately such a representation does not have a self-recognizer such as we have seen before.

Lemma 4.36. *Let $\ulcorner - \urcorner$ be a closed-term encoding function, then it does not have recognizer term as in Definition 4.3.*

Proof.
Assume, towards contradiction, that there exists some term $\mathsf{unquote}$ such that $\mathsf{unquote}\ \ulcorner M \urcorner_g =_\beta M$ for all $M \in \Lambda$. Now take a $x \notin FV(\mathsf{unquote})$ and let M be a some term where $M \neq_\beta x$, then by assumption we have

$$
\begin{aligned}
M &=_\beta x[x := M] \\
&=_\beta (\mathsf{unquote}\ \ulcorner x \urcorner)[x := M] \\
&= \mathsf{unquote}[x := M]\ \ulcorner x \urcorner[x := M] \\
&= \mathsf{unquote}\ \ulcorner x \urcorner[x := M] && \text{By } x \notin \mathbf{FV}(\mathsf{unquote}) \\
&= \mathsf{unquote}\ \ulcorner x \urcorner && \text{By } \ulcorner x \urcorner \text{ is closed} \\
&=_\beta x && \lightning
\end{aligned}
$$

□

Instead for closed-term representations we will define the self-recognizer as a self-recognizer limited to only closed-terms.

Definition 4.37.
A *closed-term self-recognizer* is a pair $(\ulcorner - \urcorner, \mathsf{unquote})$, where $\ulcorner - \urcorner : \Lambda \to \Lambda_c$ is a closed-term encoding function and $\mathsf{unquote} \in \Lambda$ an interpreter, such that the following holds for all closed terms $M \in \Lambda_c$

$$\mathsf{unquote}\ \ulcorner M \urcorner \twoheadrightarrow_\beta M$$

One of the nice features of closed-term self-recognizer is that we often have an equality term. This is clear when we use numbers as our representation of choice.

Definition 4.38. We say a representation has an equality term if there exists a term eq such that

$$\mathsf{eq}\ \ulcorner M \urcorner\ \ulcorner N \urcorner =_\beta \mathsf{true} \iff \ulcorner M \urcorner =_\beta \ulcorner N \urcorner \qquad \text{(for all } M, N \in \Lambda)$$

With an equality term also comes a mapping term. A mapping term will be very useful when proving properties of the representation, at least useful enough that we will often assume that such an equality term exists.

Corollary 4.39. *When we have a closed-term representation with an equality term, then we have an mapping term, i.e. a term* map *such that*

$$\mathsf{map}\ f\ x\ a\ b =_\beta \begin{cases} x & \textit{if } a = b \\ f\ b & \textit{otherwise} \end{cases}$$

Proof.
Given that we have a equality term, we can define the following term:

$$\mathsf{map} = \lambda f\ x\ a\ b\ .\ (\mathsf{eq}\ a\ b)\ (x)\ (f\ b)$$

which is the mapping term. □

4.3.1 Closed-Term Function Definition Schemes

For the closed-term representations we would like to have function definition schemes like we have seen before in the open-term representations. However when we look at Definition 4.11, we will find a problem with the abstraction case. With an open term self-recognizer, when some variable x is in de term, then it is also there in the encoded term. Therefore $\lambda x\ .\ \ulcorner x \urcorner$ binds the x in a open term self-recognizer. This is not the case in closed-term representations. Therefore we will be in need for a different function definition scheme, here take from Geuvers [18].

Definition 4.40. We say that a closed-term representation has a *closed-term function definition scheme*, if for all terms A_1, A_2, A_3 there exists a term H such that

$$\begin{aligned} H\ \ulcorner x_i \urcorner_b &=_\beta A_1\ \ulcorner x_i \urcorner_b\ H \\ H\ \ulcorner M\ N \urcorner_b &=_\beta A_2\ \ulcorner M \urcorner_b\ \ulcorner N \urcorner_b\ H \\ H\ \ulcorner \lambda x_i\ .\ M \urcorner_b &=_\beta A_3\ \ulcorner x_i \urcorner_b\ \ulcorner M \urcorner_b\ H \end{aligned}$$

The function definition scheme defined above is not inferior to the open-term variant, as we can adapt the lemmas above we have seen for the open-term variant to the variant presented.

Lemma 4.41. *A closed representation with a equality term and a closed-term function definition scheme has a weak recognizer.*

Proof. Let

$$\begin{aligned}
\mathsf{unquote}_1 &:= \lambda m\ h\ f\ .\ f\ m \\
\mathsf{unquote}_2 &:= \lambda m\ n\ h\ f\ .\ (h\ f\ m)\ (h\ f\ n) \\
\mathsf{unquote}_3 &:= \lambda x\ m\ h\ f\ .\ \lambda z\ .\ h\ (\mathsf{map}\ f\ x\ z)\ m
\end{aligned}$$

Let the term $\widehat{\mathsf{unquote}}$ be the term we get from making use of the function definition scheme. The we have

$$\widehat{\mathsf{unquote}}\ \ulcorner M \urcorner\ F =_\beta M[x_1 := F\ \ulcorner x_1 \urcorner, \ldots, x_n := F\ \ulcorner x_n \urcorner]$$

where $\{x_1, \ldots, x_n\} = \mathbf{FV}(M)$. We prove this by induction on M.

Base case $M = x$

$$\begin{aligned}
\widehat{\mathsf{unquote}}\ \ulcorner x_i \urcorner\ F &=_\beta F\ \ulcorner x_i \urcorner \\
&=_\beta x_i[x_i := F\ \ulcorner x_i \urcorner]
\end{aligned}$$

Inductive case $M = N_1\ N_2$
Let $\{x_1, \ldots, x_n\} = \mathbf{FV}(M)$, $\{y_1, \ldots, y_m\} = \mathbf{FV}(N)$ and $\{z_1, \ldots z_k\} = \mathbf{FV}(M\ N)$.
Note that $\mathbf{FV}(N_1\ N_2) = \mathbf{FV}(N_1) \cup \mathbf{FV}(N_2)$.

$$\begin{aligned}
& \widehat{\mathsf{unquote}}\ \ulcorner N_1\ N_2 \urcorner\ F \\
=_\beta\ & \widehat{\mathsf{unquote}}\ \ulcorner N_1 \urcorner\ F\ (\widehat{\mathsf{unquote}}\ \ulcorner N_2 \urcorner\ F) \\
=_\beta\ & N_1[x_1 := F\ \ulcorner x_1 \urcorner, ..., x_m := F\ \ulcorner x_m \urcorner] \\
& \quad (N_2[y_1 := F\ \ulcorner y_1 \urcorner, ..., y_n := F\ \ulcorner y_n \urcorner]) && \text{By IH} \\
=\ & (N_1\ N_2)[z_1 := F\ \ulcorner z_1 \urcorner, ..., z_k := F\ \ulcorner z_k \urcorner]
\end{aligned}$$

Inductive case $M = \lambda x\ .\ N$
Let $\{y_1, \ldots, y_n\} = \mathbf{FV}(\lambda x_i\ .\ N)$. Note that $\mathbf{FV}(N) = \mathbf{FV}(N) \cup \{x_i\}$.

$$\begin{aligned}
& \widehat{\mathsf{unquote}}\ \ulcorner \lambda x_i\ .\ M \urcorner\ F \\
=_\beta\ & \lambda x_i\ .\ \widehat{\mathsf{unquote}}\ \ulcorner M \urcorner\ F' && \text{where } F' := \mathsf{map}\ F\ x_i\ \ulcorner x_i \urcorner \\
=_\beta\ & \lambda x_i\ .\ M[x_i := F',\ \ulcorner x_i \urcorner, y_1 := F'\ \ulcorner y_1 \urcorner, ..., y_n := F'\ \ulcorner y_n \urcorner] && \text{By IH} \\
=_\beta\ & \lambda x_i\ .\ M[x_i := x_i, y_1 := F\ \ulcorner y_1 \urcorner, ..., y_n := F\ \ulcorner y_n \urcorner] && \text{Corollary 4.39} \\
=\ & (\lambda x_i\ .\ M)[x_1 := F\ \ulcorner x_1 \urcorner, ..., x_n := F\ \ulcorner x_n \urcorner]
\end{aligned}$$

Now define $\mathsf{unquote} := \lambda m\ .\ \widehat{\mathsf{unquote}}\ m\ I$ and let M be some closed term. Then we have

$$\begin{aligned}
\mathsf{unquote}\ \ulcorner M \urcorner &=_\beta M[x_1 := F\ x_1, \ldots, F\ x_n] \\
&= M && \text{By } M \text{ closed}
\end{aligned}$$

□

4.4 Barendregt Self-Interpreter

An example of a closed-term self-interpreter we will look at is defined by Barendregt[5].

Definition 4.42. The Barendregt encoding function is as follows defined: [5]

$$\begin{aligned}
\ulcorner x_i \urcorner_b &= \underline{\langle 0, i \rangle} \\
\ulcorner M\ N \urcorner_b &= \underline{\langle 1, \langle \ulcorner M \urcorner_b, \ulcorner N \urcorner_b \rangle \rangle} \\
\ulcorner \lambda x_i\ .\ M \urcorner_b &= \underline{\langle 2, \langle \ulcorner x_i \urcorner, \ulcorner M \urcorner_b \rangle \rangle}
\end{aligned}$$

Where $\langle -, - \rangle$ is some computable injective function that maps pairs of natural numbers to a single natural number.

A nice thing about this definition is that we can lend some theorems from computational theory and get some terms and properties for free.

Lemma 4.43. *For the Barendregt representation we have the following properties and terms.*

1. $\mathsf{eq}\ \ulcorner M \urcorner_b\ \ulcorner N \urcorner_b =_\beta \mathsf{true} \iff \ulcorner M \urcorner_b = \ulcorner N \urcorner_b$
2. *The double quote property and therefore a second fixed-point theorem.*
3. *Projection terms* Π_1 *and* Π_2 *such that* $\Pi_1 \langle m, n \rangle =_\beta m$ *and* $\Pi_1 \langle m, n \rangle =_\beta n$

Proof. This follows from that Barendregt encoding function is computable and that lambda calculus is Turing complete. □

4.4.1 Function Definition Scheme

Proposition 4.44. *Let* $A_1, A_2, A_3 \in \Lambda$*. Then there is an* $H \in \Lambda$ *such that*

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

Proof. We give the recursive equations for H.

$$\begin{aligned}
H\ \langle 0, y \rangle &= A_1\ \langle 0, y \rangle\ H \\
H\ \langle 1, y \rangle &= A_2\ (\Pi_1\ y)\ (\Pi_2\ y)\ H \\
H\ \langle 2, y \rangle &= A_3\ (\Pi_1\ y)\ (\Pi_2\ y)\ H
\end{aligned}$$

Which means that the following term H does the job:

$$
\begin{aligned}
H = Y(\lambda f\ .\ \lambda x\ .\ & \\
& \mathsf{if}\ (\mathsf{eq}\ (\Pi_1\ x)\ \overline{0})\ (A_1\ f\ x) \\
& \mathsf{elif}\ (\mathsf{eq}\ (\Pi_1\ x)\ \overline{1})\ (A_2\ (\Pi_1\ (\Pi_2\ x))\ (\Pi_2\ (\Pi_2\ x))\ f) \\
& \mathsf{else}\ (A_3\ (\Pi_1\ (\Pi_2\ x))\ (\Pi_2\ (\Pi_2\ x))\ f) \\
&)
\end{aligned}
$$

□

Corollary 4.45. *The Barendregt representation has a weak self-recognizer*

Proof. Follows from Lemma 4.43 and Proposition 4.44 □

5 Self-Interpreters For Typed Calculi

We have seen the self-intepreters for untyped lambda calculus. The question remains if we could extend this to different forms of typed lambda calculus. In this section we will explore this and give the definitions for representation and self-interpreters for typed calculus.

With typed lambda calculus we are dealing with types, therefore, when we adopt our definition of representation to the typed variant we will have to deal with the type of the representation. Here we notice that typed representations already have two very different cases. When we would consider a representation similar to the Mogensen representation for untyped lambda calculus an important question is if could be typed. To have a typed Mogensen representation has to be a function of type of the term that is being represented. If we otherwise would consider something like the Barendregt representation we are only dealing with numbers and therefore only dealing with a single type for all our representations.

Definition 5.1.
Let $\Box$ be a function from types to types, then an *encoding function* is a function $\ulcorner - : \tau \urcorner : \Lambda^{\tau} \to \Lambda^{\Box\tau}$, such that for all $M, N \in \Lambda$ we have

- Injectivity up to alpha equivalence, i.e. $\ulcorner M : \tau \urcorner =_{\alpha} \ulcorner N : \sigma \urcorner \Rightarrow M =_{\alpha} N$
- Normality, i.e. $\ulcorner M : \tau \urcorner$ is in normal form

The set $\{\ulcorner M : \tau \urcorner \mid M \in \Lambda^{\tau}$ and forall types $\tau\}$ is called a *representation* or an *encoding*.
When there is a type σ such that $\Box\tau = \sigma$ for all τ, then we the representation has a constant type.

With this definition we can both capture the Barendregt type encoding functions and the Mogensen type encoding function. However it is not clear if calculi with a simple type system, like simple typed calculus, System T and PCF, can support an open-term representations. For example we can prove that the Mogensen encoding is untypable in these calculi.

Theorem 5.2. *Mogensen encoding function is untypable in simple lambda calculus*

Proof. Let x be some variable of type τ, then we have for $\ulcorner x : \tau \urcorner_m$ the following type judgment

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\{e:\sigma\} \vdash e : (\rho_1 \to \rho_2 \to \rho_3 \to \rho_1) \to \tau \to \sigma \to \pi \qquad \dots}{\{e:\sigma\} \vdash e\ (\lambda x_1 : \rho_1 \ .\ \lambda x_2 : \rho_2 \ .\ \lambda x_3 : \rho_3 \ .\ x_1) : \tau \to \sigma \to \pi}}{\{e:\sigma\} \vdash e\ (\lambda x_1 : \rho_1 \ .\ \lambda x_2 : \rho_2 \ .\ \lambda x_3 : \rho_3 \ .\ x_1)\ x : \sigma \to \pi \qquad \{e:\sigma\} \vdash e : \sigma}}{\{e:\sigma\} \vdash e\ (\lambda x_1 : \rho_1 \ .\ \lambda x_2 : \rho_2 \ .\ \lambda x_3 : \rho_3 \ .\ x_1)\ x\ e : \pi}}{\emptyset \vdash \lambda e : \sigma \ .\ e\ (\lambda x_1 : \rho_1 \ .\ \lambda x_2 : \rho_2 \ .\ \lambda x_3 : \rho_3 \ .\ x_1)\ x\ e : \sigma \to \pi}}{\emptyset \vdash \ulcorner x : \tau \urcorner_m : \sigma \to \tau}$$

So we have the constraint that $\sigma = (\rho_1 \to \rho_2 \to \rho_3 \to \rho_1) \to \tau \to \sigma \to \pi$, which is impossible. $\Box$

This is not the only problem we have with calculi with simple types. The second hurdle we have to overcome is when we try to turn the self-interpreters definitions into typed definitions. Let's say we want a recognizer $\mathsf{unquote}$, then this is a typed term with a simple type, so let's say $\mathsf{unquote} : \tau \to \sigma$. Now the recognizer we have defined cannot be as with the untyped version, since if M_1 and M_2 do not have the same type, then $\mathsf{unquote}\ulcorner M_1 \urcorner$ and $\mathsf{unquote}\ \ulcorner M_2 \urcorner$ should also not have the same type, however here they are both of type σ. This problem however is possible to solve when instead of simple types we look at more complex type systems, like extending to polymorphic types. To keep typed calculi with a simple type system in the conversation we will change the definition.

Definition 5.3. A *typed self-recognizer* is for all types τ a pair $(\ulcorner - : \tau \urcorner, \mathsf{unquote}^\tau)$, where $\ulcorner - : \tau \urcorner : \Lambda^\tau \to \Lambda^{\Box\tau}$ is a typed encoding function and $\mathsf{unquote}_\tau \in \Lambda^{\Box\tau \to \tau}$ an family of terms, such that the following holds for all $M \in \Lambda^\tau$:

$$\mathsf{unquote}^\tau\ \ulcorner M : \tau \urcorner \twoheadrightarrow_\beta M$$

We will again call the self-recognizer weak if we have β-equivalence relation instead of a reduction.
When there exists a term $\mathsf{unquote}$

$$\mathsf{unquote}\ \tau =_\beta \mathsf{unquote}^\tau$$

we say that the *typed self-recognizer* is a *polymorphic self-recognizer.*

Definition 5.4. A *typed self-evaluator* is for all types τ a pair $(\ulcorner - : \tau \urcorner, E^\tau)$, where $\ulcorner - : \tau \urcorner : \Lambda^\tau \to \Lambda^{\Box\tau}$ is a typed encoding function and $E\tau \in \Lambda^{\Box\tau \to \tau}$ an family of terms, such that the following holds for all $M \in \Lambda^\tau$:

$$E^\tau\ \ulcorner M : \tau \urcorner =_\beta \ulcorner \mathbf{nf}(M) : \tau \urcorner$$

We will again call the self-recognizer weak if we have β-equivalence relation instead of a reduction.
When there exists a term E

$$E\ \tau =_\beta E^\tau$$

we say that the *typed self-recognizer* is a *polymorphic self-recognizer.*

Now that we have definitions for self-interpreters and representations for typed lambda calculus, it is interesting if it is possible to adapt the lemmas we have seen for untyped lambda calculus to the typed version. First thing we find is that the encoding function is again not definable for the typed version.

Lemma 5.5. *For any calculus extending $\lambda_\to$ we have that for any type τ, the encoding function $\ulcorner - : \tau \urcorner$ is not lambda definable, i.e. there does not exists a $C_\tau \in \Lambda^{\tau \to \Box\tau}$ such that $C_\tau\ M =_\beta \ulcorner M : \tau \urcorner$ for all $M \in \Lambda^\tau$.*

Proof. Assume, towards contradiction, that such a C_τ does exists. Let $M \in \Lambda^\tau$. Then again we have:

$$\ulcorner (\lambda x : \tau\ .\ x) M : \tau \urcorner =_\beta C_\tau((\lambda x : \tau\ .\ x)\ M) =_\beta C_\tau\ M =_\beta \ulcorner M : \tau \urcorner \qquad ↯$$

□

But not all theorems have a typed version. One of the theorems that does not have a counterpart is the second fixed-point theorem. By enforcing types we get a problem when looking for a fixed-point M for some term F. First thing we notice is that for the second fixed-point equation $M =_\beta F \ulcorner M \urcorner$ to be valid we need to assume that F has type $\Box\tau \to \sigma$, where τ is the type of M. From this we have to conclude that F has type $\Box\tau \to \tau$. So to adapt Lemma 4.9 we have to limit it to the type $\Box\tau \to \tau$. If we try to adapt the proof of the Lemma, we get a problem with the type when using the terms dquote and app. This can be solved by assuming a fixed type for our representation, which leads us to the following lemma.

Lemma 5.6. *Let $\ulcorner - : - \urcorner$ be an encoding function with constant type ν. Let F be a term of type $\nu \to \tau$. Then there exists a M of type τ such that*

$$M =_\beta F \ulcorner M : \tau \urcorner$$

.

Proof. Let $A := \lambda x : \nu \;.\; \mathsf{app}\; n\; (\mathsf{dquote}\; n)$. then

$$\begin{array}{lll} & M & \\ = & A \ulcorner A \urcorner & \\ \to_\beta & F\ (\mathsf{app} \ulcorner A \urcorner\ (\mathsf{dquote} \ulcorner A \urcorner)) & \text{(Definition } A\text{)} \\ \twoheadrightarrow_\beta & F\ (\mathsf{app} \ulcorner A \urcorner \ulcorner\ulcorner A \urcorner\urcorner) & \text{(Definition } \mathsf{dquote}\text{)} \\ \twoheadrightarrow_\beta & F \ulcorner A \ulcorner A \urcorner\urcorner & \text{(Definition } \mathsf{app}\text{)} \\ = & F \ulcorner M \urcorner & \end{array}$$

□

Interestingly the lack of a second fixed-point theorem will be important when we look at strongly normalizing calculi, we then need a representation without a fixed-point.

One of the things we can adapt is the function definition scheme, even though it becomes a bit messier. It would be impossible to define this for a calculus using simple types, since to do this, our H has to look into the representation, find the underlying type and then somehow create terms based on this type. Therefore, Like the second fixed-point theorem, we have to assume that our representation has a constant type.

Definition 5.7. We say a representation, with a constant type ν, has a *function definition scheme* if for terms, H_1, H_2 and H_3 with types

$$\begin{aligned} H_1 &: \nu \to (\nu \to \sigma) \to \sigma \\ H_2 &: \nu \to \nu \to (\nu \to \sigma) \to \sigma \\ H_3 &: \nu \to \nu \to (\nu \to \sigma) \to \sigma \end{aligned}$$

there exists some term $H : \nu \to \sigma$ such that we have

$$\begin{aligned}
H \ulcorner x : \tau \urcorner &=_\beta H_1 \ulcorner x : \tau \urcorner H \\
H \ulcorner M\ N : \tau \urcorner &=_\beta H_2 \ulcorner M : \tau_2 \to \tau \urcorner \ulcorner N : \tau \urcorner H \\
H \ulcorner x : \tau_1 \urcorner &=_\beta H_3 \ulcorner x : \tau_1 \urcorner \ulcorner M : \tau_2 \urcorner) H
\end{aligned}$$

Similar to the self-intepreters, if we assume that we have polymorphic types, we can define a function definition scheme that is more in line with the function definition scheme we have seen for untyped lambda calculus.

$$\begin{aligned}
H\ \tau \ulcorner x : \tau \urcorner &=_\beta H_1\ \tau \ulcorner x : \tau \urcorner H \\
H\ \tau \ulcorner M\ N : \tau \urcorner &=_\beta H_2\ \tau_1\ \tau_2 \ulcorner M : \tau_2 \to \tau \urcorner \ulcorner N : \tau \urcorner H \\
H\ \tau \ulcorner \lambda x\ .\ M : \tau \urcorner &=_\beta H_3\ \tau_2\ (\lambda x\ .\ \ulcorner M : \tau_2 \urcorner) H
\end{aligned}$$

This is more in line to the untyped version, however it still wouldn't be as powerful. For example we would not be able to define a `unquote` since the output would be a static type. Later we will see a solution for this in System F_ω by Brown and Palsberg. We can also cautiously conclude that when we want representation similar to the untyped lambda calculus version, polymorphism is a must.

6 Self-interpreters For Normalizing Calculi

With self-interpreters for strongly normalizing calculi we have a problem. It is common knowledge that it is impossible. This common knowledge is called the Normalization Barrier Conjecture. It is a popular conjecture as is shown by Brown and Palsberg [10]. They quote Turner: "For any language in which all programs terminate, there are always terminating programs which cannot be written in it - among these are the interpreter for the language itself" [38] and they quote Stuart: "Total programming languages are still very powerful and capable of expressing many useful computations, but one thing they can't do is interpret themselves" [36]. Yet Brown and Palsberg also claim that they have broken the normalization barrier [10]. So what is going on?

The conjecture itself is a derivative from a theorem from computability theory that states that a total universal function for the computable functions is impossible. This theorem is conjectured to imply that strongly normalizing lambda calculi do not have self-interpreters. However nothing is as fallible as common knowledge. Brown and Palsberg claim in their 2016 paper that they broke the normalisation barrier and gave a proof why the conjecture itself is false[10]. However the claim that they broke this barrier is notably in its absence in their 2017 paper[11], which can be attributed to the confusion of the two different self-interpreters we have seen in Section 3. Andrej Bauer explains the problem differently and claims that the disconnect between the conjecture and the Brown-Palsberg paper comes from the definition for representations[9]. Currently the Wikipedia page, or more accurately L. Parreaux, cites that this problem is a result of the conjecture being about self-evaluators and not self-reducers[27].

So now we have a problem of hidden assumptions and inconsistent definitions. Therefore this section will be dealing with clearing up some of the fog of the Normalization Barrier Conjecture. First we will look at a bit of computability theory to know where the conjecture comes from and why this does not generalise according to the arguments presented in the 2016 Brown-Palsberg paper. Secondly we will look at Andrej Bauer's argument for why we do have a form of the Normalization Barrier, given a stricter definition of representations. Thirdly we will look at Andrej Bauer's insight for the complexity required for the type of the representation. Last we will explain the role of the self-evaluator in all this.

6.1 Computability Theory and Breaking The Self-Recognizer Normalization Barrier

The first thing we need to know is what the Normalization Barrier Conjecture is and where it comes from. In computability theory there is a theorem which we can informally understand as follows; if we write an interpreter for the total, computable functions in $\mathbb{N} \to \mathbb{N}$, then that interpreter must go into an infinite loop on some inputs.

The theorem here presented will be based on the one given by Brown and Palsberg [10]. Like a lot of famous theorems in computability, the theorem is proven using a diagonal argument.

But before we can present the theorem we need some definitions.

Definition 6.1. Let $\ulcorner - \urcorner$ be an injective function that maps each total, computable function in $\mathbb{N} \to \mathbb{N}$ to an element of $\mathbb{N}$. Then a *universal function for the total,computable functions in* $\mathbb{N} \to \mathbb{N}$ is a function u in $(\mathbb{N} \times \mathbb{N}) \rightharpoonup \mathbb{N}$ such that for every total, computable function f in $\mathbb{N} \to \mathbb{N}$ we have

$$\forall v \in \mathbb{N} \, . \, u(\ulcorner f \urcorner, v) = f(v)$$

and where $\rightharpoonup$ means that u may be partial. $\mathsf{Univ}(\mathbb{N} \to \mathbb{N})$ is the set of universal functions for the total, computable functions in $\mathbb{N} \to \mathbb{N}$.

From this definition we can prove a key property of the universal function.

Lemma 6.2. *If* $u \in \mathsf{Univ}(\mathbb{N} \to \mathbb{N})$*, then* $p_u(x) = u(x, x) + 1$ *isn't total.*

Proof.
Let u and p_u be as in the statement. Towards contradiction assume that p_u is total. Since p_u is a total, computable function, we have that $\ulcorner p_u \urcorner$ is defined and therefore we have

$$\begin{aligned} & p_u(\ulcorner p_u \urcorner) \\ = \; & u(\ulcorner p_u \urcorner, \ulcorner p_u \urcorner) + 1 \\ = \; & p_u(\ulcorner p_u \urcorner) + 1 \end{aligned}$$

So we have reached a contradiction, therefore p_u isn't total. □

And from this the theorem follows.

Theorem 6.3. *If* $u \in \mathsf{Univ}(N \to N)$*, then* u *isn't total.*

Proof.
Let u be as in the statement and assume towards contradiction that u is total. Let $p_u = u(x, x) + 1$. Since u is total, we have p_u is total, however this contradicts Lemma 6.2 and therefore we have that u isn't total. □

First note that, while we can consider the universal function a self-interpreter, it does not fit seamless in our definition. This is since we consider a representation an encoding function over both functions and values, here we only get a encoding over functions.

However we can safely put it in the self-recognizer camp, since its behaviour is equivalent to a self-recognizer. To see this we could try to define the universal function on constant, total function in lambda calculus. Let's say univ is the universal function and $\ulcorner - \urcorner$ is a encoding function over terms, then we require the following:

$$\mathsf{univ}\ \ulcorner M \urcorner\ x =_\beta M\ x \qquad\qquad \forall M \in \Lambda^{\mathbb{N}\to\mathbb{N}}, x \in \Lambda^{\mathbb{N}}$$

A recognizer term unquote would have this property by definition.

Now we want to translate the theorem into a theorem for lambda calculus that extends simply typed lambda calculus. First we want to translate the property from Lemma 6.2. Here the proof relies on a clever construction of p_u and that $n \neq n+1$. We can translate $n \neq n+1$ as $M \neq \lambda x\ .\ M$. Therefore the translation of Lemma 6.2 looks like this:

Lemma 6.4. *Let L be a strongly normalizing calculus extending simple typed calculus. Let $(\ulcorner - \urcorner, \mathsf{unquote})$ be a self-recognizer. Let $P_u := \lambda x\ .\ \lambda y\ .\ ((\mathsf{unquote}\ x)\ x)$ If $u \in \mathsf{SelfRec}(L)$, then $(P_u, \ulcorner P_u \urcorner) \notin L$.*

Proof.
Suppose $u \in \mathsf{SelfRec}(L)$ and $(P_u\ \ulcorner P_u \urcorner) \in L$. We calculate:

$$\begin{aligned} & p_u \overline{p_u} \\ =_\beta & \lambda y\ .\ ((u\ \ulcorner P_u \urcorner)\ \overline{P_u}) \\ =_\beta & \lambda y\ .\ (p_u\ \ulcorner P_u \urcorner) \end{aligned}$$

From $(P_u\ \ulcorner P_u \urcorner) \in L$ we have that $(P_u\ \ulcorner P_u \urcorner)$ is strongly normalizing. From the Church-Rosser property of L, we have that $(P_u\ \ulcorner P_u \urcorner)$ has a unique normal form; let us call it v. Therefore we have

$$v =_\beta \lambda y\ .\ v$$

both are in a distinct normal form, which is a contradiction. Therefore we have $(P_u, \ulcorner P_u \urcorner) \notin L$ ☐

Now notice the lack of types for the abstractions in P_u. Well for some representations, P_u does not exists, for example in the typing $\square\tau := \tau$. It may also be possible for unquote and P_u to be typeable but $P_u\ \ulcorner P_u \urcorner$ not be typeable. While it seems we can generalise the theorems from computability theorem for some representation types, it definitely does not generalise to all types. We will see that the self-recognizer will be definable later on and therefore the normalization barrier will be broken.

The question remains however why so many people thought a self-interpreter would be impossible for strongly normalizing calculi. Here we have to note that computability theory is only about natural numbers, so it may imply that a self-recognizer on numbers, Barendregt style, is impossible. Andrej Brauer investigates this question and the question what the type requirements are for a self-recognizer [9]. We will go over it in the following sections.

6.2 Bauer's Normalization Barrier

While in the last section we have seen that we have no normalization barrier in general, Bauer proves that we have such a barrier for self-interpreters for System T [9]. In this section we generalise this proof to show that weak normalizing typed lambda calculus extending simple typed lambda calculus does not have self-recognizer with a constant type.

The first thing we find is that when we have a constant self-recognizer at some type, let's say τ, then we have a fixed-point operator at the function type, i.e. $\tau \to \tau$.

Lemma 6.5. *If a lambda calculus extending simply lambda calculus has a self-recognizer* ($\ulcorner - : \tau \urcorner, \mathsf{unquote}^\tau$) *with a constant type, i.e.* $\Box\sigma = \nu$ *for all types* σ*, then every closed term* M *of type* $\tau \to \tau$ *has a fixed-point.*

Proof.
Let M and ν be as in the statement and let $N : \nu \to \tau$, with
$N = \lambda x : \nu \,.\, M\ (\mathsf{unquote}^{\nu\to\tau}\ x\ x)$. We notice that N is a closed term, since M and $\mathsf{unquote}$ are closed. By definition of our encoding function, we have $\ulcorner N \urcorner : \nu$. Therefore $\mathsf{unquote}^{\nu\to\tau}\ \ulcorner N \urcorner\ \ulcorner N \urcorner \in \Lambda_c^\tau$. And we have:

$$\begin{array}{rl} & A \\ = & \mathsf{unquote}^{\nu\to\tau}\ \ulcorner N \urcorner\ \ulcorner N \urcorner \\ \to_\beta & N\ \ulcorner N \urcorner \\ \twoheadrightarrow_\beta & M\ (\mathsf{unquote}^{\nu\to\tau}\ \ulcorner N \urcorner\ \ulcorner N \urcorner) \\ = & M\ A \end{array}$$

Therefore A is a fixed-point of M. $\Box$

With this lemma, we can prove that there does not exists a constant self-recognizer for any weakly normalizing typed calculi.

Theorem 6.6. *If a lambda calculus extending simply typed lambda calculus has a self-interpreter* ($\ulcorner - : \tau \urcorner, \mathsf{unquote}_\tau$) *with constant encoding function, then it is not weakly normalizing.*

Proof.
For the self-interpreter, let $\Box\sigma = \nu$ for all types σ. Let τ be some type. Now let $f : \tau \to \tau$ and $x : \tau$ be variables. Then by Theorem 6.5 there exists some term $\mathit{fix} : \tau \to \tau$ such that

$$\mathit{fix}\ f\ x =_\beta f\ (\mathit{fix}\ f\ x)$$

Now by the Church-Rosser theorem we have that there exists some M such that $\mathit{fix}\ f\ x \twoheadrightarrow_\beta M$ and $f\ (\mathit{fix}\ f\ x) \twoheadrightarrow_\beta M$. Then for any N_1, N_2 we have $f\ N_1 \twoheadrightarrow_\beta f\ N_2$, since f is an variable and therefore irreducible. So

$$M =_\alpha f\ M_2 \text{ for some } M_2 \text{ where } fix\ f\ x \twoheadrightarrow_\beta M_2$$

or a bit more understandable, we have the situation as shown in Figure 3

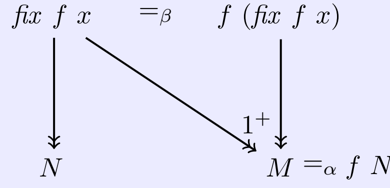


Figure 3

Now assume that we have some P such that $P =_\beta f\ P$ and there exists a i-step reduction for some $i \in \mathbb{N}$, such that $P \twoheadrightarrow^i_\beta f\ Q$, $f\ P \twoheadrightarrow_\beta f\ Q$ and $P \twoheadrightarrow_\beta Q$, i.e. the situation we can see in Figure 4.

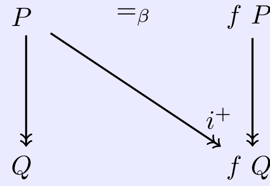


Figure 4

Then $Q =_\beta f\ Q$ and therefore by the Church-Rosser theorem there exists some R such that $Q \twoheadrightarrow_\beta R$ and $f\ Q \twoheadrightarrow_\beta R$. Now since f is a variable, it is irreducable. Therefore $R =_\alpha f\ S$ where $Q \twoheadrightarrow_\beta S$. And therefore the reduction $Q \twoheadrightarrow_\beta R =_\alpha f\ S$ has at least one step. Or in a final picture, shown in Figure 5.

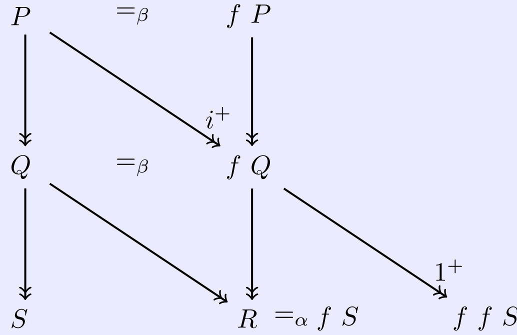


Figure 5

So therefore P has a beta reduction path of size at least $i + 1$.
So now we can conclude that for any arbitary $i \in \mathbb{N}$ there is exists a M such that $\mathit{fix}\ f\ x \twoheadrightarrow^{i^+}_\beta M$, which means that $\mathit{fix}\ f\ x$ has an inifinite beta-reduction path, and therefore it is not weakly normalizing.

□

Now with the theorem in our pocket, we can prove a version of the popular conjecture

Corollary 6.7. *(Bauer's Normalization Barrier)*
A typed lambda calculus extending simply typed lambda calculus that is strongly

normalizing does not have a self-recognizer with a constant type.

Proof. Strong normalization implies weak normalization, therefore it directly follows from Theorem 6.6 □

However we will see that this normalization barrier does not extend to self-recognizers with representations that have no constant types.

6.3 Complexity of $\Box\tau$

So it is not possible for a strong-normalizing calculus extending simply typed lambda calculus to have a self-recognizer with a constant encoding function. Then the question that arise: Are there maybe more constraints to a self-recognizer for such a calculus? Andrej Bauer [9] looks at this, and found that there are restriction to the complexity of the type of the encoded terms.

However, what do we mean when we say complexity of a type? We have some intuition on this, the type $\tau \to \tau$ should be more complex than τ. A type complexity we could propose for simple typed lambda calculus could be something as follows:

Definition 6.8. For simple typed lambda calculus, extended with the types nat and bool, and the if statement and equality, we define the function *lev* from types to natural numbers as follows

$$lev(\tau) = \begin{cases} max(1 + lev(\rho), lev(\sigma)) & \tau = \rho \to \sigma \\ 0 & \tau = \text{nat} \end{cases}$$

A flaw in this definition is that a level function is very specific for the lambda calculus in question. If we extend our lambda calculus with some other type or feature, we have to change our level function. Luckily the following definition seems to capture type complexity well.

Definition 6.9. A type σ is a retract of type τ, written $\sigma \lhd \tau$ if there is $s \in \Lambda_c^{\sigma\to\tau}$, called section, and a $r \in \Lambda_c^{\tau\to\sigma}$, called a retraction, such that

$$\lambda x : \sigma \;.\; r\ (s\ x) =_{\beta\eta} \lambda x : \sigma \;.\; x$$

And with the definition of type retraction we get some useful properties.

Lemma 6.10.

- $\sigma \lhd \sigma$
- *if* $\rho \lhd \sigma$ *and* $\sigma \lhd \tau$ *then* $\rho \lhd \tau$
- *if* $\sigma \lhd \sigma'$ *and* $\tau \lhd \tau'$ *then* $\sigma' \to \tau \lhd \sigma \to \tau'$

Proof.
Case $\sigma \lhd \sigma$
$\lambda x : \sigma \;.\; id_\sigma\ (id_\sigma\ x) =_\beta \lambda x : \sigma \;.\; id_\sigma\ x =_\beta \lambda x : \sigma \;.\; x$

Case if $\rho \lhd \sigma$ and $\sigma \lhd \tau$ then $\rho \lhd \tau$

Let ρ, σ and τ be some type such that $\rho \lhd \sigma$ and $\sigma \lhd \tau$. Then there exists some $r_1 : \rho \to \sigma$ and $s_1 : \sigma \to \rho$ such that

$$\lambda x : \rho \ . \ r_1 \ (s_1 \ x) =_{\beta\eta} \lambda x : \sigma \ . \ x$$

and a $r_2 : \sigma \to \tau$ and $s_2 : \tau \to \sigma$ such that

$$\lambda x : \sigma \ . \ r_2 \ (s_2 \ x) =_{\beta\eta} \lambda x : \sigma \ . \ x$$

. Take $r = \lambda x : \rho \ . \ r_1 \ (r_2 \ x)$ and $s = \lambda x : \tau \ . \ s_2 \ (s_1 \ x)$, then

$$\begin{aligned}
\lambda x : \rho \ . \ r \ (s \ x) &=_\alpha \lambda x : \rho \ . \ (\lambda y : \rho \ . \ r_1 \ (r_2 \ y)) \ ((\lambda z : \tau \ . \ s_2 \ (s_1 \ z)) \ x) \\
&=_\beta \lambda x : \rho \ . \ (r_1 \ (r_2 \ (s_2 \ (s_1 \ x))) \\
&=_\beta \lambda x : \rho \ . \ (r_1 \ ((\lambda y : r_2 \ (s_2 \ y)) \ (s_1 \ x))) \\
&=_{\beta\eta} \lambda x : \rho \ . \ (r_1 \ (s_1 \ x)) \\
&=_{\beta\eta} \lambda x : \rho \ . \ x
\end{aligned}$$

Case if $\sigma \lhd \sigma'$ and $\tau \lhd \tau'$ then $\sigma' \to \tau \lhd \sigma \to \tau'$
Let σ, σ', τ and τ' be some type such that $\sigma \lhd \sigma'$ and $\tau \lhd \tau'$. Then there exists some $r_1 : \sigma \to \sigma'$ and $s_1 : \sigma' \to \sigma$ such that

$$\lambda x : \rho \ . \ r_1 \ (s_1 \ x) =_{\beta\eta} \lambda x : \sigma \ . \ x$$

and a $r_2 : \tau \to \tau'$ and $s_2 : \tau' \to \tau$ such that

$$\lambda x : \tau \ . \ r_2 \ (s_2 \ x) =_{\beta\eta} \lambda x : \sigma \ . \ x$$

Let

$$r = \lambda f : \sigma \to \tau' \ . \ \lambda x : \sigma' \ . \ r_2 \ (f \ (r_1 \ x))$$

and

$$s = \lambda f : \sigma' \to \tau \ . \ \lambda x : \sigma \ . \ s_2 \ (f \ (s_1 \ x))$$

then

$$\begin{aligned}
\lambda f : \sigma' \to \tau \ . \ r \ (s \ f) &=_\alpha \lambda f : \sigma' \to \tau \ . \ (\lambda g : \sigma \to \tau' \ . \ \lambda x : \sigma' \ . \ r_2 \ (g \ (r_1 \ x))) \\
&\quad ((\lambda h : \sigma' \to \tau \ . \ \lambda y : \sigma \ . \ s_2 \ (h \ (s_1 \ y))) \ f) \\
&=_\beta \lambda f : \sigma' \to \tau \ . \ (\lambda x : \sigma' \ . \ r_2 \ ((\lambda y : \sigma \ . \ s_2 \ (f \ (s_1 \ y))) \ (r_1 \ y)))) \\
&=_\beta \lambda f : \sigma' \to \tau \ . \ (\lambda x : \sigma' \ . \ r_2 \ (s_2 \ (f \ (s_1 \ (r_1 \ y))))) \\
&=_{\beta\eta} \lambda f : \sigma' \to \tau \ . \ (\lambda x : \sigma' \ . \ f \ y) \\
&=_\eta \lambda f : \sigma' \to \tau \ . \ f
\end{aligned}$$

□

This seems to capture type complexity. For example Bauer [9] shows that his level function defined for System T implies retraction of two types and it can be shown this is true for our defined level function for simply typed lambda calculus.

Now we can use this more general notion for type complexity to define a more general level function.

Definition 6.11. A complexity function is a function lv from types to the natural

numbers such that for all σ, τ, if $lv(\sigma) \leq lv(\tau)$ then $\sigma \triangleleft \tau$.

Small problem with this is that it leaves us with an unspecified complexity function. This can be difficult to work with. However we conjecture that we can always transform a level function into something as follows:

Conjecture 6.12. *if a lambda calculus has a complexity function, then it has a complexity function lv' such that $lv'(\rho \to \sigma) = \max(1 + lv'(\rho), lv'(\sigma))$.*

So if we use a complexity function, we can just assume it has the rule

$$lv(\sigma \to \tau) = \max(1 + lv(\sigma), lv(\tau))$$

without loss of generality.

Now we can show that if the complexity of $\Box\tau$ isn't equal or higher to τ, for any complexity function, then we have fixed-points for some functions. This proof again closely follows a proof given by Bauer [9]. But first we have to introduce some notation. For any type τ, we define $(\tau)^i$ for $i \in \mathbb{N}$ as follows:

$$(\tau)^0 = \tau \qquad\qquad (\tau)^{i+1} = (\tau)^i \to (\tau)^i$$

Theorem 6.13. *Let lv be a complexity function for some typed calculus. For any type τ, if a weak Brown-Palsberg self-interpreter satisfies $lv(\Box\tau) < lv(\tau)$, then every $f \in \Lambda_c^{(\tau)^i}$, for $i \geq 1$, has a fixed-point with respect of $\beta\eta-$equivalence.*

Proof. Let $i \in \mathbb{N}$. Then we notice that

$$lv(\Box\tau \to (\tau)^i) = max(lv(\Box\tau) + 1, lv((\tau)^i)) = lv((\tau)^i)$$

Therefore we have $lv(\Box\tau \to (\tau)^i) \leq lv(\tau^i)$. Now by definition of the complexity function we have a $\Box\tau \to (\tau)^i \triangleleft \tau^i$, so therefore we have some

$$s : (\Box\tau \to (\tau)^i) \to (\tau)^i \qquad \text{and} \qquad r :\to (\tau)^i \to \Box\tau \to (\tau)^i$$

such that

$$\lambda x : \Box\tau \to (\tau)^i \ . \ r\ (s\ x) =_{\beta\eta} \lambda x : \Box\tau \to (\tau)^i \ . \ x$$

Now we let $f \in \Lambda_c^{(\tau)^i}$, $g = \lambda x : \Box\tau \ . \ f\ (r\ (\mathsf{unquote}_\tau\ x)\ x)$, and $n = \ulcorner s\ g \urcorner$. Then we have

$$\begin{aligned} r\ (\mathsf{unquote}\ n)\ n &=_\beta r\ (s\ g)\ n \\ &=_{\beta\eta} g\ n \\ &=_\beta f\ (r\ (\mathsf{unquote}\ n)\ n) \end{aligned}$$

So f has a fixed-point under $\beta\eta$-equivalence.

□

With this theorem, we can show that any $\Box\tau$ needs to be at least as complex as τ.

Corollary 6.14. *Let lv be some complexity function for a weakly normalizing calculus. Then any self-interpreter satisfies $lv(\Box\tau) \geq lv(\tau)$ for every type τ.*

Proof. Assume, toward contradiction, that for some τ we have $lv(\Box\tau) < lv(\tau)$. Let

$$s := \lambda n : (\tau \to \tau) \to \tau \to \tau \ . \ \lambda f : \tau \to \tau \ . \ x \to \tau \ . \ f \ (n \ f \ x)$$

Then s is of type $(\tau)^3$ and therefore by theorem 6.13 s has a fixed-point. However, as seen in the proof of theorem 6.6, this leads to a contradiction. $\Box$

6.4 Self-Evaluator

We have said a lot about the self-recognizer in this section and not a lot about the other self-interpreter, the self-evaluator. Would it even be possible to define a self-evaluator or is there a self-evaluator barrier? We have seen that the evaluator, in Section 4.0.3, is more difficult to define than a recognizer and makes use of fixed-point combinator. Here things changes when we are talking about a strongly normalizing calculus and the representations we can make with it.

Let say we have a strongly normalizing calculus with products and with an effective representation $\ulcorner - : \tau \urcorner$, now we can define a new representation and an evaluator as follows

$$\begin{aligned} \ulcorner M : \tau \urcorner' &:= (\ulcorner M : \tau \urcorner, \ulcorner \mathbf{nf}(M) : \tau \urcorner) \\ E^{\tau} &:= \lambda x : \Box\tau \times \Box\tau \ . \ \mathsf{pair}\ (\mathsf{snd}\ x)\ (\mathsf{snd}\ x) \end{aligned}$$

Now $(\ulcorner M : \tau \urcorner', E^{\tau})$ forms a self-evaluator.

With non-normalizing calculi we would add the requirement for the representation to be effective to avoid such trivial representations. However since we are working in a strongly normalizing calculus, this representation is also effective. And since every term has a unique normal form it is also a definable representation.

It seems like there is really not a simple requirement for this trivial self-evaluator to go away. For example we could add the constraint that we need to also be able to do single-step reductions, however then we could define still an effective representation that includes all single steps. We could include the constraint that we should be able to do all possible single step reductions and switch strategy ad-hoc. This still wouldn't solve the problem since we can build a tree of all possible strategies as representations and use this. And again since we are working in a strongly normalizing calculus these representations would both be definable and effective.

Therefore either the definition of the self-evaluator is faulty or it does not imply a lot about the strength of the representation and the calculus itself.

7 System F_ω Self-Recognizers

Before we go into a System F_ω self-recognizer, we need to acknowledge that types are complicated in System F_ω. We have seen this in Section 2.3, that types in System F_ω is a form of $\lambda_\rightarrow$. They are complicated enough that we can define a encoding function on types.

Definition 7.1.
An *encoding function on constructors* in System F_ω (a.k.a λ_ω) is a function $[\![-]\!] : \mathbf{Ctor}(\lambda_\omega) \rightarrow \mathbf{Ctor}(\lambda_\omega)$, such that for all $M, N \in \Lambda$ we have

- Injectivity up to beta equivalence, i.e. $[\![\tau]\!] =_\beta [\![\sigma]\!] \Rightarrow \tau =_\beta \sigma$

The set $\{[\![\tau]\!] \mid M \in \mathbf{Ctor}(\lambda_\omega)$ and for all types $\tau\}$ is called a *representation on constructors* or an *encoding on constructors.*

Note that we only need up to β-equivalence, because of the type derivation rules as seen in Definition 2.24. With this definition the identity function is also a encoding function on constructors, which is what we want as we will see in the next section. We will also change the definition of encoding function on terms to include the encoding function on terms.

Definition 7.2.
Let $[\![-]\!]$ be an encoding function on types. Let $\Box$ be a function from encoded types to types, then an *encoding function on terms* is a function $\ulcorner - : \tau \urcorner : \Lambda^\tau \rightarrow \Lambda^{\Box[\![\tau]\!]}$, such that for all $M, N \in \Lambda$ we have

- Injectivity up to alpha equivalence, i.e. $\ulcorner M : \tau \urcorner =_\alpha \ulcorner N : \sigma \urcorner \Rightarrow M =_\alpha N$
- Normality, i.e. $\ulcorner M : \tau \urcorner$ is in normal form

The set $\{\ulcorner M : \tau \urcorner \mid M \in \Lambda^\tau$ and forall types $\tau\}$ is called a *representation* or an *encoding.*
When there is a type σ such that $\Box[\![\tau]\!] = \sigma$ for all τ, then we the representation has a constant type.

7.1 Trivial Self-Recognizer

Brown and Palsberg showed, for the first time, that a strong self-interpreter is possible for a strongly normalizing calculus and therefore broke through the normalization barrier [10]. Before they introduce their complex self-interpreter, they define a trivial self-interpreter which inspired the one that we have seen in Section 4.1.

Definition 7.3. Let the identity function be the encoding for constructors. Let for all types τ and for all terms $M \in \Lambda^\tau$ the trivial encoding function be $\underline{\ulcorner M : \tau \urcorner_{\mathbb{1}}}$,

with $\Box[\![\tau]\!] = \Box\tau := (\Pi\alpha : \alpha \to \alpha) \to \tau$, as follows:

$$\underline{\ulcorner M : \tau \urcorner_{\mathbb{1}}} = \lambda \mathsf{id} : (\Pi\alpha : * \, . \, \alpha \to \alpha) \, . \, \ulcorner M : \tau \urcorner_{\mathbb{1}}$$

$$\begin{aligned}
\ulcorner x : \tau \urcorner_{\mathbb{1}} &= x \\
\ulcorner \lambda x : \tau_1 \, . \, M : \tau_1 \to \tau_2 \urcorner_{\mathbb{1}} &= \lambda x : \tau_1 \, . \, \ulcorner M : \tau_2 \urcorner_{\mathbb{1}} \\
\ulcorner M \; N : \tau \urcorner_{\mathbb{1}} &= \mathsf{id} \; (\tau_2 \to \tau) \, \ulcorner M : \tau_2 \to \tau \urcorner_{\mathbb{1}} \, \ulcorner N : \tau_2 \urcorner_{\mathbb{1}} \\
\ulcorner \lambda\alpha : \kappa \, . \, M : \Pi\alpha : \kappa . \tau \urcorner_{\mathbb{1}} &= \lambda\alpha : \kappa \, . \, \ulcorner M : \tau \urcorner_{\mathbb{1}} \\
\ulcorner M \; \tau_2 : \tau_1[\alpha := \tau_2] \urcorner_{\mathbb{1}} &= (\mathsf{id} \; (\Pi\alpha : \kappa \, . \, \tau_1) \, \ulcorner M : (\Pi\alpha : \kappa \, . \, \tau_1) \urcorner_{\mathbb{1}}) \; \tau_2
\end{aligned}$$

We find that the defined function is an encoding function is injective, as it maps terms to itself with some added variables added. It is also in normal form, since wherever redexes can form, namely in the application cases, we block reductions by adding the id term in front. Therefore the encoding function is a polymorphic encoding function as defined in Definition 7.2.

While it is good that the encoding function follows our definition, we also need that it is definable in System F_ω terms.

Lemma 7.4. *For all $M \in \Lambda^\tau$, if $\Gamma \vdash M : \tau$, then $\Gamma' \vdash \ulcorner M : \tau \urcorner_{\mathbb{1}} : \tau$, where $\Gamma' = \Gamma \cup \{\mathsf{id} : \Pi\alpha : * \, . \, \alpha \to \alpha\}$*

Proof. The proof is by induction on the structure of terms. The only non-trivial type judgements are the application cases. These have been done in Figure 6. □

Corollary 7.5. *If $\Gamma \vdash M : \tau$, then $\Gamma \vdash \underline{\ulcorner M : \tau \urcorner_{\mathbb{1}}} : (\Pi\alpha : * \, . \, \alpha \to \alpha) \to \tau$.*

Proof. It follows from Lemma 7.4 and the following type judgement.

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash (\Pi\alpha : * \, . \, \alpha \to \alpha) : *} \qquad \Gamma, \mathsf{id} : \Pi\alpha : * \, . \, \alpha \to \alpha \vdash \ulcorner M : \tau \urcorner_{\mathbb{1}} : \tau}{\Gamma \vdash \lambda \mathsf{id} : (\Pi\alpha : * \, . \, \alpha \to \alpha) \, . \, \ulcorner M : \tau \urcorner_{\mathbb{1}} : (\Pi\alpha : * \, . \, \alpha \to \alpha) \to \tau}$$

□

Definition 7.6. We define the recognizer for the trivial self-recognizer as follows:

$$\mathsf{unquote}_{\mathbb{1}} = \lambda\alpha : * \, . \, \lambda q : (\Pi\beta : * \, . \, \beta \to \beta) \to \alpha \, . \, q \; (\lambda\beta : * \, . \, \lambda x : \beta \, . \, x)$$

Now however, we need still an important thing. That the claim we make, that $(\underline{\ulcorner - : - \urcorner_{\mathbb{1}}}, \mathsf{unquote}_{\mathbb{1}} \; \tau)$ forms a self-interpreter, is indeed true. For this we will use some extra notation, where

$$\mathbb{1} = (\lambda\beta : * \, . \, \lambda x : \beta \, . \, x)$$

which will make the proof a bit more clear. Before we proof that it indeed forms a self-interpreter, we will proof that when we substitute id into a encoded term with, you guessed it, the polymorphic identity function $\mathbb{1}$, it beta reduces to the term itself.

Figure 6: Type judgments of the application cases

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \mathsf{id} : (\Pi\alpha : * \, . \, \alpha \to \alpha) \qquad \Gamma' \vdash (\tau_2 \to \tau) : *}{\Gamma' \vdash \mathsf{id}\ (\tau_2 \to \tau) : (\tau_2 \to \tau) \to (\tau_2 \to \tau)} \qquad \Gamma' \vdash \ulcorner M : \tau_2 \to \tau \urcorner : \tau_2 \to \tau}{\Gamma' \vdash \mathsf{id}\ (\tau_2 \to \tau)\ \ulcorner M : \tau_2 \to \tau \urcorner : \tau_2 \to \tau} \qquad \Gamma' \vdash \ulcorner N : \tau_2 \urcorner : \tau_2}{\Gamma' \vdash \mathsf{id}\ (\tau_2 \to \tau)\ \ulcorner M : \tau_2 \to \tau \urcorner\ \ulcorner N : \tau_2 \urcorner : \tau}$$

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash \mathsf{id} : (\Pi\alpha : * \, . \, \alpha \to \alpha) \qquad \Gamma' \vdash (\Pi\alpha : \kappa \, . \, \tau_1) : *}{\Gamma' \vdash \mathsf{id}\ (\Pi\alpha : \kappa \, . \, \tau_1) : (\Pi\alpha : \kappa \, . \, \tau_1) \to (\Pi\alpha : \kappa \, . \, \tau_1)} \qquad \Gamma' \vdash \ulcorner M : (\Pi\alpha : \kappa \, . \, \tau_1 \urcorner_{\mathbb{1}} : (\Pi\alpha : \kappa \, . \, \tau_1)}{\Gamma' \vdash \mathsf{id}\ (\Pi\alpha : \kappa \, . \, \tau_1)\ \ulcorner M : (\Pi\alpha : \kappa \, . \, \tau_1 \urcorner_{\mathbb{1}} : (\Pi\alpha : \kappa \, . \, \tau_1)} \qquad \Gamma' \vdash \tau_2 : *}{\Gamma' \vdash (\mathsf{id}\ (\Pi\alpha : \kappa \, . \, \tau_1)\ \ulcorner M : (\Pi\alpha : \kappa \, . \, \tau_1)\urcorner_{\mathbb{1}})\ \tau_2 : \tau_1[\alpha := \tau_2]}$$

Lemma 7.7. *For all terms $M \in \Lambda^\tau$ we have $\ulcorner M : \tau \urcorner_{\mathbb{1}}[id := \mathbb{1}] \twoheadrightarrow_\beta M$.*

Proof. We prove this by induction on the structure of System F_ω.
Base case

$$\ulcorner x : \tau \urcorner[\mathsf{id} := \mathbb{1}] = x$$

Abstraction case

$$\begin{aligned} \ulcorner \lambda x : \tau_1 \,.\, M : \tau_1 \to \tau_2 \urcorner[\mathsf{id} := \mathbb{1}] &= \lambda x : \tau_1 \,.\, \ulcorner M : \tau_2 \urcorner[\mathsf{id} := \mathbb{1}] \\ &\twoheadrightarrow_\beta \lambda x : \tau_1 \,.\, M \end{aligned}$$

Application case

$$\begin{aligned} &\ulcorner M\ N : \tau \urcorner[\mathsf{id} := \mathbb{1}] \\ &= \mathbb{1}\ (\sigma \to \tau)\ \ulcorner M : \sigma \to \tau \urcorner[\mathsf{id} := \mathbb{1}]\ \ulcorner N : \sigma \urcorner[\mathsf{id} := \mathbb{1}] \\ &\to_\beta (\lambda x : \sigma \to \tau \,.\, x)\ \ulcorner M : \sigma \to \tau \urcorner[\mathsf{id} := \mathbb{1}]\ \ulcorner N : \sigma \urcorner[\mathsf{id} := \mathbb{1}] \\ &\to_\beta \ulcorner M : \sigma \to \tau \urcorner[\mathsf{id} := \mathbb{1}]\ \ulcorner N : \sigma \urcorner[\mathsf{id} := \mathbb{1}] \\ &\twoheadrightarrow_\beta M\ N \end{aligned}$$

Type abstraction case

$$\begin{aligned} &\ulcorner \lambda \alpha : \kappa \,.\, M : \Pi \alpha : \kappa . \tau \urcorner[\mathsf{id} := \mathbb{1}] \\ &= \lambda \alpha : \kappa \,.\, \ulcorner M : \tau \urcorner[\mathsf{id} := \mathbb{1}] \\ &\twoheadrightarrow_\beta \lambda \alpha : \kappa \,.\, M \end{aligned}$$

Type application case

$$\begin{aligned} &\ulcorner M\ \tau : \sigma[\alpha := \tau] \urcorner[\mathsf{id} := \mathbb{1}] \\ &= (\mathbb{1}\ \sigma\ \ulcorner M : \sigma \urcorner[\mathsf{id} := \mathbb{1}])\ \tau \\ &\to_\beta ((\lambda x : \sigma \,.\, x)\ \ulcorner M : \sigma \urcorner[\mathsf{id} := \mathbb{1}])\ \tau \\ &\twoheadrightarrow_\beta ((\lambda x : \sigma \,.\, x)\ M)\ \tau \\ &\to_\beta M\ \tau \end{aligned}$$

□

Corollary 7.8. *The pair $(\underline{\ulcorner - : - \urcorner_{\mathbb{1}}}, \mathsf{unquote}_{\mathbb{1}}\ \tau)$ forms a self-interpreter.*

Proof.

$$\begin{aligned} &\mathsf{unquote}_{\mathbb{1}}\ \tau\ \underline{\ulcorner M : \tau \urcorner_{\mathbb{1}}} \\ &\to_\beta (\lambda q : (\Pi \beta : * \,.\, \beta \to \beta) \to \tau \,.\, q\ \mathbb{1})\ \underline{\ulcorner M : \tau \urcorner_{\mathbb{1}}} \\ &\to_\beta \underline{\ulcorner M : \tau \urcorner_{\mathbb{1}}}\ \mathbb{1} \\ &\to_\beta \ulcorner M : \tau \urcorner_{\mathbb{1}}[\mathsf{id} := \mathbb{1}] \\ &\twoheadrightarrow_\beta M \end{aligned}$$

7.2 Brown-Palsberg Self-Recognizer

7.2.1 Representing Types

For their encoding function on terms, Brown and Palsberg present a novel encoding function for types which has been designed to support three properties.
First of all they require it to code all constructors, not only types. Secondly it should preserve beta-equivalence between types and finally it should be expressive enough to for the benchmarks they defined.

Now let us assume we have some encoding function which represent terms, then the question is what type do we require the encoded term to be. The first thing to note is that a term is always a type, which is of a single kind. That, together with the beta-equivalence requirement, means we can only code the type abstraction and type application as themselves.

Now recall our encoding function for the trivial self-recognizer. There the type was for a encoded term of type τ is $(\Pi\alpha : \alpha \to \alpha) \to \tau$. The problem with this typing is that it can only accept the polymorphic identity function [39], and then can only output a term of type τ. However we want to output a type dependent on the type of the term, so that we still have the self-recognizer, but can also result in for example a natural number or boolean.
The way this can be done is assuming a constructor $F : * \to *$ and let the encoded term depend on this F. Then given F to be the identity, we want the encoded term to result in the type itself. Or if we enter the constant $F = \lambda\alpha : * \,.\, \mathsf{Nat}$, we want the resulting type to be a natural.

Now bringing this together we can define a pre-encoding function on types.

Definition 7.9. Let F be a free variable, with $F : * \to *$. We define the pre-encoding function for constructors as follows:

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![\tau_1 \to \tau_2]\!] &= F\ [\![\tau_1]\!] \to F\ [\![\tau_2]\!] \\ [\![\Pi\alpha : \kappa\ .\ \tau]\!] &= \Pi\alpha : \kappa\ .\ F\ [\![\tau]\!] \\ [\![\lambda\alpha : \kappa\ .\ C]\!] &= \lambda\alpha : \kappa\ .\ [\![\tau]\!] \\ [\![C_1\ C_2]\!] &= [\![C_1]\!]\ [\![C_2]\!] \end{aligned}$$

And then we define the encoding function to be the pre-encoding function where we abstract from F.

Definition 7.10.

$$\underline{[\![C]\!]} := \lambda F : * \to * \,.\, [\![C]\!]$$

By definition it follows that $\underline{[\![-]\!]}$ is a encoding function on constructors as defined in Definition 7.1.

7.2.1.1 Properties & Proofs

Now that we have encoding for constructors, we still want to check the requirements, properties and most of important of all, check if it is in fact an encoding function as

we have defined. Note that we already fulfilled a requirement, namely that the function works not only on types, but on all constructors. However the requirement that it preserves beta-equivalence isn't as obvious.

We will start off with checking if the (pre-)encoding function is well-formed. That is, given a constructor we want to know if the (pre -)encoded constructor is a constructor in System F_ω.

Lemma 7.11. *If* $\Gamma \vdash C : \kappa$*, then* $\Gamma, F : * \to * \vdash [\![C]\!] : \kappa$ *and* $\Gamma \vdash \underline{[\![C]\!]} : (* \to *) \to \kappa$*.*

Proof.
First let us define $\Gamma_2 := \Gamma \cup \{F : * \to *\}$.
Induction Hypothesis:

$$C_2 : \kappa_2 \text{ is a subconstructor of } C \implies (\Gamma \vdash C_2 : \kappa_2 \implies \Gamma_2 \vdash [\![C_2]\!] : \kappa_2)$$

Base case $C = \alpha$
Trivial, since $[\![\alpha]\!] = \alpha$.

Case $C = \tau_1 \to \tau_2$
By induction hypothesis we have $\Gamma_2 \vdash [\![\tau_1]\!] : *$ and $\Gamma_2 \vdash [\![\tau_1]\!] : *$. Then by $F : * \to * \in \Gamma_2$ we have $\Gamma_2 \vdash F\ [\![\tau_1]\!] : *$ and $\Gamma_2 \vdash F\ [\![\tau_2]\!] : *$ and therefore $\Gamma_2 \vdash [\![\tau_1 \to \tau_2]\!] : *$

case $C = \lambda\alpha : \kappa' \ .\ C_2$ (and $C = \Pi\alpha : \kappa' \ .\ \tau$)
From assumption $\Gamma \vdash C$ we have $\Gamma, a : \kappa' \vdash C_2 : \kappa_2$. Then by induction hypothesis we have $\Gamma_2, a : \kappa' \vdash [\![C_2]\!] : \kappa_2$, and therefore we have $\Gamma_2 \vdash [\![\lambda\alpha : \kappa_2 \ .\ C_2]\!] : \kappa$. $\Pi\alpha : \kappa' \ .\ \tau$ goes the same, except for the extra F application step, which follows from $F : * \to * \in \Gamma_2$.

case $C = C_1\ C_2 : \kappa$
By induction hypothesis we have $\Gamma_2 \vdash [\![C_1]\!] : \kappa_2 \to \kappa$ and $\Gamma_2 \vdash [\![C_2]\!] : \kappa_2$, so therefore $\Gamma_2 \vdash [\![C_1\ C_2]\!] : \kappa$. ☐

We have another property that will be important for showing that the encoding function preserves beta and alpha equivalence, namely a substitution lemma. The pre-encoding function has the nice property that a pre-encoded constructor with a substitution is the same as substituting the pre-encoded substitution in the pre-encoded constructor.

Lemma 7.12. *Let* C_1*,* C_2 *be constructors, then* $[\![C_1]\!][\alpha := [\![C_2]\!]] = [\![C_1[\alpha := C_2]]\!]$

Proof. **Base cases $C_1 = \alpha$ and $C_1 \neq \alpha$.**

$$[\![\alpha[\alpha := C_2]]\!] = [\![C_2]\!] = \alpha[\alpha := [\![C_2]\!]] = [\![\alpha]\!][\alpha := [\![C_2]\!]]$$

$$[\![\beta[\alpha := C_2]]\!] = [\![\beta]\!] = \beta = \beta[\alpha := [\![C_2]\!]] = [\![\beta]\!][\alpha := [\![C_2]\!]]$$

Case $C_1 = \tau_1 \to \tau_2$**.**

$$\begin{aligned}
[\![\tau_1 \to \tau_2[\alpha := C_2]]\!] &= [\![\tau_1[\alpha := C_2]]\!] \to [\![\tau_2[\alpha := C_2]]\!] \\
&= [\![\tau_1]\!][\alpha := C_2] \to [\![\tau_2]\!][\alpha := C_2] && \text{by IH} \\
&= ([\![\tau_1]\!] \to [\![\tau_2]\!])[\alpha := C_2] \\
&= [\![\tau_1 \to \tau_2]\!][\alpha := C_2]
\end{aligned}$$

Case $C_1 = \Pi\beta : * \,.\, \tau$**.**

$$\begin{aligned}
[\![\Pi\beta : * \,.\, \tau[\alpha := C_2]]\!] &= \Pi\beta : * \,.\, [\![\tau[\alpha := C_2]]\!] \\
&= \Pi\beta : * \,.\, [\![\tau]\!][\alpha := [\![C_2]\!]] && \text{by IH} \\
&= [\![\Pi\beta : * \,.\, \tau]\!][\alpha := [\![C_2]\!]]
\end{aligned}$$

Case $C_1 = D_1 \ D_2$

$$\begin{aligned}
[\![D_1 \ D_2[\alpha := C_2]]\!] &= [\![(D_1[\alpha := C_2]) \ (D_2[\alpha := C_2])]\!] \\
&= [\![D_1[\alpha := C_2]]\!] \ [\![D_2[\alpha := C_2]]\!] \\
&= [\![D_1]\!][\alpha := [\![C_2]\!]] \ [\![D_2]\!][\alpha := [\![C_2]\!]] && \text{by IH} \\
&= ([\![D_1]\!] \ [\![D_2]\!])[\alpha := [\![C_2]\!]] \\
&= [\![D_1 \ D_2]\!][\alpha := [\![C_2]\!]]
\end{aligned}$$

Case $C_1 = \lambda\beta : \kappa \,.\, D$

$$\begin{aligned}
[\![\lambda\beta : \kappa \,.\, C_2[\alpha := C_2]]\!] &= \lambda\beta : \kappa \,.\, [\![C_2[\alpha := C_2]]\!] \\
&= \lambda\beta : \kappa \,.\, [\![C_2]\!][\alpha := [\![C_2]\!]] && \text{by IH} \\
&= [\![\lambda\beta : \kappa \,.\, C_2]\!][\alpha := [\![C_2]\!]]
\end{aligned}$$

□

Now that we have that out of the way, we can show that the pre-coding function preserves alpha equality.

Lemma 7.13. *Let C_1 and C_2 be constructors, then we have*

$$C_1 =_\alpha C_2 \iff [\![C_1]\!] =_\alpha [\![C_2]\!]$$

Proof.
Base case $C_1 = \alpha$

$$\begin{aligned}
C_1 =_\alpha C_2 \text{ and } C_1 = \alpha &\iff C_1 = \alpha \text{ and } C_2 = \alpha \\
&\iff [\![C_1]\!] =_\alpha [\![C_2]\!] \text{ and } C_1 = \alpha
\end{aligned}$$

Case $C_1 = \Pi\alpha : \kappa \,.\, \tau$

$$\begin{aligned}
C_1 =_\alpha C_2 &\iff C_2 = \sigma_1 \to \sigma_2 \wedge \tau_1 =_\alpha \sigma_1 \wedge \tau_2 =_\alpha \sigma_2 \\
&\iff C_2 = \sigma_1 \to \sigma_2 \wedge [\![\tau_1]\!] =_\alpha [\![\sigma_1]\!] \wedge [\![\tau_2]\!] =_\alpha [\![\sigma_2]\!] && \text{by IH} \\
&\iff [\![C_1]\!] =_\alpha [\![C_2]\!]
\end{aligned}$$

The cases $C_1 = \Pi\alpha : \kappa \;.\; \tau$ and $C_2 = D_1\ D_2$ are similar to this one.

Case $C_1 = \lambda\alpha : \kappa \;.\; \tau$

$$\begin{aligned}
C_1 =_\alpha C_2 &\iff C_2 = \lambda\beta : \kappa \;.\; \sigma \wedge \tau =_\alpha \sigma[\beta := \alpha] \\
&\iff C_2 = \lambda\beta : \kappa \;.\; \sigma \wedge [\![\tau]\!] =_\alpha [\![\sigma[\beta := \alpha]]\!] && \text{by IH} \\
&\iff C_2 = \lambda\beta : \kappa \;.\; \sigma \wedge [\![\tau]\!] =_\alpha [\![\sigma]\!][\beta := \alpha] && \text{by Lemma 7.12} \\
&\iff [\![C_1]\!] =_\alpha [\![C_2]\!]
\end{aligned}$$

□

And from this, preservation of alpha equivalence from the encoding function follows trivially.

Corollary 7.14. *Let C_1 and C_2 be constructors, than $C_1 =_\alpha C_2$ if and only if $\underline{[\![C_1]\!]} =_\alpha \underline{[\![C_2]\!]}$.*

While alpha-equivalence is a nice property to have, this does not fulfill the beta-equivalence property required by Brown and Palsberg. To proof beta-equivalence, we will prove the stronger beta-reduction equivalence, i.e. if τ reduces to some σ, then the encoded τ also reduces to the encoded σ.

Lemma 7.15. *Let C_1 and C_2 be some constructors. Then we have*

$$C_1 \twoheadrightarrow_\beta C_2 \iff [\![C_1]\!] \implies [\![C_2]\!].$$

Proof.
Let C_1 and C_2 be constructors such that $C_1 \twoheadrightarrow_\beta C_2$. Then we can show $[\![C_1]\!] \twoheadrightarrow_\beta [\![C_2]\!]$ by induction on C_1.

Base case $C_1 = \alpha$
C_1 is in normal form, therefore $C_1 = C_2$, which means $[\![C_1]\!] = [\![C_2]\!]$ and therefore $[\![C_1]\!] \twoheadrightarrow_\beta [\![C_2]\!]$.

Case $C_1 = \tau_1 \to \tau_2$
By assumption we have $C_1 \twoheadrightarrow_\beta C_2$ and therefore there are types σ_1, σ_2 such that $\tau_1 \twoheadrightarrow_\beta \sigma_1$, $\tau_2 \twoheadrightarrow_\beta \sigma_2$ and $C_2 = \sigma_1 \to \sigma_2$.

$$\begin{aligned}
[\![C_1]\!] &= F\ [\![\tau_1]\!] \to F\ [\![\tau_2]\!] \\
&\twoheadrightarrow_\beta F\ [\![\sigma_1]\!] \to F\ [\![\sigma_2]\!] && \text{by IH} \\
&= [\![C_2]\!]
\end{aligned}$$

Case $C_1 = \Pi\alpha : \kappa \;.\; \tau$
By assumption we have $C_1 \twoheadrightarrow_\beta C_2$ and therefore there is a type σ such that $\tau \twoheadrightarrow_\beta \sigma$

and $C_2 = \Pi\alpha : \kappa \ . \ \sigma$. Then we have

$$\begin{aligned} [\![C_1]\!] &= \Pi\alpha : \kappa \ . \ F \ [\![\tau]\!] \\ &\twoheadrightarrow_\beta \Pi\alpha : \kappa \ . \ F \ [\![\sigma]\!] && \text{by IH} \\ &= [\![C_2]\!] \end{aligned}$$

Case $C_1 = \lambda\alpha : \kappa \ . \ D$
By assumption we have $C_1 \twoheadrightarrow_\beta C_2$ and therefore there is a constructor E such that $D \twoheadrightarrow_\beta E$ and $C_2 = \lambda\alpha : \kappa \ . \ E$. Then we have

$$\begin{aligned} [\![C_1]\!] &= \lambda\alpha : \kappa \ . \ [\![D]\!] \\ &\twoheadrightarrow_\beta \lambda\alpha : \kappa \ . \ [\![E]\!] && \text{by IH} \\ &= [\![C_2]\!] \end{aligned}$$

Case $C_1 = (\lambda\alpha : \kappa \ . \ D_1) \ D_2$
Now we have two possibilities for how C_2 looks like. The first is that the top redex did not reduce, and therefore there are constructors E_1 and E_2 where $D_1 \twoheadrightarrow_\beta E_1$, $D_2 \twoheadrightarrow_\beta E_2$ and $C_2 = (\lambda\alpha : \kappa \ . \ E_1) \ E_2$. Then

$$\begin{aligned} [\![C_1]\!] &= [\![(\lambda\alpha : \kappa \ . \ D_1)]\!] \ [\![D_2]\!] \\ &= (\lambda\alpha : \kappa \ . \ [\![D_1]\!]) \ [\![D_2]\!] \\ &\twoheadrightarrow_\beta (\lambda\alpha : \kappa \ . \ [\![E_1]\!]) \ [\![E_2]\!] && \text{by IH} \\ &= [\![(\lambda\alpha : \kappa \ . \ E_1)]\!] \ [\![E_2]\!] \\ &= [\![C_2]\!] \end{aligned}$$

Or we have that the top redex did reduce and therefore there are constructors E_1 and E_2 where $D_1 \twoheadrightarrow_\beta E_1$, $D_2 \twoheadrightarrow_\beta E_2$ and $C_2 = E_1[\alpha := E_2]$.

$$\begin{aligned} [\![C_1]\!] &= [\![(\lambda\alpha : \kappa \ . \ D_1)]\!] \ [\![D_2]\!] \\ &= (\lambda\alpha : \kappa \ . \ [\![D_1]\!]) \ [\![D_2]\!] \\ &\rightarrow_\beta [\![D_1]\!][\alpha := [\![D_2]\!]] \\ &\rightarrow_\beta [\![E_1]\!][\alpha := [\![E_2]\!]] && \text{by IH} \\ &\rightarrow_\beta [\![E_1[\alpha := E_2]]\!] && \text{by Lemma 7.12} \\ &= [\![C_2]\!] \end{aligned}$$

□

Corollary 7.16. *Let C_1 and C_2 be constructors, than we have*

$$C_1 =_\beta C_2 \implies [\![C_1]\!] =_\beta [\![C_2]\!] \iff \underline{[\![C_1]\!]} =_\beta \underline{[\![C_2]\!]}$$

7.2.2 Coding Terms Lower-Order terms

Now that we have an encoding function on types, we will look at the encoding function on terms. First we will look at with what we will call lower-order terms, i.e. the terms we also have in simple typed lambda calculus. These are variable, term abstraction and term application. Also, following in the footsteps of the paper by Brown and Palsberg, we will present the closed term version.

We again start with a pre-encoding function, where we are going to assume that the type of this pre-encoding function on terms is our pre-encoding function on types applied to the given constructor, i.e. if we have a closed term of type τ, the pre-encoded term has the type $F\ [\![\tau]\!]$.

We will start with variables. Unlike the others, we won't surround these with a free variable, since the self-recognizer will only be defined for closed terms.

$$\ulcorner x : \tau \urcorner_\omega = x$$

Next we look at the application case. We take a terms $M_1\ M_2 : \tau$, where $M_1 : \sigma \to \tau$ and $M_2 : \sigma$. Then we introduce a app term, similar to the *id* term in the trivial self-interpreter. However, unlike the trivial self-interpreter, this term is not only to block β-reduction, but is also there to make use of by binding it to different terms. Now for this be as flexible as possible, we need information. We will be only dealing with two types, namely $F\ [\![\sigma]\!]$ and $F\ [\![\tau]\!]$. Note that this is since $F\ [\![\sigma \to \tau]\!] = F\ (F\ [\![\sigma]\!] \to F\ [\![\tau]\!])$. And then we simply apply these two types and the terms, and the app term should have all the information it needs.

Finally we will look at the abstraction case. We let $M = \lambda x : \sigma\ .\ M_2$, with $M : \sigma \to \tau$ and $M_2 : \tau$. This one is very similar to the application case, except in reverse. Here we want to introduce an abs term, and again apply it to the two types we have to deal with. And then finally we apply the term and types to the abstraction.

Putting it all together we get the following pre-encoding and encoding function for lower-order terms.

Definition 7.17. We define the types Abs and App as follows:

$$\begin{aligned} \mathsf{Abs} &: \lambda F : * \to *\ .\ \Pi\alpha, \beta : *\ .\ (\alpha \to \beta) \to F\ (\alpha \to \beta) \\ \mathsf{App} &: \lambda F : * \to *\ .\ \Pi\alpha, \beta : *\ .\ F\ (\alpha \to \beta) \to \alpha \to \beta \end{aligned}$$

Let F, abs and app be free variables with $F : * \to *$, $\mathsf{abs} : \mathsf{Abs}\ F$ and $\mathsf{app} : \mathsf{App}\ F$. We define the pre-coding function for lower-order terms as follows:

$$\begin{aligned} \ulcorner x : \tau \urcorner_\omega &= x \\ \ulcorner \lambda x : \sigma\ .\ M : \sigma \to \tau \urcorner_\omega &= \mathsf{abs}\ (F\ [\![\sigma]\!])\ (F\ [\![\tau]\!])\ (\lambda x : F\ [\![\sigma]\!]\ .\ \ulcorner M : \tau \urcorner_\omega) \\ \ulcorner M\ N : \tau \urcorner_\omega &= \mathsf{app}\ (F\ [\![\sigma]\!])\ (F\ [\![\tau]\!])\ \ulcorner M : \sigma \to \tau \urcorner_\omega\ \ulcorner N : \sigma \urcorner_\omega \end{aligned}$$

$$\begin{aligned} \underline{\ulcorner M : \tau \urcorner} = \lambda F &: * \to *\ . \\ &\lambda \mathsf{abs} : \mathsf{Abs\ F}\ . \\ &\lambda \mathsf{app} : \mathsf{App\ F}\ .\ \ulcorner M : \tau \urcorner_\omega \end{aligned}$$

7.2.2.1 Properties & Proofs

Now for our first requirement. We wanted an encoded term with a type based on the encoded type of the term. To proof we have correctly defined the type of the encoding function, we will start with the proving that the pre-coding function has the type as we have previously required. We will both proof that this works for closed terms given an specific environment.

Lemma 7.18. *Let $\Gamma \vdash M : \tau$ be a lower-order term. Define Γ_2 as follows:*

$$\begin{aligned}\Gamma_2 :=& \{x : F\ [\![\sigma]\!] \mid x : \sigma \in \Gamma\} \ \cup\ \{\alpha : \kappa \mid \alpha : \kappa \in \Gamma\} \\ & \cup\ \{F : * \to *, \mathsf{app} : \mathsf{App}\ F, \mathsf{abs} : \mathsf{Abs}\ F\}\end{aligned}$$

Then $\Gamma_2 \vdash \ulcorner M : \tau \urcorner_\omega : F\ [\![\tau]\!]$.

Proof.
Note that we can conclude from the definition of Γ_2 and Lemma 7.11 that if $\Gamma \vdash C : *$ then $\Gamma_2 \vdash [\![C]\!] : *$ and $\Gamma_2 \vdash F\ [\![C]\!] : *$.

Induction hypothesis

$$N \text{ is a subterm of M} \implies (\Gamma \vdash N : \sigma \implies \Gamma_2 \vdash \ulcorner N : \sigma \urcorner_\omega : F\ [\![\sigma]\!])$$

Base case $M = x$

$$\dfrac{\dfrac{(x : \tau) \in \Gamma}{(x : F[\![\tau]\!]) \in \Gamma_2}\ \text{Def}\ \Gamma_2}{\Gamma_2 \vdash \ulcorner x : \tau \urcorner_\omega : F\ [\![\tau]\!]}$$

Case $\Gamma \vdash \lambda x : \sigma_1\ .\ N : \sigma_1 \to \sigma_2$
By $\Gamma \vdash \sigma_1 \to \sigma_2 : *$ and Lemma 7.11 we have that $\Gamma_2 \vdash F\ [\![\sigma_1 \to \sigma_2]\!] : *$. Therefore we have

$$\dfrac{\mathsf{abs} : \mathsf{Abs}\ F \in \Gamma_2 \qquad \Gamma_2 \vdash F\ [\![\sigma_1]\!] : * \qquad \Gamma_2 \vdash F\ [\![\sigma_2]\!] : *}{\Gamma_2 \vdash \mathsf{abs}\ (F\ [\![\sigma]\!])\ (F\ [\![\sigma_2]\!]) : ([\![\sigma_1]\!] \to [\![\sigma_2]\!]) \to F\ [\![\sigma_1 \to \sigma_2]\!])}\ (1)$$

By induction hypothesis and assumption of $\Gamma \vdash M : \tau$ we have:

$$\dfrac{\Gamma_2 \vdash F\ [\![\sigma_1]\!] : * \qquad \dfrac{\Gamma, (x : \sigma_1) \vdash N : \sigma_2}{\Gamma_2, (x : F\ [\![\sigma_1]\!]) \vdash \ulcorner N : \sigma_2 \urcorner_\omega : F\ [\![\sigma_2]\!]}}{\Gamma_2 \vdash (\lambda x : F\ [\![\sigma_1]\!]\ .\ \ulcorner N : \sigma_2 \urcorner_\omega) : F\ [\![\sigma_2]\!]}\ (2)$$

Taking this all together we can finish the proof for $\ulcorner \lambda x : \sigma_1\ .\ M\sigma_2 : \sigma_1 \to \sigma_2 \urcorner_\omega$ in a final prooftree.

$$\dfrac{(1) \qquad (2)}{\Gamma_2 \vdash \mathsf{abs}\ (F\ [\![\sigma]\!])\ (F\ [\![\tau]\!])\ (\lambda x : F\ [\![\sigma_1]\!]\ .\ \ulcorner N : \sigma_2 \urcorner_\omega) : F\ [\![\sigma_1 \to \sigma_2]\!]}$$

Case $N_1\ N_2 : \tau$ **and** $N_2 : \sigma_1$
By $\Gamma \vdash N_1\ N_2 : \tau$ and Lemma 7.11 we have that $\Gamma_2 \vdash F\ [\![\sigma \to \tau]\!] : *$. Therefore we have

$$\frac{\mathsf{abs} : \mathsf{Abs}\ F \in \Gamma_2 \qquad \Gamma_2 \vdash F\ [\![\sigma]\!] : * \qquad \Gamma_2 \vdash F\ [\![\tau]\!] : *}{\Gamma_2 \vdash \mathsf{abs}\ (F\ [\![\sigma]\!])\ (F\ [\![\tau]\!]) : (F\ [\![\sigma \to \tau]\!]) \to (F\ [\![\sigma]\!]) \to (F\ [\![\tau]\!])}\ (1)$$

By induction hypothesis and assumption of $\Gamma \vdash M : \tau$ we have:

$$\frac{\Gamma \vdash N_1 : \sigma \to \tau}{\Gamma_2 \vdash \ulcorner N_1 : \sigma \to \tau \urcorner_\omega : F\ [\![\sigma \to \tau]\!]}\ (2\text{a}) \qquad \frac{\Gamma \vdash N_2 : \sigma}{\Gamma_2 \vdash \ulcorner N_2 : \sigma \urcorner_\omega : F\ [\![\sigma]\!]}\ (2\text{b})$$

Taking this all together we can finish the proof for $\ulcorner N_1\ N_2 : \tau \urcorner_\omega$ in a final prooftree.

$$\frac{(1) \qquad (2a) \qquad (2b)}{\Gamma_2 \vdash \mathsf{app}\ (F\ [\![\sigma]\!])\ (F\ [\![\tau]\!])\ \ulcorner N_1 : \sigma \to \tau \urcorner_\omega\ \ulcorner N_2 : \sigma \urcorner_\omega : F\ [\![\tau]\!]}$$

□

Now note that the environment is based on the environment of the term itself and the free variables we defined in the pre-coded function. Therefore if we bind the free variables then the encoding of a closed term is itself a closed term.

Definition 7.19.

$$\mathsf{Exp} = \lambda\alpha : (* \to *) \to * \,.\, \Pi F : * \to * \,.\, \mathsf{Abs}\ F \to \mathsf{App}\ F \to (\alpha\ F)$$

Corollary 7.20. *Given some lower-order term M with $\Gamma \vdash M : \tau$ and define $\Gamma_2 := \{x : F\ [\![\sigma]\!] \mid x : \sigma \in \Gamma\} \ \cup\ \{\alpha : \kappa \mid \alpha : \kappa \in \Gamma\}$ then we have $\Gamma_2 \vdash \underline{\ulcorner M : \tau \urcorner_\omega} : \mathsf{Exp}\ \underline{[\![\tau]\!]}$*

Proof. Follows from Lemma 7.18 and the following beta-reduction:

$$\mathsf{Exp}\ \underline{[\![\tau]\!]} \twoheadrightarrow_\beta \Pi F : * \to * \,.\, \mathsf{Abs}\ F \to \mathsf{App}\ F \to F\ [\![\tau]\!]$$

□

Now the question remains if the function $\ulcorner - : - \urcorner_\omega$ is an encoding function, as we have defined in Definition 7.17. For a reminder, for this to be true the function needs to be injective and normal.

Lemma 7.21. *The function $\ulcorner - : - \urcorner_\omega$ is normal for lower-order terms, i.e. given some simple term M with $\Gamma \vdash M : \tau$ the encoded term $\underline{\ulcorner M : \tau \urcorner_\omega}$ is in normal form*

Proof.
We note that the terms abs and app are bound and therefore cannot reduce. The same is true for the constructor F. Therefore a beta reduction can only occur in the application case. In specific, if we are have a term $M = M_1\ M_2$, then $\ulcorner M_1 \urcorner_\omega$

has to be of the form $\lambda x : \tau \; . \; N$, which is impossible. □

Lemma 7.22. *The function* $\ulcorner - : - \urcorner_\omega$ *is injective for lower-order terms, i.e. given some simple term* M *and* N *with* $\Gamma_M \vdash M : \tau$ *and* $\Gamma_N \vdash N : \sigma$ *then we have*

$$\ulcorner M : \tau \urcorner_\omega =_\alpha \ulcorner N : \sigma \urcorner_\omega \implies M =_\alpha N$$

Proof.
Let M, N be as in the Lemma statement.

Base case $M = x : \tau$.
We have $\ulcorner N : \sigma \urcorner_\omega =_\alpha x : F \; [\![\tau]\!]$ So there is some variable y and type σ such that $N = y : \sigma$. Now we have $\ulcorner y : \sigma \urcorner_\omega = y : F \; [\![\sigma]\!] =_\alpha x : F \; [\![\tau]\!]$. Therefore $[\![\sigma]\!] = [\![\tau]\!]$, then by Lemma 7.12 we have $\sigma =_\alpha \tau$ and so $N =_\alpha M$.

Case $M = (\lambda x : \tau_1 \; . \; M_2) : \tau_1 \to \tau_2$
So we have $\ulcorner N : \sigma \urcorner_\omega =_\alpha \ulcorner \lambda x : \tau_1 \; . \; M_2 : \tau_1 \to \tau_2 \urcorner_\omega$, and therefore there is some y, some σ_1, σ_2 and N_2 such that $N = (\lambda y : \sigma_1 \; . \; N_2) : \sigma_1 \to \sigma_2$. Therefore we have

$$\begin{aligned} &\mathsf{abs} \; (F \; [\![\tau_1]\!]) \; (F \; [\![\tau_2]\!]) \; (\lambda x : F \; [\![\tau_1]\!] \; . \; \ulcorner M_2 : \tau_2 \urcorner_\omega) \\ &=_\alpha \mathsf{abs} \; (F \; [\![\sigma_1]\!]) \; (F \; [\![\sigma_2]\!]) \; (\lambda y : F \; [\![\sigma_1]\!] \; . \; \ulcorner N_2 : \sigma_2 \urcorner_\omega) \end{aligned}$$

Therefore by Lemma 7.12 we have $\tau_1 =_\alpha \sigma_1$ and $\tau_2 =_\alpha \sigma_2$ and by the induction hypothesis we have $M_2 =_\alpha N_2$. Therefore $M =_\alpha N$.

Case $M = M_1 \; M_2$
By $\ulcorner N : \sigma \urcorner_\omega =_\alpha \ulcorner M_1 \; M_2 : \sigma \urcorner_\omega$, which means that there exists some $N_1 : \sigma_2 \to \sigma$ and $N_2 : \sigma_2$ such that $N = N_1 \; N_2$.

$$\begin{aligned} &\mathsf{app} \; (F \; [\![\tau_2 \to \tau]\!]) \; (F \; [\![\tau_2]\!]) \; \ulcorner M_1 : \tau_2 \to \tau \urcorner_\omega \; \ulcorner M_2 : \tau_2 \urcorner_\omega \\ &=_\alpha \mathsf{app} \; (F \; [\![\sigma_2 \to \sigma]\!]) \; (F \; [\![\sigma_2]\!]) \; \ulcorner N_1 : \sigma_2 \to \sigma \urcorner_\omega \; \ulcorner N_2 : \sigma_2 \urcorner_\omega \end{aligned}$$

Therefore by Lemma 7.12 we have $\tau_2 \to \tau =_\alpha \sigma_2 \to \sigma$ and $\tau_2 =_\alpha \sigma_2$ and by the induction hypothesis we have $M_1 =_\alpha N_1$ and $M_2 =_\alpha N_2$. Therefore $M =_\alpha N$.
□

Corollary 7.23. *The function from Definition 7.17 is an encoding function for lower-order terms.*

It is good to know that we have defined the encoding functions according to our definitions, however we are still missing the most important part. What can we do with the encoding function? We will show that it at least forms a strong self-interpreter for lower-order terms. To prove this we will first define the term foldExp, which will bind the free variables of the pre-coding function.

Definition 7.24.

$$\begin{aligned}\mathsf{foldExp} : \Pi F : * \to * \,.\, \mathsf{Abs}\ F \to \mathsf{App}\ F \to \\ \Pi\alpha : (* \to *) \to * \,.\, \mathsf{Exp}\ \alpha \to F\ (\alpha\ F) \\ \mathsf{foldExp} := \lambda F : * \to * \,.\, \lambda \mathsf{abs} : \mathsf{Abs}\ F \,.\, \lambda \mathsf{app} : \mathsf{App}\ F \,. \\ \lambda\alpha : (* \to *) \to * \,.\, \lambda e : \mathsf{Exp}\ \alpha \,.\, e\ F\ \mathsf{abs}\ \mathsf{app}\end{aligned}$$

Lemma 7.25. *Let M be a closed simple term with $\emptyset \vdash M : \tau$. Now let $C : * \to *$ be a constructor and $A_1 : \mathsf{Abs}\ C$, $A_2 : \mathsf{App}\ C$ be some terms. Then we have*

$$\begin{aligned}&\mathsf{foldExp}\ C\ A_1\ A_2\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega} \\ &\quad \twoheadrightarrow_\beta \ulcorner M : \tau \urcorner_\omega[F := C, \mathsf{app} := A_1, \mathsf{abs} := A_2]\end{aligned}$$

Proof.

$$\begin{aligned}&\mathsf{foldExp}\ C\ A_1\ A_2\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega} \\ &\quad \twoheadrightarrow_\beta \underline{\ulcorner M : \tau \urcorner_\omega}\ C\ A_1\ A_2 \\ &\quad = (\lambda F : * \to * \,.\, \lambda \mathsf{abs} : \mathsf{Abs}\ F \,.\, \lambda \mathsf{app} : \mathsf{App}\ F \,.\, \ulcorner M : \tau \urcorner_\omega)\ C\ A_1\ A_2 \\ &\quad \twoheadrightarrow_\beta \ulcorner M : \tau \urcorner_\omega[\mathsf{abs} := A_1, \mathsf{app} := A_2]\end{aligned}$$

□

Now for the grand finale for encoding lower-order terms, a constructive proof that a strong self-interpreter exists for lower-order terms.

Definition 7.26.

$$\begin{aligned}\mathsf{Id} &: * \to * \\ &= \lambda\alpha : * \,.\, \alpha \\ \mathsf{unAbs} &: \mathsf{Abs}\ \mathsf{Id} \\ &= \lambda\alpha : * \,.\, \lambda\beta : * \,.\, \lambda f : \alpha \to \beta \,.\, f \\ \mathsf{unApp} &: \mathsf{App}\ \mathsf{Id} \\ &= \lambda\alpha : * \,.\, \lambda\beta : * \,.\, \lambda f : \alpha \to \beta \,.\, \lambda x : \alpha \,.\, f\ x \\ \mathsf{unquote} &: \Pi\alpha : U \,.\, \mathsf{Exp}\ \alpha \to \mathsf{Id}\ (\alpha\ \mathsf{Id}) \\ &= \mathsf{foldExp}\ \mathsf{Id}\ \mathsf{unAbs}\ \mathsf{unApp}\end{aligned}$$

Lemma 7.27. *Let M be a simple term, then for all types τ and for all simple closed terms M, the pair $(\underline{\ulcorner M : \tau \urcorner_\omega}, \mathsf{unquote}\ \underline{[\![\tau]\!]})$ forms a strong self-interpreter, i.e.*

$$\mathsf{unquote}\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta M.$$

Proof. Let the term M be as in the statement and let $S = [F := \mathsf{Id}, \mathsf{abs} := \mathsf{unAbs}, \mathsf{app} := \mathsf{unApp}]$. **Base case** $x : \tau$

$$\ulcorner x : \tau : \urcorner_\omega \ S = x : \mathsf{Id}\ \tau \twoheadrightarrow_\beta x : \tau$$

Now we take induction hypothesis as $\ulcorner M : \tau \urcorner_\omega \ S \twoheadrightarrow_\beta M$, then we have

Case $\lambda x : \sigma_1 \ . \ M : \sigma_1 \to \sigma_2$

$$\begin{aligned}
&\ulcorner \lambda x : \sigma_1 \ . \ M : \sigma_1 \to \sigma_2 \urcorner_\omega \ S \\
&\twoheadrightarrow_\beta \mathsf{unAbs}\ \sigma_1\ \sigma_2\ (\lambda x : \sigma_1 \ . \ \ulcorner M : \sigma_2 \urcorner_\omega \ S) \\
&\twoheadrightarrow_\beta \mathsf{unAbs}\ \sigma_1\ \sigma_2\ (\lambda x : \sigma_1 \ . \ M) \\
&\twoheadrightarrow_\beta \lambda x : \sigma_1 \ . \ M
\end{aligned}$$

Case $N_1 \ N_2$

$$\begin{aligned}
&\ulcorner N_1 \ N_2 : \tau \urcorner_\omega \ S \\
&\twoheadrightarrow_\beta \mathsf{unApp}\ (\sigma \to \tau)\ \sigma\ (\ulcorner N_1 : \sigma \to \tau \urcorner_\omega \ S)\ (\ulcorner N_2 : \sigma \urcorner_\omega \ S) \\
&\twoheadrightarrow_\beta \mathsf{unApp}\ (\sigma \to \tau)\ \sigma\ N_1\ N_2 \\
&\twoheadrightarrow_\beta N_1\ N_2
\end{aligned}$$

Then by Lemma 7.25 we have that the pair $(\underline{\ulcorner - : \tau \urcorner_\omega}, \mathsf{unquote}\ \underline{[\![\tau]\!]})$ form a self-interpreter for any type τ. $\square$

7.2.3 Coding Higher-Order Terms

In this section we want to finish the encoding function on terms. The remaining terms we will call the Higher-Order terms. We will handle both the abstraction and application with types separately. This is because both will require some extras, which is directly related to the fact that constructors are not terms, which will require some explanation.

7.2.3.1 Constructor Abstraction Term

First we will discuss the term that abstracts a constructor over a term, i.e. terms of the form $\Pi\alpha : \kappa \,.\, M$. So here α is being quantified over.

Now a quantifier is redundant when a term M has the type $\Pi\alpha : \kappa \,.\, \tau$ and α does not occur in τ. So for example the term $\lambda\alpha : \kappa \,.\, \overline{1}$ has a redundant quantifier. When we want to define a encoding for constructor abstraction terms similar to how we defined the encoding for abstraction terms, we will be dealing a lot with redundant quantifiers. To see this, let us assume that we have some F with $F = \lambda\alpha : * \,.\, \tau$, where τ is independent of α. Now we would like to code constructor abstraction terms as something like tabs (...) $(\Pi\alpha : \kappa \,.\, \ulcorner M : \sigma \urcorner_\omega)$ with a type of F $[\![\Pi\alpha : \kappa \,.\, \sigma]\!]$. We assumed F is constant, so therefore α is a redundant quantifier that we have to remove.

We know that the quantifier does not affect the type, now we want to prove that this also doesn't affect the beta-reduction.

Lemma 7.28. *When we have a term $\lambda\alpha : \kappa \,.\, N$ of type $\Pi\alpha : \kappa \,.\, \tau$ with $\alpha \notin \boldsymbol{FV}(\tau)$, then given some constructor $C : \kappa$ we have*

$$(\lambda\alpha : \kappa \,.\, M)\ C =_\beta M$$

Proof.
Base case $M = x$

$$x[\alpha := C] = x$$

Case $M = \lambda x : \tau_2 \,.\, N$

$$\begin{aligned}&(\lambda x : \tau_2 \,.\, (\lambda\beta : \kappa \,.\, N)\ C)[\alpha := C]\\ &=_\beta (\lambda x : \tau_2 \,.\, (\lambda\beta : \kappa \,.\, N[\alpha := C])\ C)\end{aligned}$$

Case $M = N_1\ N_2$

$$\begin{aligned}&(N_1\ N_2)[\alpha := C]\\ &= N_1[\alpha := C]\ N_2[\alpha := C]\\ &=_\beta N_1\ N_2 \text{By IH}\end{aligned}$$

Case $M = \lambda\beta : \kappa_2 \,.\, N$

$$\begin{aligned}&(\lambda\alpha : \kappa \,.\, \lambda\beta : \kappa_2 \,.\, N)\ C\\ &\to_\beta \lambda\beta : \kappa_2 \,.\, N[\alpha := C]\\ &=_\beta \lambda\beta : \kappa_2 \,.\, N && \text{by IH}\end{aligned}$$

Case $M = N\ D$

$$
\begin{aligned}
&(N\ D)[\alpha := C] \\
&=_\beta N\ D[\alpha := C] && \text{by IH} \\
&=_\alpha (\Pi\beta : \kappa\ .\ N_2)\ D[\alpha := C] \\
&=_\beta N_2[\beta := D[\alpha := C]]
\end{aligned}
$$

Then either $\alpha \notin D$ or β is redundant and then it follows from induction hypothesis.

□

From this lemma we can conclude that given a term with a redundant quantifier, we can apply any constructor without affecting the term inside. We also have that every constructor is inhabited, therefore given a kind we simply can generate a constructor of this kind. Such a constructor generator can be defined as follows:

Definition 7.29. The constructor generator $\mathcal{C}(\kappa)$.

$$
\begin{aligned}
\mathcal{C}(*) &= (\Pi\alpha : *\ .\ \alpha) \\
\mathcal{C}(\kappa_1 \to \kappa_2) &= \lambda\alpha : \kappa_1\ .\ \mathcal{C}(\kappa_2)
\end{aligned}
$$

However note that $\mathcal{C}(-)$ is not definable in System F_ω. So to still be able to strip redundant quantifiers, we attach a constructor of the right kind to every type abstraction term in the pre-coding.
To do this Brown and Palsberg define a strip term. This term will also take some additional inputs to make sure that the quantifier is redundant. The strip function is defined as follows:

Definition 7.30. Strip function

$$
\begin{aligned}
\mathcal{S}(F, \kappa, \lambda\alpha : \kappa\ .\ \tau) = {} & \lambda\beta : *\ .\ \lambda f : (\Pi\gamma : *\ .\ F\ \gamma \to \beta)\ .\ \lambda x : (\Pi\delta : \kappa\ .\ F\ [\![\tau[\alpha := \delta]]\!])\ . \\
& f\ [\![\tau[\alpha := \mathcal{C}(\kappa)]]\!]\ (x\ \mathcal{C}(\kappa))
\end{aligned}
$$

$$
\mathsf{Strip} = \lambda F : * \to *\ .\ \lambda\alpha : *\ .\ \Pi\beta : *\ .\ (\Pi\gamma : *\ .\ F\ \gamma \to \beta) \to \alpha \to \beta
$$

Lemma 7.31. *Let* $\Gamma \vdash \lambda\alpha : \kappa\ .\ N : \Pi\alpha : *\ .\ \tau$. *Then we have*

$$
\Gamma, (F : * \to *) \vdash \mathcal{S}(F, \kappa, (\lambda\alpha : \kappa\ .\ \tau)) : (\mathsf{Strip}\ F\ [\![\Pi\alpha : \kappa\ .\ \tau]\!])
$$

Proof.
First we note that $\emptyset \vdash \mathcal{C}(\kappa) : \kappa$ and $\Gamma \vdash \tau : *$ from assumption. Then we can also conclude that $\Gamma, F : * \to * \vdash [\![\tau]\!]$ by Lemma 7.11. With these in mind we can quickly check if it is well typed.

$$
\frac{\Gamma \vdash f : F\ [\![\tau[\alpha := \mathcal{C}(\kappa)]]\!] \to \beta \qquad \Gamma \vdash x : F\ [\![\tau[\alpha := \mathcal{C}(\kappa)]]\!] \qquad \Gamma \vdash [\![\tau[\alpha := \mathcal{C}(\kappa)]]\!] : \kappa}{f\ [\![\tau[\alpha := \mathcal{C}(\kappa)]]\!]\ (x\ \mathcal{C}(\kappa))}
$$

Now the question remains if the typing Strip is the correct typing for the strip

function.

$$
\begin{aligned}
&\mathsf{Strip}\ F\ [\![\Pi\alpha : \kappa\ .\ \tau]\!] \\
&\quad \twoheadrightarrow_\beta \Pi\beta : * \ .\ (\Pi\gamma\ .\ F\ \gamma \to \beta) \to [\![\Pi\alpha : \kappa\ .\ \tau]\!] \to \beta \\
&\quad =_\alpha \Pi\beta : * \ .\ (\Pi\gamma\ .\ F\ \gamma \to \beta) \to [\![\Pi\delta : \kappa\ .\ \tau[\alpha := \delta]]\!] \to \beta \\
&\quad = \Pi\beta : * \ .\ (\Pi\gamma\ .\ F\ \gamma \to \beta) \to \Pi\delta : \kappa\ .\ F\ [\![\tau[\alpha := \delta]]\!] \to \beta
\end{aligned}
$$

□

Lemma 7.32. *Let F be a constant constructor, i.e. $F = \lambda\alpha : *\ .\ \sigma$ for some closed type σ. Let $M = \lambda\alpha : \kappa\ .\ N$ of type $\Pi\alpha : \kappa\ .\ \tau$. Then we have*

$$
\begin{aligned}
&\mathcal{S}(F, \kappa, [\![\lambda\alpha : \kappa\ .\ \tau]\!])\ \sigma\ (\lambda\alpha : *\ .\ I_\sigma)\ (\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega) \\
&\quad \twoheadrightarrow_\beta \ulcorner N : \tau \urcorner_\omega
\end{aligned}
$$

Proof.

$$
\begin{aligned}
&\mathcal{S}(\lambda\alpha : *\ .\ \tau, \kappa, [\![\lambda\alpha : \kappa\ .\ \tau]\!])\ \sigma\ (\lambda\alpha : *\ .\ I_\sigma)\ (\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega) \\
&\quad = (\lambda\alpha : *\ .\ \lambda f : (\Pi : \beta : *\ .\ F\ \beta \to \alpha)\ .\ \lambda x : (\Pi\gamma : \kappa\ .\ F\ (C\ \gamma))\ . \\
&\qquad f\ ([\![\lambda\alpha : \kappa\ .\ \tau]\!]\ \mathcal{C}(\kappa))\ (x\ \mathcal{C}(\kappa)))\ \sigma\ (\lambda\alpha : *\ .\ I_\sigma)\ (\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega) \\
&\quad \twoheadrightarrow_\beta (\lambda f : (\Pi : \beta : *\ .\ \sigma \to \sigma)\ .\ \lambda x : (\Pi\gamma : \kappa\ .\ \sigma)\ .\ f\ ([\![\lambda\alpha : \kappa\ .\ \tau]\!]\ \mathcal{C}(\kappa))\ (x\ \mathcal{C}(\kappa))) \\
&\qquad (\lambda\alpha : *\ .\ I_\sigma)\ (\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega) \\
&\quad \twoheadrightarrow_\beta (\lambda\alpha : *\ .\ I_\sigma)\ ([\![\lambda\alpha : \kappa\ .\ \tau]\!]\ \mathcal{C}(\kappa))\ ((\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega)\ \mathcal{C}(\kappa)) \\
&\quad \twoheadrightarrow_\beta I_\sigma\ ((\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega)\ \mathcal{C}(\kappa)) \\
&\quad \twoheadrightarrow_\beta I_\sigma\ \ulcorner N : \tau \urcorner_\omega[\alpha := \mathcal{C}(\kappa)] \\
&\quad \twoheadrightarrow_\beta \ulcorner N : \tau \urcorner_\omega
\end{aligned}
$$

□

With the strip function defined and checked, we can define the pre-encoding function for type abstraction terms.

Definition 7.33.

$$\mathsf{TAbs} = \lambda F : * \to *\ .\ \Pi\alpha : *\ .\ \mathsf{Strip}\ F\ \alpha \to \alpha \to F\ \alpha$$

Let $F : * \to *$, $\mathsf{tabs} : \mathsf{TAbs}\ F$, then we define the pre-coding function for type abstraction terms as follows:

$$\ulcorner \lambda\alpha : \kappa\ .\ M : \Pi\alpha : \kappa . \tau \urcorner_\omega = \mathsf{tabs}\ [\![\Pi\alpha : \kappa\ .\ \tau]\!]\ \mathcal{S}(F, \kappa, [\![\lambda\alpha : \kappa\ .\ \tau]\!])\ (\lambda\alpha : \kappa\ .\ \ulcorner M : \tau \urcorner_\omega)$$

7.2.3.2 Constructor Application Term

When dealing with constructor application terms, we know it will always be in the form of $(\lambda\alpha : \kappa\ .\ N)\ \tau$. Again we are dealing with a constructor abstraction term. Now here we want to deal with the case for when the quantifier is not redundant, for example if we want to have a term that interpreters closed terms, i.e. $E\ \ulcorner\underline{(\lambda\alpha : \kappa\ .\ N)\ \tau : \sigma[\alpha := \tau]}\urcorner_\omega \twoheadrightarrow_\beta (\lambda\alpha : \kappa\ .\ N)\ \tau$. In our encoding function we make these terms by substituting the free variables in the pre-coding function. The pre-coding function is of type $F\ [\![\sigma[\alpha := \tau]]\!]$, which needs to be $\sigma[\alpha := \tau]$. To do this we have to bind F to the identity, i.e. $\lambda\alpha : * . \alpha$. Now F is not constant, and therefore also not redundant.

But we are still not there. For the encoded function, we want to give the term and the constructor to some free variable. Now note that a encoded term of type $\lambda\alpha : \kappa\ .\ N$ is of type $[\![\Pi\alpha : \kappa\ .\ \tau]\!] = \Pi\alpha : \kappa\ .\ [\![\tau]\!]$. Since, by Lemma 7.12 we have $[\![\sigma[\alpha := \tau]]\!] = [\![\sigma]\!][\alpha := [\![\tau]\!]]$, we can just apply the encoded type to the encoded term and get a term of the right type. For this we can use, what Brown and Palsberg call, an initiation function, which looks as follows: $(\lambda x : [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ .\ x\ [\![\tau]\!])$. Note that, like the constructor abstraction term with the function $\mathcal{C}(-)$, we can just add $[\![\tau]\!]$ to the term and let the term itself figure it out. However since it will not be needed to use the encoded type τ, except for initiation, we add the function to the term.

Since the initiation function is more intuitive than the strip function, proving properties for the term is not necessary. Now putting this together we can define an type application term pre-coding.

Definition 7.34.

$$\mathsf{TApp} = \lambda F : * \to * .\ \Pi\alpha : * .\ F\ \alpha \to \Pi\beta : * .\ (\alpha \to F\ \beta) \to F\ \beta$$

Let $F : * \to *$ and $\mathsf{tapp} : \mathsf{TApp}\ F$ be free variables, then we define the pre-coding function for type application terms as follows:

$$\ulcorner M\ C : \sigma[\alpha := C]\urcorner_\omega = \mathsf{tapp}\ [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ \ulcorner M : (\Pi\alpha : \kappa\ .\ \sigma)\urcorner_\omega\ [\![\sigma[\alpha := C]]\!]$$
$$(\lambda x : [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ .\ x\ [\![C]\!])$$

7.2.4 The Final encoding Function

Definition 7.35. The constructor generator $\mathcal{C}(-)$ and stripfunction $\mathcal{S}(-,-,-)$.

$$\begin{aligned} \mathcal{C}(*) &= (\Pi\alpha : * \;.\; \alpha) \\ \mathcal{C}(\kappa_1 \to \kappa_2) &= \lambda\alpha : \kappa_1 \;.\; \mathcal{C}(\kappa_2) \end{aligned}$$

$$\mathsf{Strip} = \lambda F : * \to * \;.\; \lambda\alpha : * \;.\; \Pi\beta : * \;.\; (\Pi\gamma : * \;.\; F\ \gamma \to \beta) \to \alpha \to \beta$$

$$\begin{aligned} \mathcal{S}(F, \kappa, \lambda\alpha : \kappa \;.\; \tau) &: \mathsf{Strip}\ F\ [\![\Pi\alpha : \kappa \;.\; \tau]\!] \\ &= \lambda\beta : * \;.\; \lambda f : (\Pi\gamma : * \;.\; F\ \gamma \to \beta) \;.\; \lambda x : (\Pi\delta : \kappa \;.\; F\ [\![\tau[\alpha := \delta]]\!]) \;. \\ &\quad f\ [\![\tau[\alpha := \mathcal{C}(\kappa)]]\!]\ (x\ \mathcal{C}(\kappa)) \end{aligned}$$

Definition 7.36. Let $F : * \to *$, then we define the pre-coding function for types as follows:

$$\begin{aligned} [\![\alpha]\!] &= \alpha \\ [\![\tau_1 \to \tau_2]\!] &= F\ [\![\tau_1]\!] \to F\ [\![\tau_2]\!] \\ [\![\Pi\alpha : \kappa \;.\; \tau]\!] &= \Pi\alpha : \kappa \;.\; F\ [\![\tau]\!] \\ [\![\lambda\alpha : \kappa \;.\; C]\!] &= \lambda\alpha : \kappa \;.\; [\![\tau]\!] \\ [\![C_1\ C_2]\!] &= [\![C_1]\!]\ [\![C_2]\!] \end{aligned}$$

Definition 7.37. Typing definitions

$$\begin{aligned} \mathsf{Abs} &= \lambda F : * \to * \;.\; \Pi\alpha : * \;.\; \Pi\beta : * \;.\; (F\ \alpha \to F\ \beta) \to F\ (F\ \alpha \to F\ \beta) \\ \mathsf{App} &= \lambda F : * \to * \;.\; \Pi\alpha : * \;.\; \Pi\beta : * \;.\; (F\ \alpha \to F\ \beta) \to F\ \alpha \to F\ \beta \\ \mathsf{TAbs} &= \lambda F : * \to * \;.\; \Pi\alpha : * \;.\; \mathsf{Strip}\ F\ \alpha \to \alpha \to F\ \alpha \\ \mathsf{TApp} &= \lambda F : * \to * \;.\; \Pi\alpha : * \;.\; F\ \alpha \to \Pi\beta : * \;.\; (\alpha \to F\ \beta) \to F\ \beta \end{aligned}$$

Definition 7.38. Let $F : * \to *$, $\mathsf{abs} : \mathsf{Abs}\ F$, $\mathsf{app} : \mathsf{App}\ F$, $\mathsf{tabs} : \mathsf{TAbs}\ F$ and $\mathsf{tapp} : \mathsf{TApp}\ F$ be free variables, then we define the pre-coding function for terms

as follows:

$$\begin{aligned}
\ulcorner x : \tau \urcorner_\omega &= x \\
\ulcorner \lambda x : \sigma \ . \ M : \sigma \to \tau \urcorner_\omega &= \mathsf{abs}\ [\![\sigma]\!]\ [\![\tau]\!]\ (\lambda x : F\ [\![\sigma]\!]\ .\ \ulcorner M : \tau \urcorner_\omega) \\
\ulcorner M\ N : \tau \urcorner_\omega &= \mathsf{app}\ [\![\sigma]\!]\ [\![\tau]\!]\ \ulcorner M : \sigma \to \tau \urcorner_\omega\ \ulcorner N : \sigma \urcorner_\omega \\
\ulcorner \lambda \alpha : \kappa\ .\ M : \Pi\alpha : \kappa.\tau \urcorner_\omega &= \mathsf{tabs}\ [\![\Pi\alpha : \kappa\ .\ \tau]\!]\ \mathcal{S}(F, \kappa, [\![\lambda\alpha : \kappa\ .\ \tau]\!]) \\
&\quad (\lambda\alpha : \kappa\ .\ \ulcorner M : \tau \urcorner_\omega) \\
\ulcorner M\ C : \sigma[\alpha := C] \urcorner_\omega &= \mathsf{tapp}\ [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ \ulcorner M : (\Pi\alpha : \kappa\ .\ \sigma) \urcorner_\omega\ [\![\sigma[\alpha := C]]\!] \\
&\quad (\lambda x : [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ .\ x\ [\![C]\!])
\end{aligned}$$

Definition 7.39. encoding function for types and terms

$$\underline{[\![C]\!]} = \lambda F : * \to *\ .\ [\![C]\!]$$

$$\begin{aligned}
\underline{\ulcorner M : \tau \urcorner_\omega} = \lambda F : * \to *\ . \\
&\lambda \mathsf{abs} : \mathsf{Abs\ F}\ . \\
&\lambda \mathsf{app} : \mathsf{App\ F}\ . \\
&\lambda \mathsf{tabs} : \mathsf{TAbs\ F}\ . \\
&\lambda \mathsf{tapp} : \mathsf{TApp\ F}\ .\ \ulcorner M : \tau \urcorner_\omega
\end{aligned}$$

7.2.4.1 Proofs

Now that the finished encoding function is presented, we can finalize the proofs from Section 7.2.2.1. First we will proof that the encoding function is well-formed, as in given a System F_ω term, we get a System F_ω term back.

Lemma 7.40. *Let* $\Gamma \vdash M : \tau$ *be a term. Define* Γ_2 *as follows:*

$$\begin{aligned}\Gamma_2 :=& \{x : F\ [\![\sigma]\!] \mid x : \sigma \in \Gamma\} \ \cup \ \{\alpha : \kappa \mid \alpha : \kappa \in \Gamma\} \\ & \cup \ \{F : * \to *, \mathsf{app} : \mathsf{App}\ F, \mathsf{abs} : \mathsf{Abs}\ F\, \mathsf{tapp} : \mathsf{TApp}\ F, \mathsf{tabs} : \mathsf{TAbs}\ F\}\end{aligned}$$

Then $\Gamma_2 \vdash \ulcorner M : \tau \urcorner_\omega : F\ [\![\tau]\!]$.

Proof.
Induction hypothesis

$$N \text{ is a subterm of M} \implies (\Gamma \vdash N : \sigma \implies \Gamma_2 \vdash \ulcorner N : \sigma \urcorner_\omega : F\ [\![\sigma]\!])$$

Base case - lower order terms
Follows from Lemma 7.18.

Case $M = \lambda\alpha : \kappa\ .\ N : (\Pi\alpha : \kappa\ .\ \sigma)$
From Lemma 7.31 we have that

$$\Gamma_2 \vdash \mathcal{S}(F, \kappa, (\lambda\alpha : \kappa\ .\ \tau)) : (\mathsf{Strip}\ F\ [\![\Pi\alpha : \kappa\ .\ \tau]\!])$$

Therefore

$$\Gamma_2 \vdash \ulcorner \lambda\alpha : \kappa\ .\ N : (\Pi\alpha : \kappa\ .\ \sigma)\urcorner_\omega : F\ [\![\Pi\alpha : \kappa\ .\ \tau]\!]$$

follows from

$$\dfrac{\dfrac{\text{IH}}{\Gamma_2, \alpha : \kappa \vdash \ulcorner M : \tau \urcorner_\omega : F\ [\![\tau]\!]}}{\Gamma_2 \vdash (\lambda\alpha : \kappa\ .\ \ulcorner M : \tau \urcorner_\omega) : [\![\Pi\alpha : \kappa\ .\ \tau]\!]}$$

Case $M = N\ C : \tau[\alpha := C]$

$$\dfrac{}{\Gamma_2 \vdash \ulcorner N\ C : \tau[\alpha := C]\urcorner_\omega : F\ [\![\tau[\alpha := C]]\!]}$$

follows from

$$\dfrac{\text{IH}}{\Gamma_2 \vdash \ulcorner M : \Pi\alpha : \kappa\ .\ \sigma \urcorner : F\ [\![\Pi\alpha : \kappa\ .\ \sigma]\!]}$$

and, by making use of Lemma 7.12,

$$\dfrac{\dfrac{\dfrac{\Pi\alpha : \kappa\ .\ F\ [\![\sigma]\!] = [\![\Pi\alpha : \kappa\ .\ \sigma]\!]}{\Gamma_2, x : [\![\Pi\alpha : \kappa\ .\ \sigma]\!] \vdash x : \Pi\alpha : \kappa\ .\ F\ [\![\sigma]\!]} \qquad \dfrac{}{[\![C]\!] : \kappa}}{\Gamma_2, x : [\![\Pi\alpha : \kappa\ .\ \sigma]\!] \vdash x\ [\![C]\!] : (F\ [\![\sigma]\!])[\alpha := [\![C]\!]]}}{\Gamma_2 \vdash (\lambda x : [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ .\ x\ [\![C]\!]) : [\![\Pi\alpha : \kappa\ .\ \sigma]\!] \to F\ [\![\sigma[\alpha := C]]\!])}$$

□

Now we update the Exp typing from which follows that the encoding function is well-

formed.

Definition 7.41.

$$\begin{aligned}\mathsf{Exp} = &\lambda\alpha : (* \to *) \to * \,.\, \Pi F : * \to * \,.\, \mathsf{Abs}\ F \to \mathsf{App}\ F \to \\ &\mathsf{TAbs}\ F \to \mathsf{TApp}\ F \to F\ (\alpha\ F)\end{aligned}$$

Corollary 7.42. *Given some simple term M with $\Gamma \vdash M : \tau$ and Γ_2 as defined in 7.40 we have*

$$\Gamma_2 \vdash \underline{\ulcorner M : \tau \urcorner_\omega} : \mathsf{Exp}\ \underline{[\![\tau]\!]}$$

The Lemma for proving that $\ulcorner - : - \urcorner_\omega$ is an encoding function needs some updating. For this we only have to prove the higher-order term cases.

Lemma 7.43. *The function $\ulcorner - : - \urcorner_\omega$ is normal, i.e. given some term M with $\Gamma \vdash M : \tau$ the encoded term $\underline{\ulcorner M : \tau \urcorner_\omega}$ is in normal form*

Proof.
We have seen in 7.21 why the lower order terms are in normal form. Now note that the terms tabs and tapp are bound and therefore cannot reduce. Therefore a beta reduction can only occur in the type application case. For this to occur, if we have a term $M = M_1\ \tau$, then $\ulcorner M_1 \urcorner_\omega$ has to be of the form $\lambda\alpha : \kappa \,.\, M$, which is impossible. □

Lemma 7.44. *The function $\ulcorner - : - \urcorner_\omega$ is injective, i.e. given some term M and N with $\Gamma_M \vdash M : \tau$ and $\Gamma_N \vdash N : \sigma$ then we have*

$$\ulcorner M : \tau \urcorner_\omega =_\alpha \ulcorner N : \sigma \urcorner_\omega \implies M =_\alpha N$$

Proof.
Base case: see Lemma 7.22. Let M, N be as in the Lemma statement.

Case $M = (\lambda\alpha : \kappa \,.\, M_2) : (\Pi\alpha : \kappa \,.\, \tau_2)$
We have $\ulcorner N : \sigma \urcorner_\omega =_\alpha \ulcorner \lambda\alpha : \kappa \,.\, M_2 : (\Pi\alpha : \kappa \,.\, \tau_2) \urcorner_\omega$, and therefore there is some β, κ_2, σ_2 and N_2 such that $N = \lambda\beta : \kappa_2 \,.\, N_2 : (\Pi\beta : \kappa_2 \,.\, \sigma_2)$. Therefore we have

$$\begin{aligned}&\mathsf{tabs}\ [\![\Pi\alpha : \kappa \,.\, \tau_2]\!]\ \mathcal{S}(F, \kappa, [\![\lambda\alpha : \kappa \,.\, \tau_2]\!])\ (\lambda\alpha : \kappa \,.\, \ulcorner M_2 : \tau_2 \urcorner_\omega) \\ &=_\alpha \mathsf{tabs}\ [\![\Pi\beta : \kappa_2 \,.\, \sigma_2]\!]\ \mathcal{S}(F, \kappa_2, [\![\lambda\beta : \kappa_2 \,.\, \sigma_2]\!])\ (\lambda\beta : \kappa_2 \,.\, \ulcorner N_2 : \sigma_2 \urcorner_\omega)\end{aligned}$$

Therefore by Lemma 7.12 we have $\Pi\alpha : \kappa \,.\, \tau_2 =_\alpha \Pi\beta : \kappa_2 \,.\, \sigma_2$ and by the induction hypothesis we have $M_2 =_\alpha N_2$. Therefore $M =_\alpha N$.

Case $M = M_1\ C : \tau_2[\alpha := C]$
We have $\ulcorner N : \sigma \urcorner_\omega =_\alpha \ulcorner M_1\ C : \tau_1[\alpha := C] \urcorner_\omega$, and therefore there is some β, σ_2, C_2

and N_1 such that $N = N_1\ C_2 : \sigma_2[\alpha := C_2]$. Therefore we have

$$\begin{aligned}
&\mathsf{tapp}\ [\![\Pi\alpha : \kappa\ .\ \tau_2]\!]\ \ulcorner M_1 : (\Pi\alpha : \kappa\ .\ \tau_2)\urcorner_\omega\ [\![\tau_2[\alpha := C]]\!]\ (\lambda x : [\![\Pi\alpha : \kappa\ .\ \tau_2]\!]\ .\ x\ [\![C]\!]) \\
&=_\alpha \mathsf{tapp}\ [\![\Pi\beta : \kappa\ .\ \sigma_2]\!]\ \ulcorner N_1 : (\Pi\beta : \kappa\ .\ \sigma_2)\urcorner_\omega\ [\![\sigma_2[\alpha := C_2]]\!] \\
&\qquad (\lambda x : [\![\Pi\beta : \kappa\ .\ \sigma_2]\!]\ .\ x\ [\![C_2]\!])
\end{aligned}$$

Therefore by Lemma 7.12 we have $\Pi\alpha : \kappa\ .\ \tau_2 =_\alpha \Pi\beta : \kappa\ .\ \sigma_2$ and $\tau_2[\alpha := C] = \sigma_2[\alpha := C_2]$ and therefore we have to have $C =_\alpha C_2$. By the induction hypothesis we have $M_2 =_\alpha N_2$ and therefore $M =_\alpha N$. □

Corollary 7.45. *The function from Definition 7.38 is an encoding function.*

Now for the self-interpreter we have to redefine the foldExp term.

Definition 7.46.

$$\begin{aligned}
&\mathsf{foldExp} : \lambda F : * \to *\ .\ \mathsf{Abs}\ \mathsf{TAbs}\ F \to \mathsf{TApp}\ F \to \mathsf{App}\ F \to F \to \\
&\qquad \lambda\alpha : (* \to *) \to *\ .\ \mathsf{Exp}\ \alpha \to F\ (\alpha\ F) \\
&\mathsf{foldExp} := \lambda F : * \to *\ .\ \lambda \mathsf{abs} : \mathsf{Abs}\ F\ .\ \lambda \mathsf{app} : \mathsf{App}\ F\ .\ \lambda \mathsf{tabs} : \mathsf{TAbs}\ F\ .\ \lambda \mathsf{tapp} : \mathsf{TApp}\ F\ . \\
&\qquad \lambda\alpha : (* \to *) \to *\ .\ \lambda e : \mathsf{Exp}\ \alpha\ .\ e\ F\ \mathsf{abs}\ \mathsf{app}\ \mathsf{tabs}\ \mathsf{tapp}
\end{aligned}$$

Lemma 7.47. *Let M be a closed term with $\emptyset \vdash M : \tau$. Now let $F : * \to *$ be a constructor and $A_1 : \mathsf{Abs}\ F$, $A_2 : \mathsf{App}\ F, A_3 : \mathsf{TAbs}\ F$, $A_4 : \mathsf{TApp}\ F$ be some terms. Then we have*

$$\begin{aligned}
&\mathsf{foldExp}\ C\ A_1\ A_2\ A_3\ A_4\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau\urcorner_\omega} \\
&\qquad \twoheadrightarrow_\beta \ulcorner M : \tau\urcorner_\omega[F := F, \mathsf{app} := A_1, \mathsf{abs} := A_2, \mathsf{tabs} := A_3, \mathsf{tabs} := A_4]
\end{aligned}$$

Now to proof that the encoding function has a term that makes it a self-interpreter, we give the definition of the interpreter.

Definition 7.48. The interpreter for Brown-Palsberg encoding function.

$$
\begin{aligned}
\mathsf{Id} &: * \to * \\
&= \lambda\alpha : * \,.\, \alpha \\
\mathsf{unAbs} &: \mathsf{Abs}\ \mathsf{Id} \\
&= \lambda\alpha : * \,.\, \lambda\beta : * \,.\, \lambda f : \alpha \to \beta \,.\, f \\
\mathsf{unApp} &: \mathsf{App}\ \mathsf{Id} \\
&= \lambda\alpha : * \,.\, \lambda\beta : * \,.\, \lambda f : \alpha \to \beta \,.\, \lambda x : \alpha \,.\, f\ x \\
\mathsf{unTAbs} &: \mathsf{TAbs}\ \mathsf{Id} \\
&= \lambda\alpha : * \,.\, \lambda s : \mathsf{Strip}\ \mathsf{Id}\ \alpha \,.\, \lambda f : \alpha \,.\, f \\
\mathsf{unTApp} &: \mathsf{TApp}\ \mathsf{Id} \\
&= \lambda\alpha : * \,.\, \lambda f : \alpha \,.\, \lambda\beta : * \,.\, \lambda g : \alpha \to \beta \,.\, g\ f \\
\mathsf{unquote} &: \Pi\alpha : (* \to *) \to * \,.\, \mathsf{Exp}\ \alpha \to \mathsf{Id}\ (\alpha\ \mathsf{Id}) \\
&= \mathsf{foldExp}\ \mathsf{Id}\ \mathsf{unAbs}\ \mathsf{unApp}\ \mathsf{unTAbs}\ \mathsf{unTApp}
\end{aligned}
$$

Which brings us to the last part, proving that they form a self-interpreter.

Lemma 7.49. *For all types τ the pair $(\underline{\ulcorner - : \tau \urcorner_\omega}, \mathsf{unquote}\ \underline{[\![\tau]\!]})$ forms a strong closed self-interpreter, i.e. for all types τ and for all closed terms $M : \tau$ we have*

$$\mathsf{unquote}\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta M.$$

Proof. Let the term M be as in the statement and let

$$S = [F := \mathsf{Id}, \mathsf{abs} := \mathsf{unAbs}, \mathsf{app} := \mathsf{unApp}\,\mathsf{tabs} := \mathsf{unTAbs}, \mathsf{tapp} := \mathsf{unTApp}].$$

Base case: lower-order terms
Similar to Lemma 7.27.

Case $M = \lambda\alpha : \kappa \,.\, N : (\Pi\alpha : \kappa \,.\, \sigma)$

$$
\begin{aligned}
&\ulcorner \lambda\alpha : \kappa \,.\, N : \Pi\alpha : \kappa \,.\, \sigma \urcorner_\omega\ S \\
&\twoheadrightarrow_\beta \mathsf{unTAbs}\ (\Pi\alpha : \kappa \,.\, \sigma)\ \mathcal{S}(\mathsf{Id}, \kappa, \lambda\alpha : \kappa \,.\, \sigma)\ ((\lambda\alpha : \kappa \,.\, \ulcorner N : \sigma \urcorner_\omega)S) \\
&\twoheadrightarrow_\beta \mathsf{unTAbs}\ (\Pi\alpha : \kappa \,.\, \sigma)\ \mathcal{S}(\mathsf{Id}, \kappa, \lambda\alpha : \kappa \,.\, \sigma)\ (\lambda\alpha : \kappa \,.\, N) \\
&\twoheadrightarrow_\beta \lambda\alpha : \kappa \,.\, N
\end{aligned}
$$

Case $M = N\ C : \sigma_2[\alpha := C]$

$$
\begin{aligned}
&\ulcorner M = N\ C : \sigma_1[\alpha := C] \urcorner_\omega\ S \\
&\twoheadrightarrow_\beta \mathsf{unTApp}\ (\Pi\alpha : \kappa \,.\, \sigma_2)\ (\ulcorner N : (\Pi\alpha : \kappa \,.\, \sigma_2) \urcorner_\omega\ S)\ (\sigma_1[\alpha := C]) \\
&\qquad (\lambda x : (\Pi\alpha : \kappa \,.\, \sigma_2) \,.\, x\ C) \\
&\twoheadrightarrow_\beta (\lambda x : (\Pi\alpha : \kappa \,.\, \sigma_2) \,.\, x\ C)\ (\ulcorner N : (\Pi\alpha : \kappa \,.\, \sigma_2) \urcorner_\omega\ S) \\
&\twoheadrightarrow_\beta (\lambda x : (\Pi\alpha : \kappa \,.\, \sigma_2) \,.\, x\ C)\ N \\
&\twoheadrightarrow_\beta N\ C
\end{aligned}
$$

Then by Lemma 7.47 we have that the pair $(\underline{\ulcorner - : \tau \urcorner_\omega}, \mathsf{unquote}\ \underline{[\![\tau]\!]})$ form a self-interpreter for any type τ. $\square$

7.3 Function Definition Scheme

We have seen the term $\mathsf{foldExp}$ and that we can define a recognizer with it. If we think about it, it suspiciously looks like a term for a function definition scheme.

Definition 7.50. (BP Function Definition Scheme)
Let $[\![-]\!]$ be an encoding function on constructors. Let $\ulcorner - : \tau \urcorner \in \Lambda^\tau \to \Lambda^{[\![\tau]\!]}$ be an encoding function on terms. Let $F : * \to *$ and let $H_1 : \mathsf{Abs}\ F, H_2 : \mathsf{App}\ F, H_3 : \mathsf{TAbs}\ F$ and $H_4 : \mathsf{TApp}\ F$. Then there exists a $H : \Pi\alpha : * \,.\, \alpha \to F\ \alpha$ such that:

$$
\begin{aligned}
&H\ [\![\sigma_1 \to \sigma_2]\!]\ \ulcorner \lambda x : \sigma_1\ .\ M : \sigma_1 \to \sigma_2 \urcorner \\
&=_\beta H_1\ ([\![\sigma_1]\!]\ F)\ ([\![\sigma_2]\!]\ F)\ (H\ [\![\sigma_2]\!]\ \underline{\ulcorner M : \sigma_2 \urcorner_\omega})
\end{aligned}
$$

$$
\begin{aligned}
&H\ [\![\sigma_2]\!]\ \underline{\ulcorner M\ N : \sigma_2 \urcorner_\omega} \\
&=_\beta H_2\ ([\![\sigma_1]\!]\ F)\ ([\![\sigma_2]\!]\ F)\ ((H\ [\![\sigma_1 \to \sigma_2]\!]\ \underline{\ulcorner M : \sigma_1 \to \sigma_2 \urcorner_\omega}) \\
&\qquad (H\ [\![\sigma_1]\!]\ \underline{\ulcorner N : \sigma_1 \urcorner_\omega}))
\end{aligned}
$$

$$
\begin{aligned}
&H\ [\![\Pi\alpha : \kappa.\tau]\!]\ \underline{\ulcorner \lambda\alpha : \kappa\ .\ M : \Pi\alpha : \kappa.\tau \urcorner_\omega} \\
&=_\beta H_3\ ([\![\Pi\alpha : \kappa\ .\ \tau]\!]\ F)\ \mathcal{S}(F, \kappa, ([\![\lambda\alpha : \kappa\ .\ \tau)\ F]\!]) \\
&\qquad (\lambda\alpha : \kappa\ .\ (H\ [\![\tau]\!]\ \underline{\ulcorner M : \tau \urcorner_\omega}))
\end{aligned}
$$

$$
\begin{aligned}
&H\ [\![\sigma[\alpha := C]]\!]\ \underline{\ulcorner M\ C : \sigma[\alpha := C] \urcorner_\omega} \\
&=_\beta H_4\ ([\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ F)\ (H\ [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ \underline{\ulcorner M : (\Pi\alpha : \kappa\ .\ \sigma) \urcorner_\omega}) \\
&\qquad ([\![\sigma[\alpha := C]]\!]\ F)\ (\lambda x : [\![\Pi\alpha : \kappa\ .\ \sigma]\!]\ F)\ .\ x\ ([\![C]\!]\ F))
\end{aligned}
$$

Lemma 7.51. *The Brown-Palsberg encoding function has a BP function definition scheme.*

Proof. Let
$$H := \mathsf{foldExp}\ F\ H_1\ H_2\ H_3\ H_4$$

Then we have

$$
\begin{array}{ll}
 & H\ \underline{\llbracket \sigma_1 \to \sigma_2 \rrbracket}\ \underline{\ulcorner \lambda x : \sigma_1\ .\ M : \sigma_1 \to \sigma_2 \urcorner_\omega} \\
\twoheadrightarrow_\beta & \ulcorner \lambda x : \sigma_1\ .\ M : \sigma_1\ \to \sigma_2 \urcorner_\omega\ S \\
= & H_1\ (\llbracket \sigma_1 \rrbracket\ S)\ (\llbracket \sigma_2 \rrbracket\ S)\ (\ulcorner M : \sigma_2 \urcorner_\omega\ S) \\
=_\beta & H_1\ (\underline{\llbracket \sigma_1 \rrbracket}\ F)\ (\llbracket \sigma_2 \rrbracket\ F)\ (H\ \underline{\llbracket \sigma_2 \rrbracket}\ \underline{\ulcorner M : \sigma_2 \urcorner_\omega})
\end{array}
$$

$$
\begin{array}{ll}
 & H\ \underline{\llbracket \sigma_2 \rrbracket}\ \underline{\ulcorner M\ N : \sigma_2 \urcorner_\omega} \\
\twoheadrightarrow_\beta & \ulcorner M\ N : \sigma_2 \urcorner_\omega\ S \\
= & H_2\ (\llbracket \sigma_1 \rrbracket\ S)\ (\llbracket \sigma_2 \rrbracket\ s)\ ((\ulcorner M : \sigma_1 \to \sigma_2 \urcorner_\omega\ S)\ (\ulcorner N : \sigma_1 \urcorner_\omega\ S)) \\
=_\beta & H_2\ (\underline{\llbracket \sigma_1 \rrbracket}\ F)\ (\underline{\llbracket \sigma_2 \rrbracket}\ F)\ ((H\ \underline{\llbracket \sigma_1 \to \sigma_2 \rrbracket}\ \underline{\ulcorner M : \sigma_1 \to \sigma_2 \urcorner_\omega}) \\
 & (H\ \underline{\llbracket \sigma_1 \rrbracket}\ \underline{\ulcorner N : \sigma_1 \urcorner_\omega}))
\end{array}
$$

$$
\begin{array}{ll}
 & H\ \underline{\llbracket \Pi\alpha : \kappa . \tau \rrbracket}\ \underline{\ulcorner \lambda\alpha : \kappa\ .\ M : \Pi\alpha : \kappa . \tau \urcorner_\omega} \\
\twoheadrightarrow_\beta & \ulcorner \lambda\alpha : \kappa\ .\ M : \Pi\alpha : \kappa . \tau \urcorner_\omega\ S \\
= & H_3\ (\llbracket \Pi\alpha : \kappa\ .\ \tau \rrbracket\ S)\ \mathcal{S}(F, \kappa, \llbracket \lambda\alpha : \kappa\ .\ \tau \rrbracket\ S)\ (\lambda\alpha : \kappa\ .\ (\ulcorner M : \tau \urcorner_\omega\ S)) \\
=_\beta & H_3\ (\underline{\llbracket \Pi\alpha : \kappa\ .\ \tau \rrbracket}\ F)\ \mathcal{S}(F, \kappa, \underline{\llbracket \lambda\alpha : \kappa\ .\ \tau \rrbracket}\ F)\ (\lambda\alpha : \kappa\ .\ (H\ \underline{\llbracket \tau \rrbracket}\ \underline{\ulcorner M : \tau \urcorner_\omega}))
\end{array}
$$

$$
\begin{array}{ll}
 & H\ \underline{\llbracket \sigma[\alpha := C] \rrbracket}\ \underline{\ulcorner M\ C : \sigma[\alpha := C] \urcorner_\omega} \\
\twoheadrightarrow_\beta & \ulcorner M\ C : \sigma[\alpha := C] \urcorner_\omega\ S \\
= & H_4\ (\llbracket \Pi\alpha : \kappa\ .\ \sigma \rrbracket\ S)\ (\ulcorner M : (\Pi\alpha : \kappa\ .\ \sigma) \urcorner_\omega\ S)\ \llbracket \sigma[\alpha := C] \rrbracket \\
 & (\lambda x : \llbracket \Pi\alpha : \kappa\ .\ \sigma \rrbracket\ .\ x\ \llbracket C \rrbracket) \\
=_\beta & H_4\ (\underline{\llbracket \Pi\alpha : \kappa\ .\ \sigma \rrbracket}\ F)\ (H\ \underline{\llbracket \Pi\alpha : \kappa\ .\ \sigma \rrbracket}\ \underline{\ulcorner M : (\Pi\alpha : \kappa\ .\ \sigma) \urcorner_\omega})\ (\underline{\llbracket \sigma[\alpha := C] \rrbracket}\ F) \\
 & (\lambda x : (\underline{\llbracket \Pi\alpha : \kappa\ .\ \sigma \rrbracket}\ F)\ .\ x\ (\underline{\llbracket C \rrbracket}\ F))
\end{array}
$$

□

Note that this function definition scheme is not as strong as the Mogensen and Barendregt function definition scheme. This is since we miss the variable case and instead of giving the function H to $H_1, \ldots H_4$, we already apply it to the next part of the code. It would be interesting to see if a self-interpreter can be made with such a function definition scheme.

7.4 Applications

Now the question remain, how powerful is the BP self-interpreter. To get a feeling for this, we will show some applications and there after restriction. Brown and Palsberg present in their paper [10] that the encoding is strong enough to do testing predicates for abstraction and application, size measurement, normal-form checking and continuation-passing-style transformation. Here we will expand on this by showing that we can differentiate between the four types of terms and we will show we that we can compare the structures of two terms, resulting in a α-equivalence testing term, given that the types are the same.

7.4.1 Term Type Tester

First we will define a term such that we know what for a type of term it is. Let say we have a closed term M and we want to know if it is an (constructor) abstraction term or (constructor) application term. Then we can define a function that does this as follows:

$$f(M) = \begin{cases} 1 & \text{if } M = \lambda x : \sigma \;.\; N \\ 2 & \text{if } M = N_1\ N_2 \\ 3 & \text{if } M = \lambda\alpha : \sigma \;.\; N \\ 4 & \text{if } M = N\ C \end{cases}$$

Now we want to define a term, let's say isTerm, such that $\mathsf{isTerm}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \overline{f(M)}$. We can do this by defining four different terms and then make use of the $\mathsf{foldExp}$ lemma. Let us first define $\mathsf{KNat} = \lambda\alpha : * \;.\; \mathsf{Nat}$ and define the four terms as follows:

$$\begin{aligned}
\mathsf{TAbs} : \mathsf{Abs\ KNat} &= \lambda\alpha : * \;.\; \lambda\beta : * \;.\; \lambda f : \mathsf{Nat} \to \mathsf{Nat} \;.\; \overline{1} \\
\mathsf{TApp} : \mathsf{App\ KNat} &= \lambda\alpha : * \;.\; \lambda\beta : * \;.\; \lambda f : \mathsf{Nat} \;.\; \lambda x : \mathsf{Nat} \;.\; \overline{2} \\
\mathsf{TTAbs} : \mathsf{TAbs\ KNat} &= \lambda\alpha : * \;.\; \lambda s : \mathsf{Strip\ KNat}\ \alpha \;.\; \lambda f : \alpha \;.\; \overline{3} \\
\mathsf{TTapp} : \mathsf{TApp\ KNat} &= \lambda\alpha : * \;.\; \lambda f : \mathsf{Nat} \;.\; \lambda\beta : * \;.\; \lambda i : \alpha \to \beta \;.\; \overline{4}
\end{aligned}$$

Now by the function definition scheme we can say there is a term isTerm that makes uses of the defined terms. That this term can distinguish between terms can be shown by proving every case, like is done below for the abstraction term case.

$$\begin{aligned}
&T\ \underline{\ulcorner \lambda x : \sigma \;.\; N : \sigma \to \tau \urcorner_\omega} \\
&\quad \twoheadrightarrow_\beta \ulcorner \lambda x : \sigma \;.\; N : \sigma \to \tau \urcorner_\omega\ S \\
&\quad = \mathsf{TAbs}\ (F\ [\![\sigma]\!])\ (F\ [\![\tau]\!])\ (\lambda x : F\ [\![\sigma]\!] \;.\; (\ulcorner M : \tau \urcorner_\omega\ S)) \\
&\quad \twoheadrightarrow_\beta \overline{1}
\end{aligned}$$

From this, it follows that we know what term we are dealing with, and therefore we have the following corollary:

Corollary 7.52. *We have terms* isAbs, isApp, isTAbs *and* isTApp *such that*

$$\begin{aligned}
\mathsf{isAbs}\ M =_\beta \mathsf{true} &\iff M =_\alpha \lambda x : \tau\ .\ M_2 \\
\mathsf{isApp}\ M =_\beta \mathsf{true} &\iff M =_\alpha M_1\ M_2 \\
\mathsf{isTAbs}\ M =_\beta \mathsf{true} &\iff M =_\alpha \lambda \alpha : \kappa\ .\ M_2 \\
\mathsf{isTApp}\ M =_\beta \mathsf{true} &\iff M =_\alpha M_2\ \tau
\end{aligned}$$

7.4.2 Normal Form Checker

For the second application of the Brown-Palsberg self-interpreter we will define a term that can check if the encoded term is in normal form or not. While this seems like a case of applying our previous defined term isTerm, this is unfortunately not possible. To see this, we can think about what happens when we want to check if an application term $M = M_1\ M_2$ is in normal form. For the encoded version we would have to define a term of type $\mathsf{App\ KBool}$ which is true if M_1 is an abstraction. Since we have defined our F to be KBool, if we apply our term using $\mathsf{foldExp}$, we would get two times the type Bool and two terms of type Bool as input, that represent if those terms were in normal form or not. So the problem we now have is that we lost the information of the encoded term, so therefore we can't check what type they were, and therefore we can't create a normal form term.

Now to get around this, instead of using a singular Boolean as output, we will use a pair of Booleans. The first Boolean will represent if the term is in normal form or not, and the second Boolean will represent if the term is not in normal form or if the term is an abstraction. Now if we are in the application case and the second Boolean of our first term is false, we know that either that term is an application or is not in normal form. Either way, we can conclude that the application is not in normal form.

Now we can define our term that will check if a encoded term is in normal form or not. For this we will use some the terms $\mathsf{pairb} := \mathsf{pair\ Bool}$, $\mathsf{fstb} := \mathsf{fst\ Bool}$ and $\mathsf{sndb} := \mathsf{sndBool}$ and the constructors $\mathsf{Bools} := \mathsf{Pair\ Bool}$ and $\mathsf{KBools} := \lambda\alpha : * \,.\, \mathsf{Bools}$.

$$
\begin{aligned}
\mathsf{NFAbs} &: \mathsf{Abs\ KBools} \\
&:= \lambda\alpha : * \,.\, \lambda\beta : * \,.\, \lambda f : \mathsf{Bools} \to \mathsf{Bools} \,. \\
&\quad \mathsf{pairb}\ (\mathsf{fstb}\ (f\ (\mathsf{pairb\ true\ true})))\ \mathsf{false} \\
\mathsf{NFApp} &: \mathsf{App\ KBools} \\
&:= \lambda\alpha : * \,.\, \lambda\beta : * \,.\, \lambda x : \mathsf{Bools} \,.\, \lambda y : \mathsf{Bools} \,. \\
&\quad (\lambda z : \mathsf{Bool} \,.\, \mathsf{pairb}\ z\ z)\ (\mathsf{and}\ (\mathsf{sndb}\ x)\ (\mathsf{fstb}\ y)) \\
\mathsf{NFTAbs} &: \mathsf{TAbs\ KBools} \\
&:= \lambda\alpha : * \,.\, \lambda s : \mathsf{Strip\ KBools}\ \alpha \,.\, \lambda f : \alpha \,. \\
&\quad \mathsf{pairb}\ (\mathsf{fstb}\ (s\ \mathsf{Bools}\ (\lambda\alpha : * \,.\, \lambda x : \mathsf{Bools} \,.\, x)\ f))\ \mathsf{false} \\
\mathsf{NFTApp} &: \mathsf{TApp\ KBools} \\
&:= \lambda\alpha : * \,.\, \lambda f : \mathsf{Bools} \,.\, \lambda\beta : * \,.\, \lambda \mathsf{inst} : \alpha \to \mathsf{Bools} \,. \\
&\quad (\lambda z : \mathsf{Bool} \,.\, \mathsf{pairb}\ z\ z)\ (\mathsf{sndb}\ f)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{isNFPair} &:= \mathsf{foldExp\ KBools\ NFAbs\ NFApp\ NFTAbs\ NFTApp} \\
\mathsf{isNF} &:= \lambda\alpha : (* \to *) \to * \,.\, \lambda t : \mathsf{Exp}\ \alpha \,.\, \mathsf{fstb}\ (\mathsf{isNFPair}\ \alpha\ t)
\end{aligned}
$$

Now we claim that the term isNF, given a closed term M in the encoding of Brown-Palsberg, reduces to true if M is in normal form and else reduces to false, i.e. we want to prove the following statement

$$M \text{ is in normal form} \iff \mathsf{isNF}\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \mathsf{true}$$

This statement will be proven in two steps. First we will proof the necessary condition, since in this case we will be able to ignore free variables, which makes the proof easier

Lemma 7.53. *Given a term $M : \tau$, we have*

$$M \text{ is not in normal form} \implies \mathsf{isNFPair}\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega}\ \twoheadrightarrow_\beta\ \mathsf{pairb\ false\ false}$$

Proof. Let

$$S := [F := \mathsf{KBools}, \mathsf{abs} := \mathsf{nfabs}, \mathsf{app} := \mathsf{nfapp}, \mathsf{tabs} := \mathsf{nftabs}, \mathsf{tapp} := \mathsf{nftapp}]$$

Assume M is not in normal form. Then there is at least one subterm N of M such that N is either of the form $N = (\lambda x : \sigma\ .\ N_1)\ N_2$ or $N = N_2\ \sigma$.

Base case $N = (\lambda x : \sigma\ .\ N_1)\ N_2$

$$\begin{aligned}
\ulcorner N : \tau \urcorner_\omega\ S \twoheadrightarrow_\beta\ & (\lambda x : \mathsf{Bool}\ .\ \mathsf{pairb}\ x\ x)\ (\mathsf{and}\ (\mathsf{sndb}\ (\ulcorner(\lambda x : \sigma\ .\ N_1) : \sigma \to \tau \urcorner_\omega)\ S) \\
& (\mathsf{fstb}\ (\ulcorner N_2 : \sigma \urcorner_\omega\ S))) \\
\twoheadrightarrow_\beta\ & (\lambda x : \mathsf{Bool}\ .\ \mathsf{pairb}\ x\ x)\ (\mathsf{and\ false}\ (\mathsf{fstb}\ (\ulcorner N_2 : \sigma \urcorner_\omega\ S))) \\
\twoheadrightarrow_\beta\ & (\lambda x : \mathsf{Bool}\ .\ \mathsf{pairb}\ x\ x)\ \mathsf{false} \\
\twoheadrightarrow_\beta\ & \mathsf{pairb\ false\ false}
\end{aligned}$$

Base case $N = (\lambda \alpha : \kappa\ .\ N_2)\ C$

$$\begin{aligned}
\ulcorner N : \tau[\alpha := C] \urcorner_\omega\ S \twoheadrightarrow_\beta\ & (\lambda z : \mathsf{Bool}\ .\ \mathsf{pairb}\ z\ z) \\
& (\mathsf{sndb}\ (\ulcorner(\lambda \alpha : \kappa\ .\ N_2) : (\Pi \alpha : \kappa\ .\ \tau) \urcorner_\omega\ S)) \\
\twoheadrightarrow_\beta\ & (\lambda z : \mathsf{Bool}\ .\ \mathsf{pairb}\ z\ z)\ \mathsf{false} \\
\twoheadrightarrow_\beta\ & \mathsf{cpair\ false\ false}
\end{aligned}$$

Induction case $M = \lambda x : \sigma_1\ .\ M_2$

$$\begin{aligned}
\ulcorner M : \tau \urcorner_\omega\ S \twoheadrightarrow_\beta\ & \mathsf{nfAbs\ Bools\ Bools}\ (\lambda x : \mathsf{Bools}\ .\ \ulcorner M_2 : \sigma_2 \urcorner_\omega) \\
\twoheadrightarrow_\beta\ & \mathsf{pairb}\ (\mathsf{fstb}\ ((\lambda x : \mathsf{Bools}\ .\ \ulcorner M_2 : \sigma_2 \urcorner_\omega)\ (\mathsf{pairb\ true\ true})))\ \mathsf{false} \\
\twoheadrightarrow_\beta\ & \mathsf{pairb}\ (\mathsf{fstb}\ ((\lambda x : \mathsf{Bools}\ .\ \mathsf{pairb\ false\ false})\ (\mathsf{pairb\ true\ true})))\ \mathsf{false} \\
\twoheadrightarrow_\beta\ & \mathsf{pairb}\ (\mathsf{fstb}\ (\mathsf{pairb\ false\ false}))\ \mathsf{false} \\
\twoheadrightarrow_\beta\ & \mathsf{pairb\ false\ false}
\end{aligned}$$

Induction case $M = M_1\ M_2$

$$\begin{aligned}
\ulcorner M_1\ M_2 : \tau \urcorner_\omega\ S \twoheadrightarrow_\beta\ & \mathsf{nfApp\ Bools\ Bools}\ (\ulcorner M_1 : \sigma \to \tau \urcorner_\omega\ S)\ (\ulcorner M_2 : \sigma \urcorner_\omega\ S) \\
\twoheadrightarrow_\beta\ & (\lambda z : \mathsf{Bool}\ .\ \mathsf{pairb})\ (\mathsf{and}\ (\mathsf{sndb}\ (\ulcorner M_1 : \sigma \to \tau \urcorner_\omega\ S)) \\
& (\mathsf{fstb}\ (\ulcorner M_2 : \sigma \urcorner_\omega\ S))) \\
\twoheadrightarrow_\beta\ & (\lambda z : \mathsf{Bool}\ .\ \mathsf{pairb})\ (\mathsf{and\ false\ false}) \\
=\ & \mathsf{pairb\ false\ false}
\end{aligned}$$

Induction case $M = \lambda : \alpha : * \ . \ M_2$

$$\begin{aligned}
&\ulcorner \lambda : \alpha : * \ . \ M_2 : (\Pi\alpha : * \ . \ \sigma\urcorner_\omega \ S \\
&\twoheadrightarrow_\beta \mathsf{nfTAbs} \ [\![\Pi\alpha : \kappa \ . \ \tau]\!] \ \mathcal{S}(F, \kappa, [\![\lambda\alpha : \kappa \ . \ \tau]\!]) \ (\lambda\alpha : \kappa \ . \ \ulcorner M_2 : \sigma\urcorner_\omega \ S) \\
&\twoheadrightarrow_\beta \mathsf{pairb} \ (\mathsf{fstb} \ (\ulcorner M_2 : \sigma\urcorner_\omega \ S) \ \mathsf{false} \\
&\twoheadrightarrow_\beta \mathsf{pairb} \ \mathsf{false} \ \mathsf{false}
\end{aligned}$$

Induction case $M_2 \ C$

$$\begin{aligned}
\ulcorner M_2 \ C : \sigma[\alpha := C]\urcorner_\omega \ S \twoheadrightarrow_\beta \ & \mathsf{nfTApp} \ [\![\Pi\alpha : \kappa \ . \ \sigma]\!] \ \ulcorner M : (\Pi\alpha : \kappa \ . \ \sigma)\urcorner_\omega \ [\![\sigma[\alpha := C]]\!] \\
& (\lambda x : [\![\Pi\alpha : \kappa \ . \ \sigma]\!] \ . \ x \ [\![C]\!]) \\
& \twoheadrightarrow_\beta (\lambda z : \mathsf{Bool} \ . \ \mathsf{pairb} \ z \ z) \ (\mathsf{sndb} \ (\ulcorner M : (\Pi\alpha : \kappa \ . \ \sigma)\urcorner_\omega \ S)) \\
& \twoheadrightarrow_\beta (\lambda z : \mathsf{Bool} \ . \ \mathsf{pairb} \ z \ z) \ \mathsf{false} \\
& = \mathsf{pairb} \ \mathsf{false} \ \mathsf{false}
\end{aligned}$$

Therefore we have for a closed term $\emptyset \vdash M : \tau$ not in normal form that

$$\mathsf{foldExp} \ \mathsf{KBools} \ \mathsf{NFAbs} \ \mathsf{NFApp} \ \mathsf{NFTAbs} \ \mathsf{NFTApp} \ \underline{[\![\tau]\!]} \ \underline{\ulcorner M : \tau\urcorner_\omega} \twoheadrightarrow_\beta \mathsf{pairb} \ \mathsf{false} \ \mathsf{false}$$

□

This time to prove the sufficient conditions we have to deal with the free variables.

Lemma 7.54. *Given a term* $\emptyset \vdash M : \tau$*, we have*

$$\begin{aligned}
& M \textit{ is an abstraction and is in normal form} \\
& \qquad \Longrightarrow \mathsf{isNFPair} \ \underline{[\![\tau]\!]} \ \underline{\ulcorner M : \tau\urcorner_\omega} \ \twoheadrightarrow_\beta \ \mathsf{pairb} \ \mathsf{true} \ \mathsf{false} \\
& M \textit{ is an application and is in normal form} \\
& \qquad \Longrightarrow \mathsf{isNFPair} \ \underline{[\![\tau]\!]} \ \underline{\ulcorner M : \tau\urcorner_\omega} \ \twoheadrightarrow_\beta \ \mathsf{pairb} \ \mathsf{true} \ \mathsf{true}
\end{aligned}$$

Proof. First we define a substitution term, let

$$S := [F := \mathsf{KBools}, \mathsf{abs} := \mathsf{nfabs}, \mathsf{app} := \mathsf{nfapp}, \mathsf{tabs} := \mathsf{nftabs}, \mathsf{tapp} := \mathsf{nftapp}]$$

By Lemma 7.47 we have

$$\mathsf{isNFPair} \ \underline{[\![\tau]\!]} \ \underline{\ulcorner M : \tau\urcorner_\omega} \twoheadrightarrow_\beta \ulcorner M : \tau\urcorner_\omega \ S$$

Now to prove the statement in the Lemma, we will have to prove that $\mathsf{fstb}(\ulcorner : \ \urcorner_\omega S) \twoheadrightarrow_\beta$ true. We will do this by proving that for a closed term $M : \tau$ we have $\ulcorner M : \tau\urcorner_\omega \ S \twoheadrightarrow_\beta$ $\mathsf{pairb} \ \mathsf{true} \ \mathsf{false}$ if M is an abstraction term and $\ulcorner M : \tau\urcorner_\omega \ S \twoheadrightarrow_\beta \mathsf{pairb} \ \mathsf{true} \ \mathsf{true}$ if M is an application term. However to do this this for only closed term is not obvious at all. Therefore we instead will proof a different statement, for which we will define some special terms.

$$\begin{aligned}
\mathsf{fvand}_N &:= \mathsf{and} \ (\mathsf{and} \ x_1 \ \ldots (\mathsf{and} \ x_{n-1} \ x_n)) \ (\mathsf{and} \ y_1 \ \ldots (\mathsf{and} \ y_{m-1} \ y_m)) \\
& \quad \text{where } \{x_1, \ldots, x_n\} \subseteq \{\mathsf{fstb} \ e : e \in \mathbf{FV}(N)\} \text{ and} \\
& \quad \{y_1, \ldots, y_m\} \subseteq \{\mathsf{sndb} \ e : e \in \mathbf{FV}(N)\} \\
\mathsf{fvandp}_N &:= \mathsf{pairb} \ \mathsf{fvand}_N \ \mathsf{fvand}_N
\end{aligned}$$

Note that fvand_N is defined as multiple terms and that fvand_N can be equal to true when n and m are zero. Now we will prove that for an open term $M : \tau$ we have $\ulcorner M : \tau \urcorner_\omega \ S =_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ false})\ \mathsf{fvandp}_M$ if M is an abstraction term and $\ulcorner M : \tau \urcorner_\omega \ S =_\beta \mathsf{andp}(\mathsf{pairb\ true\ false})\ \mathsf{fvandp}_M$ if M is an application.

Base case $M = x$:

$$x\ S = x =_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ true})\ \mathsf{fvandp}_M$$

Induction Case $M = \lambda x : \sigma\ .\ N$

$$\begin{aligned}
&\ulcorner \lambda x : \sigma\ .\ M_2 : \sigma_1 \to \sigma_2 \urcorner_\omega\ S\\
&\twoheadrightarrow_\beta \mathsf{nfApp\ PairB\ PairB}\ (\lambda x : \mathsf{PairB}\ .\ \ulcorner M_2 : \sigma_2 \urcorner_\omega\ S)\\
&\twoheadrightarrow_\beta \mathsf{pairb}\ (\mathsf{fstb}\ ((\lambda x : \mathsf{PairB}\ .\ (\ulcorner M_2 : \sigma_2 \urcorner_\omega\ S))\ (\mathsf{pairb\ true\ true})))\ \mathsf{false}\\
&\twoheadrightarrow_\beta \mathsf{pairb}\ (\mathsf{fstb}\ ((\ulcorner M_2 : \sigma_2 \urcorner_\omega\ S)[x := \mathsf{pairb\ true\ true}]))\ \mathsf{false}\\
&=_\beta \mathsf{pairb}\ ((\mathsf{and\ true\ fvand}_N)[x := \mathsf{pairb\ true\ true}])\ \mathsf{false}\\
&=_\beta \mathsf{pairb}\ (\mathsf{and\ true\ fvand}_M)\ \mathsf{false}\\
&=_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ false})\ \mathsf{fvandp}_M
\end{aligned}$$

Induction Case $M = N_1\ N_2$

$$\begin{aligned}
&\ulcorner N_1\ N_2 : \tau \urcorner_\omega\ S\\
&\twoheadrightarrow_\beta \mathsf{nfApp}\ \sigma \to \tau\ \sigma\ (\ulcorner N_1 : \sigma \to \tau \urcorner_\omega\ S)\ (\ulcorner N_2 : \sigma \urcorner_\omega\ S)\\
&\twoheadrightarrow_\beta (\lambda z : \mathsf{PairB}\ .\ \mathsf{pairb}\ z\ z)\ (\mathsf{and}\ (\mathsf{sndb}\ (\ulcorner N_1 : \sigma \to \tau \urcorner_\omega\ S))\ (\mathsf{fstb}\ (\ulcorner N_2 : \sigma \urcorner_\omega\ S)))\\
&=_\beta (\lambda z : \mathsf{PairB}\ .\ \mathsf{pairb}\ z\ z)\ (\mathsf{and}\ (\mathsf{sndb}\ x)\ (\mathsf{fstb}\ y))\\
&(\mathsf{andp}\ (\mathsf{pairb\ true\ true})\ \mathsf{fvandp}_{N_1})\ (\mathsf{andp}\ (\mathsf{pairb\ true}\ ?)\ \mathsf{fvandp}_{N_2})\\
&=_\beta (\lambda z : \mathsf{PairB}\ .\ \mathsf{pairb}\ z\ z)\ (\mathsf{and}\ (\mathsf{and\ true\ fvand}_{N_1})\ (\mathsf{and\ true\ fvand}_{N_2}))\\
&=_\beta (\lambda z : \mathsf{PairB}\ .\ \mathsf{pairb}\ z\ z)\ (\mathsf{and\ true\ fvand}_M)\\
&\twoheadrightarrow_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ true})\ \mathsf{fvandp}_M
\end{aligned}$$

Induction Case $M = \lambda \alpha : \kappa\ .\ N$

$$\begin{aligned}
&\ulcorner \lambda \alpha : \kappa\ .\ M_2 : (\Pi \alpha : \kappa\ .\ \tau) \urcorner_\omega\ S\\
&\twoheadrightarrow_\beta \mathsf{NFTabs}\ [\![\Pi \alpha : \kappa\ .\ \tau]\!]\ S(F, \kappa, [\![\lambda \alpha\ .\ \tau]\!])\ (\lambda \alpha : \kappa\ .\ (\ulcorner M_2 : \tau \urcorner_\omega\ S))\\
&\twoheadrightarrow_\beta \mathsf{pairb}\ (\mathsf{fstb}\ (S(F, \kappa, [\![\lambda \alpha\ .\ \tau]\!])\ \mathsf{Bools}\ (\lambda \alpha : \kappa\ .\ \lambda x : \mathsf{Bools}\ .\ x)\\
&\qquad (\ulcorner M_2 : \tau \urcorner_\omega\ S)))\ \mathsf{false}\\
&\twoheadrightarrow_\beta \mathsf{pairb}\ (\mathsf{fstb}\ \ulcorner M_2 : \tau \urcorner_\omega)\ \mathsf{false}\\
&=_\beta \mathsf{pairb}\ (\mathsf{and\ true\ fvand}_N)\ \mathsf{false}\\
&=_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ false})\ \mathsf{fvandp}_M
\end{aligned}$$

Induction Case $M = M_2\ C$

$$\begin{aligned}
&\ulcorner M_2\ C : \sigma[\alpha := C] \urcorner_\omega\ S\\
&\twoheadrightarrow_\beta \mathsf{nfTApp}\ [\![\Pi \alpha : \kappa\ .\ \sigma]\!]\ \ulcorner M_2 : \Pi \alpha : \kappa\ .\ \sigma \urcorner_\omega\ [\![\sigma[\alpha := C]]\!]\ (\lambda x : [\![\Pi \alpha : \kappa\ .\ \sigma]\!] x\ [\![C]\!])\\
&\twoheadrightarrow_\beta (\lambda z : \mathsf{PairB}\ .\ \mathsf{pairb}\ z\ z)\ (\mathsf{sndb}\ (\ulcorner M_2 : \Pi \alpha : \kappa\ .\ \sigma \urcorner_\omega\ S))\\
&=_\beta (\lambda z : \mathsf{PairB}\ .\ \mathsf{pairb}\ z\ z)\ (\mathsf{and\ true\ fvand}_N)\\
&=_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ true})\ \mathsf{fvandp}_M
\end{aligned}$$

Now for a closed term $M : \tau$ we have that the term fvandp_M is always equivalent to pairb true true. Therefore we have that

$$\ulcorner M : \tau \urcorner_\omega \ S =_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ true})\ (\mathsf{pairb\ true\ true}) \twoheadrightarrow_\beta \mathsf{pairb\ true\ true}$$

if M is an abstraction and

$$\ulcorner M : \tau \urcorner_\omega \ S =_\beta \mathsf{andp}\ (\mathsf{pairb\ true\ true})\ (\mathsf{pairb\ true\ true}) \twoheadrightarrow_\beta \mathsf{pairb\ true\ true}$$

if M is an application. Therefore we can conclude, that if M is closed term, we have

$$\mathsf{isNF}\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \mathsf{true}$$

□

Corollary 7.55. *Given a term $\emptyset \vdash M : \tau$ we have*

$$\mathsf{isNF}\ \underline{[\![\tau]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \mathsf{true} \iff M \textit{ is in normal form}$$

Proof. □

7.4.3 Term Type Counter

Now for something a bit more difficult, instead of checking we could count the types of the all the subterms we have for a closed term. So for example we could define a function $f : \Lambda \to \mathbb{N}$.

$$f(M) := \begin{cases} 2 & \text{if } M = x \\ 3 \cdot f(N) & \text{if } M = \lambda x : \sigma \, . \, N \\ 5 \cdot f(N_1) \cdot f(N_2) & \text{if } M = N_1 \; N_2 \\ 7 \cdot f(N) & \text{if } M = \lambda \alpha : \sigma \, . \, N \\ 11 \cdot f(N) & \text{if } M = N \; \sigma \end{cases}$$

Then f would count types of subterms using a Gödel pairing function.

Now we would like to define a term count such that count $\underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \overline{f(M)}$. And again we would to make use of foldExp. So let us define the four terms.

$$\begin{aligned}
\mathsf{countAbs} : \mathsf{Abs\ KNat} &= \lambda \alpha : * \, . \, \lambda \beta : * \, . \, \lambda f : \mathsf{Nat} \to \mathsf{Nat} \, . \, \mathsf{mult}\ \overline{3}\ (f\ \overline{2}) \\
\mathsf{countApp} : \mathsf{App\ KNat} &= \lambda \alpha : * \, . \, \lambda \beta : * \, . \, \lambda x : \mathsf{Nat} \, . \, \lambda y : \mathsf{Nat} \, . \, \mathsf{mult}\ \overline{5}\ (\mathsf{mult}\ x\ y) \\
\mathsf{countTAbs} : \mathsf{TAbs\ KNat} &= \lambda \alpha : * \, . \, \lambda s : \mathsf{Strip\ KNat}\ \alpha \, . \, \lambda x : \alpha \, . \\
&\quad \mathsf{mult}\ \overline{7}\ (s\ \mathsf{Nat}\ (\lambda \alpha : * \, . \, \lambda y : \mathsf{Nat} \, . \, y)\ x) \\
\mathsf{countTApp} : \mathsf{TApp\ KNat} &= \lambda \alpha : * \, . \, \lambda x : \mathsf{Nat} \, . \, \lambda \beta : * \, . \, \lambda i : \alpha \to \beta \, . \, \mathsf{mult}\ \overline{11}\ x
\end{aligned}$$

Now we define count as follows:

$$\mathsf{count} = \mathsf{foldExp\ KNat\ countAbs\ countApp\ countTAbs\ countTapp}$$

Now to proof that it works, we will first proof a different results, namely we first proof what we will get if we don't require M to be closed and only up to β-equivalence.

However before we do the proof, to make it hopefully clearer we will introduce some functions on the terms.

$$g(M) := \begin{cases} 1 & \text{if } M = x \\ 3 \cdot f(N) & \text{if } M = \lambda x : \sigma \, . \, N \\ 5 \cdot f(N_1) \cdot f(N_2) & \text{if } M = N_1 \; N_2 \\ 7 \cdot f(N) & \text{if } M = \lambda \alpha : \sigma \, . \, N \\ 11 \cdot f(N) & \text{if } M = N \; \sigma \end{cases}$$

$$\begin{aligned}
\#_{\mathrm{b}}(N) &= \text{number of bound variables in N} \\
\mathsf{prodf}_N &= \mathsf{mult}\ x_1\ (\mathsf{mult}\ x_2\ (\ldots\ (\mathsf{mult}\ x_{k-1}\ x_k)\ldots)) \\
&\qquad \text{where } [x_1, \ldots x_k] \text{ are free variables in } N
\end{aligned}$$

Now let $[x_1, \ldots x_k]$ be the list of free variables in the term N, than we define the term prodf_N as follows:

$$\mathsf{prodf}_N := \mathsf{mult}\ x_1\ (\mathsf{mult}\ x_2\ (\ldots\ (\mathsf{mult}\ x_{k-1}\ x_k)\ldots))$$

Lemma 7.56. *Let*

$$S = [F := \mathsf{KNat}, \mathsf{abs} := \mathsf{countAbs}, \mathsf{app} := \mathsf{countTApp}, \\ \mathsf{tabs} := \mathsf{countTAbs}, \mathsf{tapp} := \mathsf{countTApp}]$$

Let M be a term, then we have

$$\ulcorner M : \tau \urcorner_\omega\ S =_\beta \mathsf{mult}\ (\mathsf{mult}\ \overline{g(M)}\ \overline{2^{\#_\flat(M)}})\ \mathsf{prodf}_M$$

Proof.
The proof will be done by induction on the derivative of M.

Induction hypothesis:

$$N \text{ is a subterm in } M \implies \ulcorner N : \sigma \urcorner_\omega\ S =_\beta \mathsf{mult}\ (\mathsf{mult}\ \overline{g(N)}\ \overline{2^{\#_\flat(N)}})\ \mathsf{prodf}_N$$

Base case $M = x$

$$\begin{aligned} \ulcorner x : \tau \urcorner_\omega\ S &= x : \mathsf{Nat} \\ &=_\beta \mathsf{mult}\ (\mathsf{mult}\ \overline{1}\ \overline{1})\ x \\ &= \mathsf{mult}\ (\mathsf{mult}\ \overline{g(x)}\ \overline{2^{\#_\flat(x)}})\ \mathsf{prodf}_x \end{aligned}$$

Case $M = \lambda x : \sigma\ .\ N$

$$\begin{aligned} &\ulcorner \lambda x : \sigma\ .\ N : \sigma \to \tau \urcorner_\omega\ S \\ &=_\beta \mathsf{countAbs}\ \mathsf{Nat}\ \mathsf{Nat}\ (\lambda x : \mathsf{Nat}\ .\ (\ulcorner M : \tau \urcorner_\omega\ S)) \\ &=_\beta \mathsf{mult}\ \overline{3}\ ((\lambda x : \mathsf{Nat}\ .\ (\ulcorner M : \tau \urcorner_\omega\ S))\ \overline{2}) \\ &=_\beta \mathsf{mult}\ \overline{3}\ \mathsf{Nat}\ .\ (\ulcorner M : \tau \urcorner_\omega\ S)[x := 2] \\ &=_\beta \mathsf{mult}\ \overline{3}\ (\mathsf{mult}\ \overline{g(N)}\ \overline{2^{\#_\flat(N)}})\ \mathsf{prodf}_N)[x := 2] && \text{by IH} \\ &=_\beta \mathsf{mult}\ \overline{g(\lambda x : \sigma\ .\ N)}\ \overline{2^{\#_\flat(\lambda x:\sigma\ .\ N)}})\ \mathsf{prodf}_{\lambda x:\sigma\ .\ N} \end{aligned}$$

Case $M = N_1\ N_2$

$$\begin{aligned} &\ulcorner N_1\ N_2 : \tau \urcorner_\omega\ S \\ &=_\beta \mathsf{countApp}\ \mathsf{Nat}\ \mathsf{Nat}\ (\ulcorner N_1 : \sigma \to \tau \urcorner_\omega\ S)\ (\ulcorner N_2 : \sigma \urcorner_\omega\ S) \\ &=_\beta \mathsf{mult}\ \overline{5}\ (\mathsf{mult}\ (\ulcorner N_1 : \sigma \to \tau \urcorner_\omega\ S)\ (\ulcorner N_2 : \sigma \urcorner_\omega\ S)) \\ &=_\beta \mathsf{mult}\ \overline{5}\ (\mathsf{mult}\ (\mathsf{mult}\ (\mathsf{mult}\ \overline{g(N_1)}\ \overline{2^{\#_\flat(N_1)}})\ \mathsf{prodf}_{N_1}) \\ &\qquad (\mathsf{mult}\ \overline{g(N_2)}\ \overline{2^{\#_\flat(N_2)}})\ \mathsf{prodf}_{N_2}) && \text{By IH} \\ &=_\beta \mathsf{mult}\ (\mathsf{mult}\ \overline{g(N_1\ N_2)}\ \overline{2^{\#_\flat(N_1\ N_2)}})\ \mathsf{prodf}_{N_1\ N_2} \end{aligned}$$

Case $M = \lambda\alpha : \kappa\ .\ N$

$$
\begin{aligned}
&\ulcorner \lambda\alpha : \kappa\ .\ N : \Pi\alpha : \kappa.\tau \urcorner_\omega\ S \\
&=_\beta \mathsf{countTAbs}\ [\![\Pi\alpha : \kappa\ .\ \tau]\!]\ \mathcal{S}(F, \kappa, [\![\lambda\alpha : \kappa\ .\ \tau]\!])\ (\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega\ S) \\
&=_\beta \mathsf{mult}\ \overline{7}\ (\mathcal{S}(F, \kappa, [\![\lambda\alpha : \kappa\ .\ \tau]\!])\ \mathsf{Nat}\ (\lambda\alpha : *\ .\ \lambda y : \mathsf{Nat}\ .\ y) \\
&\qquad\qquad (\lambda\alpha : \kappa\ .\ \ulcorner N : \tau \urcorner_\omega)) \\
&=_\beta \mathsf{mult}\ \overline{7}\ (\ulcorner N : \tau \urcorner_\omega\ S)[\alpha := \mathcal{C}(\kappa)] \\
&=_\beta \mathsf{mult}\ \overline{7}\ (\mathsf{mult}\ \overline{g(N)}\ \overline{2^{\#_\flat(N)}})\ \mathsf{prodf}_N) && \text{By IH} \\
&=_\beta \mathsf{mult}\ (\mathsf{mult}\ \overline{g(\lambda\alpha : \kappa\ .\ N)}\ \overline{2^{\#_\flat(\lambda\alpha:\kappa\ .\ N)}})\ \mathsf{prodf}_{\lambda\alpha:\kappa\ .\ N}
\end{aligned}
$$

Case $M = N\ \tau$

$$
\begin{aligned}
&\ulcorner N\ \tau : \tau[\alpha := \sigma] \urcorner_\omega\ S \\
&=_\beta \mathsf{countTApp}\ (\Pi\alpha : \kappa\ .\ \mathsf{Nat})\ (\ulcorner N : (\Pi\alpha : \kappa\ .\ \tau) \urcorner_\omega\ S)\ ([\![\tau[\alpha := \sigma]]\!]\ S) \\
&\qquad (\lambda x : (\Pi\alpha : \kappa\ .\ \mathsf{Nat})\ .\ (x\ [\![\sigma]\!])\ S) \\
&=_\beta \mathsf{mult}\ \overline{11}\ (\ulcorner N : (\Pi\alpha : \kappa\ .\ \tau) \urcorner_\omega\ S) \\
&=_\beta \mathsf{mult}\ \overline{11}\ (\mathsf{mult}\ \overline{g(N)}\ \overline{2^{\#_\flat(N)}})\ \mathsf{prodf}_N) && \text{by IH} \\
&=_\beta \mathsf{mult}\ (\mathsf{mult}\ \overline{g(N\ \sigma)}\ \overline{2^{\#_\flat(N\ \sigma)}})\ \mathsf{prodf}_{N\ \sigma}
\end{aligned}
$$

□

Now that we know what we get from any term, the closed term case follows almost directly by applying substitution.

Corollary 7.57. *Let M be a closed term then we have*

$$\mathsf{count}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \overline{f(M)}$$

Proof.
Let $S = [F := \mathsf{KNat}, \mathsf{abs} := \mathsf{countAbs}, \mathsf{app} := \mathsf{countTApp}, \mathsf{tabs} := \mathsf{countTAbs}, \mathsf{tapp} := \mathsf{countTApp}]$, then we have that

$$\mathsf{count}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \ulcorner M : \tau \urcorner_\omega\ S$$

By the previous lemma and M is closed we have

$$\ulcorner M : \tau \urcorner_\omega\ S =_\beta \mathsf{mult}\ \overline{g(M)}\ \overline{2^m}$$

Note $g(M) \cdot 2^m = f(M)$ and hence $\mathsf{mult}\ \overline{g(M)}\ \overline{2^m} =_\beta \overline{f(M)}$. Since $f(M)$ is a single number and therefore in normal form, we can conclude

$$\mathsf{count}\ \underline{\ulcorner M : \tau \urcorner_\omega} \twoheadrightarrow_\beta \overline{f(M)}$$

□

Instead of counting the term, we can also turn a closed term, ignoring type, into an unique number. First what we do is turn this term into the Bruijn notation. Now, given

an injective pairing function, we can define a function f that turns every closed term in de Bruijn notation into a unique number.

$$f(M) := \begin{cases} \langle 1, n \rangle & \text{if } M = n \\ \langle 2, f(N) \rangle & \text{if } M = \lambda : \sigma \;.\; N \\ \langle 3, \langle f(N_1), f(N_2) \rangle & \text{if } M = N_1 \; N_2 \\ \langle 4, f(N) \rangle & \text{if } M = \lambda\alpha : \sigma \;.\; N \\ \langle 5, f(N) \rangle & \text{if } M = N \; C \end{cases}$$

For the pairing function we will use the Cantor pairing function. We can define this as follows in System F_ω.

Definition 7.58.

$$\mathsf{cntr} = \lambda x : \mathsf{Nat} \;.\; \lambda y : \mathsf{Nat} \;.\; \mathsf{add}\ (\mathsf{div}\ (\mathsf{mult}\ (\mathsf{add}\ x\ y)\ (\mathsf{add}\ (\mathsf{mult}\ x\ y)\ \overline{1}))\ \overline{2})\ y$$

Defining the arithmetization term is quite straight forward, except for the variable cases. This is since the term $\lambda x : \sigma \;.\; \lambda y : \sigma \;.\; y$ should result in a different number than that of the term $\lambda x : \sigma \;.\; \lambda y : \sigma \;.\; x$. Specifically, since we are using de Bruijn notation, y is represented by the number 1 and x is represented by the number 2. To do this we will use a similar trick as with the term isNF. We will, instead of returning a singular number, return a pair of numbers, where the first number will represent the arithmetization and second number will represent a counter for the number of abstractions.

To define the term, we will introduce the following terms $\mathsf{pairn} := \mathsf{pair\ Nat}$, $\mathsf{fstn} := \mathsf{fst\ Nat}$ and $\mathsf{sndn} := \mathsf{sndNat}$ and the constructors $\mathsf{Nats} := \mathsf{Pair\ Nat}$ and $\mathsf{KNats} := \lambda\alpha : * \;.\; \mathsf{Nats}$.

Definition 7.59.

$$\begin{aligned}
\mathsf{arthabs} : \mathsf{Abs\ KNats} \quad &= \lambda\alpha : * \;.\; \lambda\beta : * \;.\; \lambda f : \mathsf{Nats} \to \mathsf{Nats} \;. \\
&\quad (\lambda c : \mathsf{Nat} \;.\; (\lambda b : \mathsf{Nat} \;.\; \mathsf{pairn}\ (\mathsf{cntr}\ \overline{2}\ b)\ (\mathsf{suc}\ c)) \\
&\quad (\mathsf{fstn}\ (f\ (\mathsf{pair}\ 0\ c)))) \ (\mathsf{sndn}\ (f\ (\mathsf{pairn}\ \overline{0}\ \overline{0}))) \\
\mathsf{arthapp} : \mathsf{App\ KNats} \quad &= \lambda\alpha : * \;.\; \lambda\beta : * \;.\; \lambda x : \mathsf{Nats} \;.\; \lambda y : \mathsf{Nats} \;. \\
&\quad : \mathsf{Nat} \;.\; \mathsf{pairn}\ (\mathsf{cntr}\ \overline{3}\ (\mathsf{cntr}\ (\mathsf{fstn}\ x)\ (\mathsf{fstn}\ y))) \\
&\quad (\mathsf{max}\ (\mathsf{sndn}\ x)\ (\mathsf{sndn}\ y)) \\
\mathsf{arthtabs} : \mathsf{TAbs\ KNats} \quad &= \lambda\alpha : * \;.\; \lambda s : \mathsf{Strip\ KNats}\ \alpha \;.\; \lambda x : \alpha \;. \\
&\quad \mathsf{pairn}\ (\mathsf{cntr}\ \overline{4}\ (s\ \mathsf{Nats}\ (\lambda\alpha : * \;.\; \lambda y : \mathsf{Nats} \;.\; y) \\
&\quad (\mathsf{fstn}\ (s\ x)))) \ (\mathsf{sndn}\ (s\ x)) \\
\mathsf{arthtapp} : \mathsf{TApp\ KNats} \quad &= \lambda\alpha : * \;.\; \lambda x : \mathsf{Nats} \;.\; \lambda\beta : * \;.\; \lambda i : \alpha \to \beta \;. \\
&\quad \mathsf{pairn}\ (\mathsf{cntr}\ \overline{5}\ (\mathsf{fstn}\ x))\ (\mathsf{sndn}\ x)
\end{aligned}$$

$$\mathsf{arth} = \mathsf{foldExp\ KNats\ arthAbs\ arthApp\ arthTAbs\ arthTapp}$$

Unfortunatly the proof proved to be out of scope for this thesis.

7.5 Restrictions

While the self-interpreter defined by Brown and Palsberg seems strong, there are some restriction that it has, which the Mogensen and Barendregt self-interpreter do not have. We will present two of these restrictions, the first one being that we can't turn a encoded term into the normal form of the encoded term and the second the limitations between comparing two encoded terms of different types.

7.5.0.1 Type Recognition

Now before we start, it is good to remind ourselves that we can't compare constructors in System F_ω. This follows from that constructors are not terms. We can easily prove that such a term does not exists.

Corollary 7.60. *Let C_1, C_2 be some constructors of kind κ with $C_1 \neq_\beta C_2$. Then there does not exists a term E such that $E\ C_1\ C_2 =_\beta \mathsf{false}$ and $E\ C_1\ C_1 =_\beta \mathsf{true}$.*

Proof. Assume such a term E exists and $C_1 \neq_\beta C_2$. Now note that E is of type $\Pi\alpha : \kappa\ .\ \Pi\beta : \kappa\ .\ \mathsf{Bool}$, which means that both α and β are redundant quantifiers. We also know that there is some term M such that $E =_\alpha \lambda\alpha : \kappa\ .\ \lambda\beta : \kappa\ .\ M$. By Lemma 7.28 we therefore have the following contradiction

$$\mathsf{true} =_\beta E\ C_1\ C_1 =_\beta M =_\beta E\ C_1\ C_2 =_\beta \mathsf{false}$$

□

For the Brown-Palsberg self-interpreter we have something similar. Given two terms of different types, we don't have a term that can check if they have different types. This is the same as proven there is no can't check α-equivalence between two closed terms. This is since we already have proven that we can compare structure between two encoded terms. And we have α-equivalence between two closed terms if and only if both the structure and type is the same.

Now to proof that α-equivalence we define the term idCode. This term will be used to turn a encoded type into a encoded term for the identity term of this type.

Definition 7.61.

$$\begin{aligned}
&\mathsf{idCode} : \Pi\alpha : (* \to *) \to *\ .\ \mathsf{Exp}\ \alpha\\
&\mathsf{idCode} := \lambda\alpha : (* \to *) \to *\ .\ \lambda F : * \to *\\
&\qquad \lambda\mathsf{abs} : \mathsf{Abs\ F}\ .\\
&\qquad \lambda\mathsf{app} : \mathsf{App\ F}\ .\\
&\qquad \lambda\mathsf{tabs} : \mathsf{TAbs\ F}\ .\\
&\qquad \lambda\mathsf{tapp} : \mathsf{TApp\ F}\ .\ \mathsf{abs}\ (\alpha\ F)\ (\alpha\ F)\ (\lambda x : F\ (\alpha\ F)\ .\ x)
\end{aligned}$$

Now note that if we input an encoded type, for example σ, then we get from the term idCode the encoded identity function of σ, i.e. we have $\mathsf{idCode}\ \underline{[\![\sigma]\!]} \twoheadrightarrow_\beta \underline{\ulcorner \lambda x : \sigma\ .\ x : \sigma \to \sigma \urcorner_\omega}$. Now with this construction we can prove that a term that checks α-equivalence is impossible.

Lemma 7.62. *There does not exists a polymorphic term E such that*

$$E\ \underline{[\![\tau]\!]}\ \underline{[\![\sigma]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega}\ \underline{\ulcorner N : \sigma \urcorner_\omega} =_\beta \mathsf{true} \iff M =_\alpha N$$

Proof. Assume, towards contradiction, that such a E term exists. Now fix some type τ and σ such that $\tau \neq_\beta \sigma$ and then define the term E_2 as follows:

$$E_2 := \lambda\alpha : (* \to *) \to * \ .\ \lambda\beta : (* \to *) \to * \ .\ E\ \alpha\ \beta\ (\mathsf{idCode}\ \alpha)\ (\mathsf{idCode}\ \beta)$$

Then we have $E_2\ \underline{[\![\tau]\!]}\ \underline{[\![\tau]\!]} \neq_\beta E_2\ \underline{[\![\tau]\!]}\ \underline{[\![\sigma]\!]}$ which contradicts Corollary 7.60 □

From this we can directly prove that there does not exists a term that can check if two terms have different types.

Corollary 7.63. *There does not exists a polymorphic term E such that*

$$E\ \underline{[\![\tau]\!]}\ \underline{[\![\sigma]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega}\ \underline{\ulcorner N : \sigma \urcorner_\omega} =_\beta \mathsf{true} \iff M =_\beta N$$

What is more, from this prove it also follows there is not term that can check β-equivalence. This follows since the identity term is in normal form, and therefore two identity terms from idCode are beta equivalent if and only if they are of the same type.

Corollary 7.64. *There does not exists a polymorphic term E such that*

$$E\ \underline{[\![\tau]\!]}\ \underline{[\![\sigma]\!]}\ \underline{\ulcorner M : \tau \urcorner_\omega}\ \underline{\ulcorner N : \sigma \urcorner_\omega} =_\beta \mathsf{true} \iff M =_\beta N$$

8 Discussion

(Trivial) Self-Interpreters - This document has been about self-interpreters and what they support. However in the end we could not conclude with a good definition for a self-interpreter. To quote Bauer: "In order to shed further light on self-interpreters for total languages we need a definition of self-interpreters which takes into account structural properties of self-interpreters that distinguishes the interpreter by Brown and Palsberg from the one given in Theorem 3.2 [trivial self-recognizer]."
It is still an open question what a good definition is, however we believe that a representation supporting a function definition scheme will be an important part of the puzzle.

Self-Evaluators - We have shown in Section 6.4 that self-evaluation is not a hard problem to solve for strongly normalizing calculi. However we understand self-evaluation from not strongly normalizing calculi to be a hard problem, as discussed in Section 4.0.3. The gap between the difficulty of these problems is striking and can be explained by the short cut you can take when working in a strongly normalizing calculi. Removing the short cut it is unclear if a self-evaluator is possible at all, in fact it is likely that it is impossible at all since in the self-evaluators shown we required a fixed-point operator. It may be possible to proof that any representation with a term `app`, such that `app` $\ulcorner M \urcorner \ulcorner N \urcorner =_\beta \ulcorner M\ N \urcorner$, does not have a self-evaluator by making use of Bauer's [9] argument in Theorem 2.5 in combination with the trivial self-evaluators. This leads us to the following conjecture

Conjecture 8.1. *Any self-evaluator for a strongly normalizing calculus is a form of a trivial self-evaluator.*

Function Definition Schemes - A central point of this document regarding self-interpreter was if they had function definition schemes. We have seen that these function definition schemes are powerful and that assuming some simple terms we could prove that a representation would also form a self-recognizer or self-evaluator. There seems to be a correlation between what the structural properties is of a self-interpreter and the function definition scheme. I would be interesting to investigate how strong this relation is and especially if there are things we can prove that we can not do if we cannot define a function definition scheme.

Performance - While in this document we have looked at self-interpreters from a theoretical point of view, they are mostly looked at a viewpoint of a concrete program with performance metrics. When thinking about self-interpreters we think about them as they are implemented in modern lanuages. This makes performance also often a consideration for the lambda calculus version. For example Mogensen presented the Mogensen representation since "The size of the representations of the λ-terms using this [Barendregt's] schema grows (at least) exponentially in the size of the terms. Operations on this [Barendregt's] representation are also extremely expensive"[25] and with that came the first self-recognizer for untyped lambda calculus that was not weak. For Brown and Palsberg this was also an consideration and implemented their self-recognizer in Haskell [10]. So therefore it will also be important how these structural properties of self-interpreters links to performance metrics like size of the representation and expensiveness of operations.

9 Related Work

For more related work we refer the reader to both of Brown and Palsberg papers [10][11], as they have an excellent piece which this section is based upon.

Typed Self-Interpretation
There are plenty of self-recognizers for typed calculi in the literature. The first typed self-recognizer was shown by Rendel, et al. [31]. They showed it for an extension of System F_ω, which is not strongly-normalizing. With their representation also came a size operation, proving a stronger representation than the trivial self-recognizers we have seen. Jay and Palsberg presented a typed self-recognizer for a combinator calculus, which was the first self-recognizer for a language with decidable type checking [20]. We have seen that Brown and Palsberg presented the first self-recognizer for a strongly-normalizing calculus and they have also presented the first self-evaluator, which was for a not strongly-normalizing extension of System F_ω [11].

Dependent Types
In this document we have taken a look at System F_ω for self-interpretation. We have seen that System F_ω includes two of the three properties of Barendregt's lambda cube, polymorphism and type operators. The last dependency we need to get to the calculus of constructions is dependent types. Dependent types are also used in proof assistants, as most are based on the calculus of constructions. It is possible to make use of dependent types to ensure that only well-typed terms are represented. For example Shürmann et al. represented System F_ω in LF [34]. Chapman presented a meta-circular representation of a dependent type theory in Agda [12]. These representations are especially useful for machine-checked proofs of the meta-theorems for the represented language.
Altenkirch and Kaposi formalize a simple dependent type theory in another type theory [1]. A key problem of defining a typed representation of a depended typed representation is that types, terms, type context and type equality are all dependent on each other. Their solution relies on making use of Quotient-Inductive Types. It is not full self-representation, since QIT was not represented and the authors cite it as an open challenge.

Typed Meta-Programming
What we have looked at in this document, typed self-interpretation, is a form of typed meta-programming. With typed meta-programming we have a type representation of one language in possible another language. That this can be useful is shown by Chen and Xi, who demonstrated that types can make meta-programming less error-prone [13].

10 Conclusion

We have investigated what a self-interpreter means in different types of lambda calculus and especially if it is possible in strongly normalizing lambda calculi. To do this we first found out what was meant with self-interpretation in a more general sense. Using Figure 2 and some diagram chasing we concluded that we can distinguish between two types of self-interpreters; a self-recognizer and a self-evaluator.

To explore what it is that lets self-interpreters have good structural properties, we looked at two different untyped lambda calculus self-interpreters that have strong structural properties, one define by Mogensen and one defined by Barendregt. We compared these to novel trivial self-recognizer inspired by the one given by Brown and Palsberg. Making use of the function definition schemes we could prove structural properties for the Mogensen and Barendregt representations, whereas we found that the trivial self-recognizer did not support a function definition scheme.

From there we use our obtained knowledge to investigate self-interpreters for strongly normalizing typed lambda calculi. However there is a problem, it is common knowledge that this is impossible, the so called Normalization Barrier Conjecture. To defeat this barrier we first needed to know what it meant. In the analysis of the conjecture we find that Brown and Palsberg has broken this barrier down and we answered an open question of them, as we find that a self-evaluators are possible in a strongly normalizing calculus.

We end with an investigation of the Brown-Palsberg self-interpreter itself and confirm some of the strong structural properties it has.

The result is a comprehensive study of self-interpreters which includes a novel way to investigate self-interpreters and generalisations of previous results. It can be used in further research to answer open questions like:

- What is a good definition for a self-interpreter such that it has strong structural properties?
- Are all self-evaluators for strongly normalizing languages trivial?

And with that we conclude that while the normalization barrier may be death, we may end up saying long live the normalization barrier.

Aknowledgements
I would like to thank my father Dick Verkleij for proof-reading english in record times with strict deadlines without complaining. I would like to thank Freek Wiedijk for accepting to be the second reader. And I would like to thank Herman Geuvers for having the patients for supervising a difficult student like me.

References

[1] Thorsten Altenkirch and Ambrus Kaposi. "Type Theory in Type Theory Using Quotient Inductive Types". In: *SIGPLAN Not.* 51.1 (Jan. 2016), pp. 18–29. ISSN: 0362-1340. DOI: 10.1145/2914770.2837638. URL: https://doi.org/10.1145/2914770.2837638.

[2] H. P. Barendregt. "Functional programming and lambda calculus". In: *J. van Leeuwen (ed.), Handbook of Theoretical Computer Science* B (1990), pp. 323–363. URL: http://hdl.handle.net/2066/17230.

[3] H. P. Barendregt. *Lambda calculi with types*. Onbepaald. 1992. URL: http://hdl.handle.net/2066/17231.

[4] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of functional programming* 1.2 (1991), pp. 125–154.

[5] Henk Barendregt. "Reflection: a powerfull and ubiquitous logical mechanism". In: (2007).

[6] Henk Barendregt and Erik Barendsen. "Introduction to lambda calculus". In: (2000).

[7] Henk P Barendregt. "Self-interpretation in lambda calculus". In: (1991).

[8] Henk P Barendregt et al. *The Lambda Calculus: its syntax and semantics, volume 103 of Studies in Logic*. North Holland, 1984.

[9] Andrej Bauer. *On Self-Interpreters for System T And Other Typed λ-Calculi*. 2016. URL: http://math.andrej.com/wp-content/uploads/2016/01/self-interpreter-for-T.pdf.

[10] Matt Brown and Jens Palsberg. "Breaking through the normalization barrier: a self-interpreter for f-omega". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 5–17.

[11] Matt Brown and Jens Palsberg. "Typed self-evaluation via intensional type functions". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 415–428.

[12] James Chapman. "Type Theory Should Eat Itself". In: *Electr. Notes Theor. Comput. Sci.* 228 (Jan. 2009), pp. 21–36. DOI: 10.1016/j.entcs.2008.12.114.

[13] Chiyan Chen and Hongwei Xi. "Meta-Programming through Typeful Code Representation". In: *SIGPLAN Not.* 38.9 (Aug. 2003), pp. 275–286. ISSN: 0362-1340. DOI: 10.1145/944746.944730. URL: https://doi.org/10.1145/944746.944730.

[14] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363. ISSN: 00029327, 10806377. URL: http://www.jstor.org/stable/2371045.

[15] Alonzo Church and J. B. Rosser. "Some Properties of Conversion". In: *Transactions of the American Mathematical Society* 39.3 (1936), pp. 472–482. ISSN: 00029947. URL: http://www.jstor.org/stable/1989762.

[16] Nicolaas Govert De Bruijn. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem". In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5. Elsevier. 1972, pp. 381–392.

[17] Herman Geuvers. "A short and flexible proof of strong normalization for the calculus of constructions". In: *International Workshop on Types for Proofs and Programs*. Springer. 1994, pp. 14–38.

[18] Herman Geuvers. "Self-interpretation in Lambda Calculus". 2015. URL: http://www.cs.ru.nl/~herman/onderwijs/%202015Reflection/lecture6.pdf.

[19] Dave Herman. *Narcissus.* URL: https://wiki.mozilla.org/Narcissus (visited on 08/24/2021).

[20] Barry Jay and Jens Palsberg. "Typed Self-Interpretation by Pattern Matching". In: *SIGPLAN Not.* 46.9 (Sept. 2011), pp. 247–258. ISSN: 0362-1340. DOI: 10.1145/2034574.2034808. URL: https://doi.org/10.1145/2034574.2034808.

[21] S. C. Kleene. "λ-definability and recursiveness". In: *Duke Mathematical Journal* 2.2 (1936), pp. 340–353. DOI: 10.1215/S0012-7094-36-00227-2. URL: https://doi.org/10.1215/S0012-7094-36-00227-2.

[22] Stephen Cole Kleene et al. *Introduction to metamathematics.* Vol. 483. van Nostrand New York, 1952.

[23] Ben Lynn. Apr. 2017. URL: https://benlynn.blogspot.com/2017/04/much-time-has-passed-since-my-last-entry.html (visited on 05/31/2021).

[24] *Meta-circular evaluator.* May 2021. URL: https://en.wikipedia.org/wiki/Meta-circular_evaluator (visited on 08/21/2021).

[25] Torben Mogensen. "Efficient Self-Interpretation in Lambda Calculus". In: *Journal of Functional Programming* 2 (Oct. 1994). DOI: 10.1017/S0956796800000423.

[26] Matthew Naylor. "Evaluating Haskell in Haskell". In: *The Monad.Reader* 10 (2008), pp. 25–32.

[27] *Normalization property (abstract rewriting).* Jan. 2021. URL: https://en.wikipedia.org/wiki/Normalization_property_(abstract_rewriting) (visited on 05/31/2021).

[28] Christine Paulin-Mohring. *Introduction to the calculus of inductive constructions.* 2015.

[29] Frank Pfenning and Peter Lee. "Metacircularity in the polymorphic λ-calculus". In: *Theoretical Computer Science* 89.1 (1991), pp. 137–159.

[30] Tillmann Rendel, Klaus Ostermann, and Christian Hofer. "Typed Self-Representation". In: *SIGPLAN Not.* 44.6 (June 2009), pp. 293–303. ISSN: 0362-1340. DOI: 10.1145/1543135.1542509. URL: https://doi-org.ru.idm.oclc.org/10.1145/1543135.1542509.

[31] Tillmann Rendel, Klaus Ostermann, and Christian Hofer. "Typed Self-Representation". In: vol. 44. May 2009, pp. 293–303. DOI: 10.1145/1543135.1542509.

[32] Armin Rigo and Samuele Pedroni. "PyPy's approach to virtual machine construction". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications.* 2006, pp. 944–953.

[33] Raúl Rojas. "A tutorial introduction to the lambda calculus". In: *arXiv preprint arXiv:1503.09060* (2015).

[34] Carsten Schürmann, Dachuan Yu, and Zhaozhong Ni. "A Representation of F_ω in LF". In: *Electronic Notes in Theoretical Computer Science - ENTCS* 58 (Nov. 2001), pp. 79–96. DOI: 10.1016/S1571-0661(04)00280-4.

[35] Morten Heine Sørensen. "Strong Normalization from Weak Normalization in Typedλ-Calculi". In: *Information and Computation* 133.1 (1997), pp. 35–71.

[36] Tom Stuart. *Understanding Computation: Impossible Code and the Meaning of Programs.* O'Reilly, 2013.

[37] William W Tait. "Intensional interpretations of functionals of finite type I". In: *The journal of symbolic logic* 32.2 (1967), pp. 198–212.

[38] David A Turner. “Total Functional Programming.” In: *J. Univers. Comput. Sci.* 10.7 (2004), pp. 751–768.

[39] Philip Wadler. “Theorems for free!” In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, pp. 347–359.

[40] Mateusz Zakrzewski. “Definable functions in the simply typed lambda-calculus”. In: *CoRR* abs/cs/0701022 (2007). arXiv: `cs/0701022`. URL: `http://arxiv.org/abs/cs/0701022`.