Closure Calculus is Better than the Pure λ-Calculus

Xuanyi Chew

29 September 2019

Contents

Contents **1**

I Introduction and Preliminaries 4

1 Introduction **5**

1.1 Of Lambda Calculi, Closure Calculus and The Pure λ-Calculus . 5
1.2 "Better" Means Simpler and Faster 8
1.3 Novel Contributions 9
1.4 Significance of the Objectives to the Aims 9
1.5 Methods 12
1.6 Tests For Success 13
1.7 Structure of This Report 13
1.8 Notation Conventions 14

2 On Abstract Reduction Systems, Term Rewriting Systems and Various Preliminaries **16**

2.1 Terms 20
2.2 Substitution 22
2.3 Rewrite Rules 22
2.4 Normal Forms 23
2.5 Confluence and Termination 23
2.6 Evaluation and Operational Semantics 24
2.7 Some Remarks on ARSs and TRSs 25

3 The Pure λ-Calculus **26**

3.1 Terms 27
3.2 Substitution 28
3.3 Rewrite Rules 29
3.4 Normal Forms 31
3.5 Confluence and Termination 32
3.6 Evaluation and Operational Semantics 34
3.7 Some Remarks on the Pure λ-Calculus 37

4 Closure Calculus **38**
4.1 Terms 39
4.2 Substitution 42
4.3 Rewrite Rules 43
4.4 Normal Forms 44
4.5 Confluence and Termination 45
4.6 Evaluation and Operational Semantics 47
4.7 Some Remarks on Closure Calculus 48

II Implementation and Comparisons **50**

5 On Implementations and Fair Comparisons **51**
5.1 Why Compare Reductions to Normal Forms? 51
5.2 On the Implementation of Operational Semantics in General . . 53
5.3 On the Implementation of the Lambda Calculi 54
5.4 δ-Terms and δ-Rules 61
5.5 The Test and Benchmark Programs 62

6 Closure Calculus is Simpler **65**
6.1 On the Simplicity of the Lambda Calculi 65
6.2 Complexity Metrics 67
6.3 Closure Calculus Is Simpler To Implement 67

7 Closure Calculus Terms Reduce to Normal Forms Faster **70**
7.1 Performance Metrics 70
7.2 Closure Calculus Has Fewer Meta Operations 71
7.3 Closure Calculus is Faster; Uses Less Memory 73
7.4 Further Analysis 75

III Implications **78**

8 Interesting Artefacts of Closure Calculus **79**
8.1 Natural Numbers 79
8.2 List and Pair 83
8.3 On The Church-Encoded *pred* 84
8.4 Evaluation Functions With Local Variables 86
8.5 Extracting the Body of an Abstraction: 88
8.6 Concerning Identity 88

9 Discussions **90**
9.1 What Does This Mean For Programming Language Theory . . . 90
9.2 Three Potential Objections 91
9.3 A Survey of Compilation Methodologies for Popular Programming languages 91
9.4 Answering the Objections 92

9.5 On The Oddity of $x \mapsto x$ 94

10 Conclusion **97**

Bibliography **98**

Part I

Introduction and Preliminaries

Chapter 1

Introduction

Closure Calculus (Jay 2018) - a recently introduced lambda calculus - is both simpler and faster than the pure λ-calculus. This is attributable to the fact that Closure Calculus has no meta-theory requirement. This leads to simpler implementations and consequently, faster reduction to normal forms. In this regard, it is better than the pure λ-calculus.

Reader Note: Are you able to tell that the first sentence is B.J's "That..." thesis statement in disguise?

This has some implications for programming language design, which this report alludes to. The majority of this report is focused on the comparisons between Closure Calculus and the pure λ-calculus across a number of metrics. However, that Closure Calculus being better than the pure λ-calculus has a practical implication, thus the design of the comparisons were motivated by the potential implications.

The concepts (lambda calculi, reduction, etc.) and notions (simpler and faster) will be given more formal treatment and careful considerations in the chapters to come. For now it is sufficient to give an informal introduction as follows.

1.1 Of Lambda Calculi, Closure Calculus and The Pure λ-Calculus

A lambda calculus is a formal system for expressing computations. They were first introduced by Church (1936) to investigate the foundations of mathematics in form of the questions first posed by Hilbert. Today, there are many lambda calculi of which Closure Calculus and the pure λ-Calculus are instances.

KW: Lambda Calculi

The comparison of Closure Calculus and the pure λ-Calculus is the main objective of this report. The linchpin of the comparisons is that of reduction and normal forms. A term in either lambda calculus may be reduced to a simpler term. A normal form of a lambda calculus is one where the terms may not reduce any further. These explanations are brief here - Chapter 2 explores the notions of reductions and normal forms in a more formal manner, as well as gives a place to lambda calculi in a map of computational systems. Chapter 5

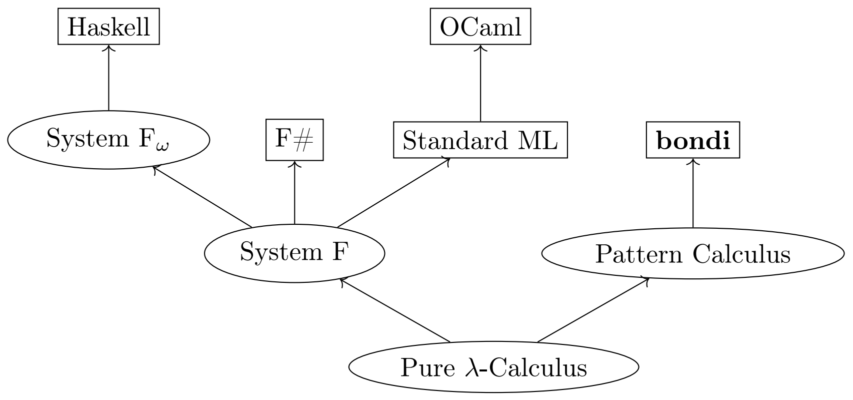


Figure 1.1.1: Family tree of some functional programming languages

explains why it it is only fair to compare the calculi by reductions of terms to their normal forms.

Various lambda calculi are used as the theoretical basis for many functional programming languages. Figure 1.1.1 shows a family tree of functional programming languages (squares) based on their dependence on a theoretical basis (ellipses). While this report mainly deals with untyped lambda calculi, here we briefly acknowledge the importance of typed lambda calculi. Many functional programming languages use a typed lambda calculus as their theoretical basis.

Typed lambda calculi are extensions of the pure λ-calculus. Indeed, there can be many kinds of typed lambda calculi, each depending on a more primitive variant. Figure 1.1.2 shows Barendregt's framework for discussing typed lambda calculi. The arrows indicate a dependence on objects of a previous typed lambda calculus. For example, the $\lambda 2$ (also known as System F) depends on the simply-typed λ-calculus, $\lambda \rightarrow$.

KW: Closure Calculus

Closure Calculus began life as an exercise to re-state the pure λ-calculus without requiring meta-theory. The result is a lambda calculus that has all the desirable qualities of the pure λ-calculus without the requisite meta-theory. It is a good theory of functions, Turing-complete, and is amenable to equational reasoning. Its properties of progress and preservation are proven. Closure Calculus draws from earlier works on explicit substitution (Abadi et al. 1990) as well as work on closure conversion (Reynolds 1972; Steele 1978; Appel 2007). A more complete account of Closure Calculus is given in Chapter 4.

In this report, we introduce two flavours of Closure Calculus - Closure Calculus with Names (CCN) and Closure Calculus without names (CCDB), which is inspired by de Bruijn indices. This report shows that both variants Closure Calculus is better than the pure λ-calculus.

KW: Pure λ

The pure λ-Calculus is the lambda calculus as introduced by Church in 1936. As introduced, there are three main rules, dubbed α, β and η. Now there are many lenses through which one may consider these rules. For example, from a term rewriting point of view, the α rule is considered to be a rewrite rule. The

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda\Pi\omega \\
\lambda 2 \arrow[rr] \arrow[ur] & & \lambda\Pi 2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[rr] \arrow[uu] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[rr] \arrow[uu] \arrow[ur] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 1.1.2: The λ-Cube

same rule is not considered to be a reduction rule when viewed from the lens of programming languages. In this report, only the β-reduction rule is considered as a reduction rule, although a formal treatment of all the rules will be given in Chapter 3.

KW: Meta-theory

In the pure λ-calculus, reduction is done by means of substitution. Despite being vital to the notion of reduction, almost no formal treatment is given to substitutions in the original introduction of the pure λ-calculus. Subsequent works have attempted to formalise the notion of substitution. Thus explicit substitutions (Abadi et al. 1990; Archambault-Bouffard and Monnier ????; Munoz 1996) were introduced. Along the way there has been difficulties - for example Mellies (1995) observed that a typed lambda calculus with explicit substitution may not terminate.

Another vital meta-theory is that of α-equivalence (Thompson 1991, pp. 34). α-equivalence makes the pure λ-calculus complicated but is usually glossed over. Two expressions are considered α-equivalent if they are equivalent after all the bound variables have been renamed to match each other. It's been shown that the optimal algorithm to determine α-equivalence is quadratic(Morazán and Schultz 2008). The requirement for having a meta-theory on substitution and α-equivalence is also why λ-calculus is only considered a confluent term rewriting system up to α-equivalence. Attempts to overcome the meta-theory requirement, such as with de Bruijn indices, or high order abstract syntaxes only serves to introduce new meta-theory or complicate any implementation.

1.2 "Better" Means Simpler and Faster

The major claim of this report is that Closure Calculus is better than the pure λ-calculus. The word "better" is defined in two ways - Closure Calculus is simpler than the pure λ-calculus; and programs written in Closure Calculus reduce to normal forms faster than a similar program written in the pure λ-calculus. This is often shortened to the utterance "Closure Calculus is simpler and faster than the pure λ-calculus", or simply "Closure calculus is better than the pure λ-calculus" in this work.

KW: Simpler

Closure Calculus is simpler than the pure λ-calculus in two sense of the word. First, it is simpler to implement than the pure λ-calculus. Second, it is simpler in that the reduction rules are straightforwards. The first sense of the word "simple" implies that a comparison is done on the implementations. The second sense of the word implies a comparison is done on the theories.

When comparing implementations, simplicity is measured in a variety of metrics, from lines of codes to number of logical branches required. A Closure Calculus interpreter is simpler than a pure λ-calculus interpreter across these metrics.

When comparing the theory of Closure Calculus and the theory of the pure λ-calculus, it gets a little trickier. On the surface, it appears that the reduction rules of Closure Calculus is not axiomatic. Upon first read of the reduction rules of Closure Calculus, it will not be obvious to the reader on how reductions would work. However, this is not because Closure Calculus is more complicated. Rather, it is that the pure λ-calculus requires some hidden assumptions. This is commonly known as meta-theory. The lack of meta-theory is why Closure Calculus is considered to be simpler.

To put it plainly, the reduction rules of Closure Calculus are straightforwards - what you see is what you get; while the reduction rules of the pure λ-calculus requires more contextual information. This has implications when implementing the calculi in question. These implications may be extended to the case of programming languages, many of which are at least partial implementations of the calculi.

Further expositions on the relative simplicity of Closure Calculus to that of the pure λ-calculus can be found in Chapter 6.

KW: Faster

The reduction rules of Closure Calculus being simpler reveals its implication in the performance of the respective implementations of the lambda calculi. Programs written in Closure Calculus reduce to normal forms faster than a similar program written in the pure λ-calculus. This is measured in three ways - count of operations; absolute time taken; and memory required to reduce the terms to normal form. In all three metrics, Closure Calculus outperforms the pure λ-calculus across a number of tasks. This result is attributable to the fact that there is little-to-no meta-theory requirement in Closure Calculus. Further analysis is given in Chapter 7.

The count of operations includes counting the application of reduction rules and meta-operations. Readers may protest that it is not fair to include meta-operations in the count of operation. However, this comparison is fair when viewed with the lens of implementing programming languages. Both reduction operations and meta-operations use computational resources. The rationale for comparing reductions to normal forms (as opposed to comparing reduction to head normal forms) is given in Chapter 5. However, the brief explanation is that the comparison of reductions to normal forms is the only fair way to compare the performance of the calculi.

1.3 Novel Contributions

The main novel contribution of this work is to show that

1. The implementations of Closure Calculus are simpler than equivalent implementations of the pure λ-calculus;
2. Programs written in Closure Calculus reduces to normal forms faster than the equivalent programs in the pure λ-calculus.

This is done in service of the view that Closure Calculus is more suitable as a foundation for functional programming languages than pure λ-calculus.

Table 1.2 shows a summary of the extent that this work is novel.

Approach	Simple	Simpler (Fewer Lines of Code)	Simpler (Fewer Meta-Operations vs Baselines)	Faster than Baselines
Normal Order λ-calculus (Naïve) - Baseline 1	✓	✗	✗	✗
Call-by-Value λ-calculus (Naïve) - Baseline 2	✓	✗	✗	✗
Call-by-Name λ-calculus (Naïve) - Baseline 3	✓	✗	✗	✗
This Work (CCN)	✓	✓	✓	✓
This Work (CCDB)	✓	✓	✓	✓

Table 1.2: Summary of Contributions

1.4 Significance of the Objectives to the Aims

Stakeholders

The primary audience for this report are functional programming language implementors whose aims are given in the following subsection. Now this report is about the comparisons of Closure Calculus to the pure λ-calculus. The metrics of comparison are based on reductions to normal forms. This may not interest some implementors of functional programming languages, as reduction to normal form is not the priority.

Nonetheless, there is value in understanding implementations of calculi. The implications of this report transfer well to implementations of programming languages. This work shows what it means to have an implementation of a calculus with few or no side conditions. If such an implementation is simpler than an implementation of the meta-theory heavy pure λ-calculus, it follows that an implementation of a programming language based on Closure Calculus would also be simpler. If the programs run faster on an implementation of Closure Calculus than on a similarly implemented pure λ-calculus, then by extension, programs would presumably also run faster on a programming language built on top of Closure Calculus.

This report shows that there may be utility in using Closure Calculus as a theoretical basis for a functional programming language. This may be an interesting topic for future explorations.

Reader note: By this point, readers should be able to tell that people like Simon Peyton Jones, and Xavier Leroy are the target audience of this report. Y/N ?

Aims

This report is written for functional programming language implementors who have concerns over the following issues:

1. Having a simpler-to-implement compilation and runtime components
2. Having faster programs at runtime.

All things being equal, a programming language that is simpler to implement is preferred over a programming language that is complex to implement. The notion of "implementation of programming language" is intentionally left vague as the primary comparisons done in this report concerns the actual lambda calculi themselves. It suffices to say that an implementation of a programming language involves both compile time and runtime components of a programming language.

Furthermore, highly performant programs are always in demand - implementors of programming languages are always adding various optimisations passes in compilers so that the resulting programs run faster.

Given that functional programming languages are built on a theoretical foundations of a lambda calculus, it is fair to say that the aim of functional programming language implementors is to find a better lambda calculus to form the theoretical basis of functional programming languages.

Objectives

The objective of this report is to show that Closure Calculus is better than the pure λ-calculus. Closure Calculus is better in two ways - it is simpler and faster. Simplicity is compared in two ways: comparison of the theory and comparison of the implementations. The latter is done by implementing interpreters for Closure Calculus and the Pure λ-Calculus, and comparing the simplicity of each interpreter. An additional hypothesis naturally occurs: is Closure Calculus faster than the pure λ-calculus? To answer that, equivalent programs are written in both Closure Calculus and the pure λ-calculus and the reductions to their normal forms are compared.

There are also side benefits to comparing the interpreters. First, an implementation of a Closure Calculus interpreter shows that Closure Calculus is practical, and not just another Turing Tarpit, where "...everything is possible but nothing of interest is easy" (Perlis 1982).

Furthermore an implementation of Closure Calculus serves as a tool to explore some implications of the theory. In the implementing an interpreter for Closure Calculus, potential issues around the calculus may be discovered.

These inform the viability of using Closure Calculus as the theoretical basis for a programming language, a topic ripe for exploration in the near future. There are several implications of using Closure Calculus as a theoretical basis for a programming language. Two of which serve the aims of the stakeholders. The remaining implications are explored in in Chapter 8.

Thus, the implementation of a Closure Calculus interpreter, and the subsequent comparisons against interpreters for the pure λ-calculus serves the objective to show that Closure Calculus is better than the pure λ-calculus.

Significance

If it can be shown that Closure Calculus is simpler to implement than the pure λ-calculus, what logically follows is that programming languages that use Closure Calculus as a theoretical basis will also be simpler to implement when compared to a pure λ-calculus counterpart. Thus the first objective is significant to the first aim of the stakeholders.

The second objective is a consequent to the fact that Closure Calculus is simpler than the pure λ-calculus in theory, hence implementations would be faster due to less work being done. What follows from this is that programming languages that use Closure Calculus as a theoretical basis would yield faster programs at run time too. This is significant to the second aim of the stakeholders.

Overall, this has an impact in the design and implementation of future functional programming languages. In particular, it shows that Closure Calculus can be used as a basis to replace λ-calculus in compilation pipelines.

The significance of the objectives to the aims of the stakeholder is summarised in the Table 1.3 - the main aims, objectives and significance are bolded while the unbolded entries are componentwise breakdowns of the aims, objectives and significance:

Aims	Objectives	Significance if outcome of objectives is a success
Find a better lambda calculus for the theoretical basis of functional programming languages.	**Show Closure Calculus is a better lambda calculus than the pure λ-calculus**	**Closure Calculus is a good candidate for the theoretical basis of functional programming languages.**
Simpler compilation and runtime components	Implement an interpreter for Closure Calculus and compare against interpreters of λ-calculus to show that Closure Calculus is simpler	Implementations of programming languages would be simpler, leaving less surface area for errors to manifest.
Faster programs at run time	Implementing an interpreter for Closure Calculus allows exploration of implications: Faster program execution	Functional programming languages that generates binaries that are more performant

Table 1.3: Significance of the objectives to the aims

1.5 Methods

In order to achieve the stated objectives, the following is done.

First, two interpreters for Closure Calculus are implemented. This corresponds to the two variants of Closure Calculus - CCN and CCDB. These interpreters are naively implemented without consideration for implementation efficiency.

Second, multiple interpreters for λ-calculus are implemented. They correspond to each of the following operational semantics: call-by-value, call-by-name and normal order.

Third, a test suite of equivalent programs for both variants of Closure Calculus, and λ-calculus was created. This allows for meaningful comparison between evaluation of program-input pairs for Closure Calculus and λ-calculus.

Fourth, the test suite of programs is run for each interpreter. Comparisons on the work done by each interpreter.

Last, benchmarks are performed across select programs. This allows for comparison of performance of Closure Calculus versus the performance of λ-calculus in evaluating their respective programs.

Work is measured as the number of term reductions and the number of meta-operations required to evaluate a program-input pair. In particular, the meta-operation of substitution is considered for λ-calculus, while the meta-operation of variable equality is considered for all implementations of Closure Calculus and λ-calculus.

Additional meta-operations like set union and fresh variable generation are also counted, but not considered work done for the purposes of comparison. This is due to the recursive nature of these meta-operations. For example, capture-avoiding fresh variable generation subsumes variable equality checks.

1.6 Tests For Success

The tests for success would be that the implementation for both Closure Calculus interpreters are indeed simpler in the following ways:

1. The interpreters for Closure Calculus are simpler than the interpreters for λ-calculus by some commonly used complexity metrics: Lines of Code, COCOMO and Cyclomatic Complexity.
2. When interpreting programs-input pairs of the respective calculi, the Closure Calculus implementations do less work. This is compared by means of counting reduction steps and meta operations.
3. Consequent to having to do less work, the Closure Calculus implementations should be more performant than the λ-calculus implementations. This will be compared by various benchmarks on equivalent program-input pairs.

1.7 Structure of This Report

This report is structured in a way that benefits readers of different experience levels. The report is split into three parts: Preliminaries, Comparisons and Implications. An appendix follows.

Preliminaries contains the chapters that are considered necessary preliminaries to the implementation and results. The last three chapters of the Preliminaries share the same internal structure. The chapters discuss Term Reduction Systems, the pure λ-Calculus and Closure Calculus respectively. Readers already familiar with each of these may opt to skip them. However, it should be noted that these three chapters are structured thus so readers may infer from structure of the chapters alone, that Closure Calculus is itself simpler than the pure λ-calculus.

The Comparisons section starts with a chapter explaining how fair comparisons are to be made. This is followed by a chapter which shows that Closure Calculus is simpler and a chapter which shows that Closure Calculus is faster.

The last part of this report explores the implications of Closure Calculus. The Discussions chapter frames the use of Closure Calculus in the context of implementing industrial functional programming languages and explores the implications of using Closure Calculus as a theoretical basis of functional programming languages. The chapter titled Interesting Artefacts of Closure Calculus reports on the other implications that Closure Calculus may have. The final chapter reinforces the main thesis of this report: that Closure Calculus is better than the pure λ-calculus.

Figure 1.7.1 shows the dependency order of the chapters in this report. It is the recommended way to read this report.

$$\begin{tikzcd}
 & & \text{Chapter 1} \arrow[dl] \arrow[d] \arrow[dr] & \\
\textbf{Preliminaries} & \text{Chapter 2} \arrow[r] \arrow[rr, bend right] & \text{Chapter 3} \arrow[d] \arrow[dr] & \text{Chapter 4} \arrow[dl] \arrow[d] \arrow[dd, bend left] \\
\textbf{Comparisons} & \text{Chapter 5} \arrow[r] \arrow[rr, bend right] & \text{Chapter 6} \arrow[d] & \text{Chapter 7} \arrow[dl] \\
\textbf{Implications} & \text{Chapter 10} & \text{Chapter 9} \arrow[l] & \text{Chapter 8} \arrow[l]
\end{tikzcd}$$

Figure 1.7.1: Dependency Order of Chapters

1.8 Notation Conventions

The term "lambda calculus" is used when referring to a generic member of the family of calculi, of which Closure Calculus and the pure λ-calculus are members. By contrast, "λ-calculus" or "pure λ-calculus" refers to the lambda calculus first introduced by Church.

Languages

This work involves descriptions of calculi, expressed as formal languages. A more careful treatment of languages will be given in Chapter chapter 5. However, here we will give a brief introduction to formal languages.

Let there be a set letters called an alphabet, Σ. A word is a combination of letters from the alphabet. We say that words are formed over the alphabet. The set of all words is written as Σ^*. Combinations of words are called sentences, terms, or expressions depending on context. A sentence may be well-formed or gibberish. The rules that govern whether a sentence is well formed is called a grammar. Just as English has its own grammar, so too do the languages of Closure Calculus and the pure λ-calculus. A language $\mathscr{L}$ over an alphabet Σ is a subset of Σ^* with a grammar.

Unless called for by the discussion, we omit defining a language by its alphabets and letters. Instead, we define languages by their grammar, presented in BNF.

When discussing the calculi expressed as a language, it is inevitable we will require a language of languages - a meta-language - to describe the calculi. It is vital to separate the meta-language from the object language. A meta-language is a language used to describe the object language. The object language in question may be of Closure Calculus or the pure λ-calculus. Reminders will be placed at appropriate points to remind reader of the nature of the language in question.

Variables and Placeholders

Further, this report concerns the notion of variables. A more complete treatment of variables is provided in the appendix. However, it is important to clarify the difference between a placeholder and a variable. A placeholder is a meta-syntactic notion to help readability of this report. Placeholders will be defined using the symbol := and the names of placeholders will be in **boldface**. An example follows:

Example. Given the following:

$$\mathbf{id} := \lambda x.x \tag{1}$$

$$(\lambda y.\mathbf{id}\ y)\ z \to^* z \tag{2}$$

Expression (2) should be read as $\lambda y.\underline{(\lambda x.x)}\ y)\ z \to^* z$, where the underlined subexpression was defined as **id**.

Placeholders may also be defined in midst of longer reduction sequences. In such situations, the symbol ≡ will be used. The ≡ symbol will also be used when denoting that two expressions are equivalent.

Example. Given a reduction sequence

$$\begin{aligned} &\ldots \\ &\to^* (\lambda x.x)\ (\lambda x.x) && (1) \\ &\equiv \mathbf{id}\ \mathbf{id} && (2) \\ &\equiv (\lambda x.x)\ \mathbf{id} && (3) \\ &\ldots \end{aligned}$$

Expression (2) means that the previous line is to be abbreviated into placeholders. Expression (3) "desugars" one of the **id** back into its long form while keeping the other as a placeholder. It says that expression (3) is equivalent to expression(2).

Other Notes on Notation

Any words in NOUN face will have a corresponding entry in the nomenclature section.

M and N are usually used throughout this report as valid arbitrary terms.

Chapter 2

On Abstract Reduction Systems, Term Rewriting Systems and Various Preliminaries

This chapter provides only an overview of rewriting systems. It is enough only to give specificity to the Pure λ-calculus and Closure Calculus. For a more thorough treatment of term rewriting systems and abstract rewriting systems, see TeReSe (Bezem et al. 2003).

The pure λ-calculus and Closure Calculus are term rewriting systems. Often the notion of a term rewriting system is confused with that of an abstract reduction system. To clarify, a reduction system is a more general notion of a term rewriting system (Klop 1990; Baader and Nipkow 1999).

A term rewriting system can be thought as a subset of abstract reduction systems which also include systems like string rewriting systems and Post-Thue systems. Figure 2.0.1 depicts the relationship between reduction systems, term rewriting systems, and calculi in the λ-calculus family.

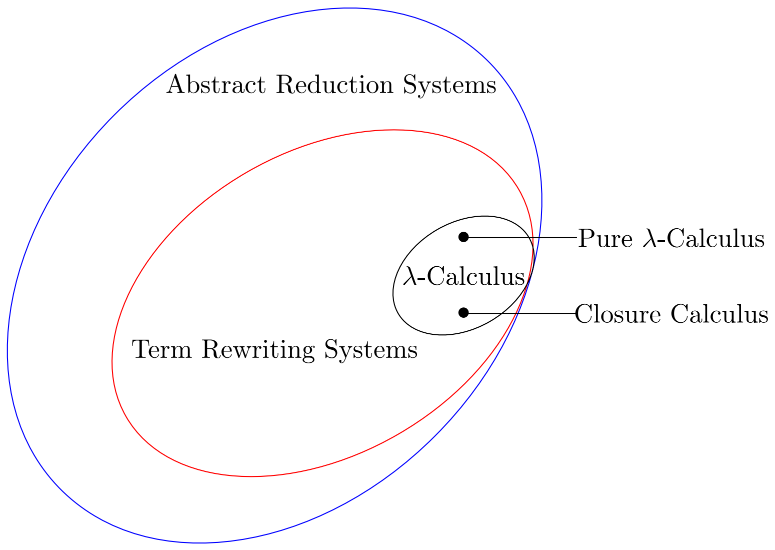


Figure 2.0.1: Reduction systems and term rewriting systems

Following in the footsteps of Bezem et al. (2006), a basic, informal example goes a long way in illustrating the notion of "rewriting" and "reduction". This example will be used as the running example throughout this chapter.

Example. Let $expr = (1 + 2) + (3 + 4)$, we shall now consider the evaluation of this expression.

We may say that the expression $1 + 2 + 3 + 4$ evaluates to 10. Being such a simple arithmetic expression, we may often forget it's done in multiple steps, each step reducing the length of the expression, as follows:

$$\begin{aligned} expr &= \underline{1+2} + 3 + 4 \\ &= \underline{\color{red}{3} + 3} + 4 \\ &= \underline{\color{red}{6} + 4} \\ &= \color{red}{10} \end{aligned}$$

Each "step" is headed with the $=$ symbol. The resulting expression lies to the right of the $=$ symbol. Red and *underlined* terms will be explained shortly.

Several observations can be made.

First, consider each "step". The results of each "step" gets progressively shorter. This is the notion of "reduction" - i.e. for $1 + 2 \rightarrow 3$, we can say that $1+2$ *reduces to* 3. The $\rightarrow$ is called a *reduction relation.* The $=$ symbol is used to represent an equivalence relation - i.e. when both expressions on either side of the $\rightarrow$ denote the same number, and when the relation is symmetric, transitive and reflexive. Do note that the equivalence relation is dependent on the term rewriting system. The pure λ-calculus has its own notion of an equivalence relation, as does Closure Calculus.

Next, note that the end result, 10, is an expression that may not be reduced any further. When an expression may not be reduced any further, we say that the expression is in NORMAL FORM.

Then, observe that within each expression, there are sub-expressions. The underlined sub-expressions are chosen to be reduced next. The result of the reduction is bolded in the following line. In this specific instance, the choice of which sub-expression to reduce first does not matter, in that reduction to normal form will yield the same normal form as the steps above. This property is called the UNIQUENESS OF NORMAL FORMS.

$$\begin{aligned} expr &= 1 + 2 + \underline{3 + 4} \\ &= \underline{1 + 2} + 7 \\ &= \underline{3 + 7} \\ &= 10 \end{aligned}$$

Last, consider the underlined sub-expressions and the red results on the following lines. We may analyse the underlined sub-expression as a CONTEXT in which a reduction occurs. Once the expression in the context is reduced, the result is place in the stead of the original expression. We may think of them visually as boxes.

$$\boxed{\boxed{\boxed{1+2}+3}+4} \tag{step 0}$$

$$\boxed{\boxed{\boxed{1+2 \to 3}+3}+4} \tag{step 1}$$

$$\boxed{\boxed{\boxed{3}+3}+4} \tag{step 1a}$$

$$\boxed{\boxed{3+3}+4} \tag{step 2}$$

...

Observe in the preceding example that the red box stays in place, but the values in them change. It is in this sense that we call this "rewriting".

A note must be made. In step 2, the red box is removed because $1 + 2 \to 3$, and 3 is in normal form, it's no longer reducible, hence no longer a context. A more careful treatment of contexts, and what can or cannot be a context will be provided in the sections below. The specific definitions of contexts in λ-calculus and Closure Calculus will be provided in their respective chapters.

With this general overview done, we may now proceed to more carefully define an abstract reduction system, followed by defining term rewriting systems. This example will be used as a running example to clarify certain ideas.

Abstract Reduction Systems

An abstract reduction system $\langle A, \{\rightarrow_\alpha\}\rangle$ consists of a structure that contains a set of objects A; and a set of binary relations $\{\rightarrow_\alpha\}$ on A (Klop 1990). The binary relations are called REDUCTION RELATIONS.

There may be many kinds of reduction relations in a reduction system. This is denoted by the subscripted α in the arrow. If there is only one kind of reduction relation, then we may omit the subscript.

We give the following definitions to further discussion on Closure Calculus:

- If for $a, b \in A$ and $a \rightarrow_\alpha b \in \{\rightarrow_\alpha\}$, then we call b the one-step α-reduct of a. In other words, we may say it as "a reduces to b in one step".

- If for $a, b, c \in A$, $a \rightarrow_\alpha b \in \{\rightarrow_\alpha\}$ and $b \rightarrow_\alpha c \in \{\rightarrow_\alpha\}$ then $a \rightarrow_\alpha b \rightarrow_\alpha c$ is a valid reduction for which a transitive, reflexive closure of $\rightarrow_\alpha$ over $\{a, b, c\}$ exists. The transitive, reflexive closure of $\rightarrow_\alpha$ is written $\rightarrow^*_\alpha$. When we write $a \rightarrow^*_\alpha c$, this implies there exists a finite but possibly zero length sequence of reductions between a and c. We may say this as "a reduces to c".

- If for $a \in A$, there are no $b \in A$ such that $a \rightarrow_\alpha b$, then we say a is in NORMAL FORM. Following this, if for $a, b \in A$ and $a \rightarrow^*_\alpha b$, if there is no $c \in A$ such that $b \rightarrow_\alpha c$, then we say that b is the normal form for a.

Term Rewriting System

A term rewriting system (TRS) is a more specific form of an abstract reduction system. Where in an abstract rewriting system there's a set of objects A, in a term rewriting system the objects are first order terms. The reduction relations are then the rewrite rules.

λ-calculus and Closure Calculus are a higher-order term rewriting systems because the calculi have notions of bound variables. Nonetheless, it's useful to have an understanding first order term rewriting systems.

The Syntax of Term Rewriting Systems

A term rewriting system is made up of a pair (Σ, R), of signature Σ and rules R. The signature Σ is defined as being made up of:

1. Variables drawn from an countably infinite set, Var. They are written $x, y, z...$ or $x_0, x_1, x_2...$ when indices are needed.

2. A set of operator symbols. Each symbol has an arity - a natural number which denotes how many "arguments" the operator is supposed to have. For example, in the pure λ-calculus, the operator λ has an arity of 2: the name of the bound variable and the body. The identity operator I in Closure Calculus has an arity of 0 - it does not require any arguments for it to be complete.

2.1 Terms

The set of terms or expressions, $Terms(\Sigma)$ can therefore be defined inductively over the alphabet Σ as such:

1. $\forall x, y, z ... \in Var,\ x, y, z ... \in Terms(\Sigma)$.
2. if O is an operator with an arity of n ; and $x_0, x_1 ... x_n \in Term(\Sigma)$ then $O\, x_0\, x_1 \dots x_n \in Term(\Sigma)$.

Defined thus, a term is the object that being operated upon by the rules of the rewriting system. A term may nest - terms may contain other terms. A term with no variables in it is called a closed term.

When discussing terms in the context of term rewriting systems, the word "terms" will be used. The alternate name "expressions" will be used when discussing terms in the context of implementations and in showing actual reduction steps. Most terms are defined in BNF form in this report.

Example. We may define the terms of our running example in BNF form as follows:

$$\begin{aligned} t ::= &\ x \in Var \\ \mid &\ n \in \mathbb{N} \\ \mid &\ + t\ t \end{aligned}$$

The BNF is read thus - t is a meta-variable which stands for a term. Then we define x to be a meta-variable denoting a variable symbol. In future discussions, "$\in Var$" is assumed automatically. n is any natural number. Last, the operator $+$ is an operator with an arity of 2. We usually write it infix, such that $(+\ t\ t)$ is written $t + t$.

Contexts

A rough notion of rewriting contexts was introduced in the introduction to this chapter. This section refines the notion of a context. A context is a syntactic device to aid in description of terms and operations on terms.

First consider the alphabet Σ. Extend it with a new symbol $\Box$, making it $\Sigma \cup \{\Box\}$.

A term over the extended alphabet $\Sigma \cup \{\Box\}$, if it contains one or more occurrences of $\Box$, is called a CONTEXT.

What does $\Box$ denote? It denotes a hole - an incomplete term, so to speak. Holes may be filled in with other terms.

A one-hole context is written $C[]$. If a term $t \in Terms(\Sigma)$ can be written such that $t \equiv C[s]$, where $s \in Terms(\Sigma)$, then the term s is called a subterm. $C[]$ is then the prefix of the term t.

Example. Let $\Sigma = \{\lambda x., \lambda y., \lambda z., App\} \cup Var$, where the operators $\lambda x.$, $\lambda y.$ and $\lambda z.$ have arities of 1 each[1]. The *App* operator has an arity of 2. Instead of writing it $App(Term_1, Term_2)$, we will write it infix, like so $(Term_1\ Term_2)$. Now we have the following:

- There are no closed terms possible (at least in a first order term rewriting system - we now begin to see why a higher order system is required to describe λ-calculus).
- $\lambda x.x$ is a term with a single occurrence of the variable x. This is sometimes called a linear term.
- $\lambda x.(x\ x)$ is a term with two occurrences of the variable x. Then $\lambda x.\Box$ and $\lambda x.(\Box\ \Box)$ are prefixes for $\lambda x.(x\ x)$. More specifically, $(\Box\ \Box)$ is a sub-context of the context in $\lambda x.\Box$. Hence $(x\ x)$ is a subterm of $\lambda x.(x\ x)$.
- $\lambda x.\lambda y.x$ is a term. Then $\lambda x.\Box$ and $\lambda x.\lambda y.\Box$ are prefixes of $\lambda x.\lambda y.x$. We say $\lambda y.x$ is a subterm for $\lambda x.\lambda y.x$.

NB: λ-Calculus is a Higher-Order Rewriting System

It should be noted that λ-calculus is a higher-order term rewriting system. This is due to notion of bound and free variables in λ-Calculus. A finer treatment on bound and free variables will be given in the following chapter on λ-Calculus. However, for now, consider the following expression:

$$\begin{tikzcd}[column sep=tiny] \lambda x . & \underline{x} \arrow[l, bend left] & y \end{tikzcd}$$

The latter x is bound to the binding variable of λ. y is a free variable.

The existence of variable binding requires that the TRS be considered on a higher order - there must be a meta-term and the terms of the calculus itself. Three extensions to the alphabet Σ are required in order to define meta-terms (meta-variable t):

1. Variables as defined in the first-order TRS (meta-variable x).
2. Operators, as defined in the first-order term rewriting system. However in particular, the abstraction operator λ, in conjunction with a binding variable symbol x must be defined. It is usually written as $\lambda x.t$.
3. Applications of meta-variables $f(t_0, t_1 ... t_n)$.

[1]We sidestep the fact that λ-calculus requires a higher order system to be described by fixing both the number of operators, and the operators themselves. The reason for using λ notation is because it would serve as a more illustrative example for readers who might already be familiar with λ-calculus

The meta-terms are only used to describe the terms of the calculi and their rules. Further, the rewrite rules are closed meta-terms - there are no meta-variables in rewrite rules.

If there appears to be confusion, it is normal. Describing λ-calculi in general require a "meta"-level. However, as will be presented later, there are meta-operations that exist in the pure λ-calculus that doesn't exist in Closure Calculus. It is in this sense that the theory of Closure Calculus is simpler than the theory of pure λ-calculus.

Reader note: Should there be an example here to explain what I mean?

2.2 Substitution

The lengthy build up of contexts and the notion of holes to be filled serves one purpose: to facilitate the discussion of the notion of substitutions.

A substitution σ is a function that replaces variables with terms. It can be conceptually thought as a mapping of variables to terms, $\sigma : Var \to Term$. Application of substitution to terms are subject to satisfying

$$\sigma(O\ t_1...t_n) \equiv O\ \sigma(t_1)...\sigma(t_n)$$

for every n-ary operator O. If O has an arity of 0, then $\sigma(O) \equiv O$.

The domain of a substitution σ is written as $Dom(\sigma)$. It is a restriction on the replacement of variables. Only variables found in the domain of σ will be replaced. For example, if the variables x and y exist in the term t_1, but only $x \in Dom(\sigma)$, then σt_1 will only replace x in t_1, and not y.

A substitution is written $\{u_0/x_0, u_1/x_1...u_n/x_n\}$, where each u/x is to be read "substitute u for x".

Substitutions may be applied on terms. It is written as such: $\{u/x\}t$. What this means is to substitute all instances of the variable x in term t with term u.

A note on notation: the use of curly braces indicates that substitution is a meta-operation. In the chapter on Closure Calculus, an alternate notation will be used to represent the idea of substitution, as substitution in Closure Calculus is a first class object.

2.3 Rewrite Rules

Having defined what a term is, and what a substitution is, all that remains in defining a term rewriting system is the notion of a REWRITE RULE. A rewrite rule for a signature Σ is a binary relation $\to$ on the terms of $Terms(\Sigma)$. It is written as $\rho : l \to r$, where ρ is the name of the rule.

Two conditions are imposed as follows:

1. The left hand side of the binary relation, l is not a variable.

2. Every variable in the right hand side, r must occur in l.

A reduction rule $l \to r$ may be thought of as a scheme, particularly when it comes to the pure λ-calculus. An instance of a rule ρ is obtained by applying a substitution σ, called an atomic reduction step, written $l^\sigma \to_\rho r^\sigma$.

Upon application of the substitution σ, the left hand side l^σ is a term (expression) immediately reducible. For that, we call such such terms/expressions a reducible expression, or REDEX, for short.

A rewrite step therefore is the idea of rewriting a redex *within* a context:

$$C[l^\sigma] \to_\rho C[r^\sigma]$$

The $\to_\rho$ is called the one-step reduction relation generated by ρ. In the later chapters on the pure λ-calculus and Closure Calculus, reduction rules listed are one-step reduction relations, and are written with the harpoon symbol, $\rightharpoonup$.

2.4 Normal Forms

When an expression no longer contain any redex as its subterms, it's called a NORMAL FORM (NF). There are variants of the normal form - Weak Normal Form, Head Normal Form and Weak Head Normal Form. These pertain specifically to the pure λ-calculus and will be discussed in the chapter on the pure λ-calculus. There is only one kind of normal form in Closure Calculus.

A complicated expression may have multiple redexes ready to be reduced. There is hence a need to choose which redex to reduce first. This is called a REDUCTION STRATEGY. A reduction strategy corresponds to small-step semantics (Wright et al. 1994). However, reduction strategies only make sense if the reduction rule is confluent. Confluence is explored in the next section.

A TRS is strongly normalising if every sequence of rewrites yields a NF. A TRS is weakly normalising if there exists a sequence of rewrites of a term that yields a NF. Neither Closure Calculus nor the pure λ-calculus are strongly normalising. However, in Closure Calculus, all programs and values can be written in normal form. This appealing property will be explored further in the chapter on Closure Calculus.

2.5 Confluence and Termination

When reducing a term, occasionally a choice has to be made on which subterms to reduce first. A reduction system is confluent if the same result is yielded, no matter the choice.

Using our running example, we see that $(1 + 2) + (3 + 4)$ may be reduced in two ways. Both reductions ultimately reduce to $(3 + 7)$, which may be further reduced to 10.

$$\begin{tikzcd}
& (1+2)+(3+4) \arrow[dl, "\underline{(1+2)}+(3+4)"'] \arrow[dr, "(1+2)+\underline{(3+4)}"] & \\
3+(3+4) \arrow[dr] & & (1+2)+7 \arrow[dl] \\
& 3+7 &
\end{tikzcd}$$

We say that the reduction is confluent, because regardless of which path we take, we get the same result within one step. This is important - it would be rather unhelpful for a calculus to have reduction rules that diverge on choice of redex!

The confluence property implies that there is only one NF for each expression. This property is called uniqueness of normal forms. This is a useful property

The reduction of terms in the running example terminates. This is to say if we continuously apply reduction rules, eventually the terms come to a normal form.

In the chapters on the pure λ-calculus and Closure Calculus, we will explore terms that will never terminate under their specific reduction rules.

A final note on confluence and termination - a reduction rule may lead to non-termination, but can still be confluent. This requires the notion of local confluence, which will be introduced in the chapter on the pure λ-calculus.

2.6 Evaluation and Operational Semantics

Terms in a term rewriting system may be evaluated. An evaluation of a term results in a "final result" of the reduction. These "final results" are called VALUES. Note that the notion of "final" result may be arbitrary. The OPERATIONAL SEMANTICS of a term rewriting system defines how evaluation is to proceed.

Usually, values are a subset of the syntax. This does not have to be the case. Using the running example in this chapter, we may define the terms and values, followed by its operational semantics as follows:

$$\begin{aligned} t, u &:= t + u \mid n \in \mathbb{N} && \text{(terms)} \\ v &:= n \in \mathbb{N} && \text{(values)} \end{aligned}$$

$$\frac{n, m \in \mathbb{N}}{n + m \to [\![n + m]\!]} \quad \frac{t \to v}{t + u \to v + u} \quad \frac{u \to v}{t + u \to t + v}$$

The syntax of the TRS is presented in BNF form. This differs from an earlier example where the terms are defined inductively. Here it is defined recursively.

Terms take two forms: the first is $t + u$, where t and u are meta-variables that may be replaced with another term; the second is a number n, drawn from the natural numbers. Here, n is also a meta-variable.

The values of this TRS is defined to be only a natural number.

The operational semantics of this TRS is written as a derivation tree. The statement over a line is a premise while the statement under a line is the conclusion.

Informally, the first rule says that if m and n are both numbers, then the expression $n + m$ will be evaluated as the result of adding the two numbers up.

The second rule states that if a term t evaluates to a value v, then the expression $t + u$ is to be evaluated as $v + u$. u is a meta-variable standing in for another term.

The last rule is like the second rule, except applied to the second operand of the $+$ operator.

Together, these three rules define a term reduction system that was used as a running example in this chapter. Hence we may reduce $1 + 2 + 3$.

2.7 Some Remarks on ARSs and TRSs

Abstract reduction systems and term reduction systems are a large field of study. TeReSe (Bezem et al. 2003) remains one of the better guides to both.

However, an interesting observation may be made - tree walking interpreters (introduced in a later chapter) are good ways to implement the operational semantics of TRSs.

Chapter 3

The Pure λ-Calculus

Church's discovery of λ-calculus was an unintentional side effect of his pursuit of the answer to the consistency of mathematics itself. Church had believed "...the entities of formal logic are abstractions, invented because of their use in describing and systematising facts of experience or observation, and their properties, determined in rough outline by this intended use, depend for their exact character on the arbitrary choice of the inventor"(Church 1932, pp. 348), and hence set out to formalise the very notion of functions.

The first version of the λ-calculus was very quickly determined to be inconsistent (Kleene and Rosser 1935) - it contained a contradiction. That contradiction spurned Church to refine his calculus, giving birth to the pure λ-calculus that is familiar to this day (Church 1936). The epistemological nature of Church's λ-calculus has since been overshadowed by the main thesis of his paper, that "...an effectively calculable function of positive integers [can be defined] by identifying it with the notion of a recursive function of positive integers (or of a lambda-definable function of positive integers)" (Church 1936, pp. 356).

The pure λ-calculus is extremely simple. There are only three terms: *variables*, *applications* and λ-*abstractions*. Despite that, it is capable of representing any computable functions and thus, is Turing-complete (Barendregt 1985). This resulted in λ-calculus being used as a theoretical foundation and inspiration for many functional programming languages. ML, Ocaml and Haskell have a lambda calculus as the theoretical foundation while LISP was inspired by the pure λ-calculus.

This chapter proceeds as follows: first the terms of the pure λ-calculus is given. Then the notion of substitution is defined. This is followed by definitions of the rewrite rules and the various normal forms that terms of the pure λ-calculus may take. A brief tour on the confluence and termination properties of λ-calculus follows. Next, evaluations and various operational semantics of the pure λ-calculus are considered. The chapter then closes with some remarks on the pure λ-calculus.

3.1 Terms

The terms of the pure λ-calculus is given as follows in BNF form.

$$\begin{array}{rlr} t, u := & & \text{(term)} \\ & x & \text{(variable symbol)} \\ & \mid (t\ u) & \text{(application)} \\ & \mid \lambda x.t & \text{(abstraction)} \end{array}$$

Figure 3.1.1: Terms of λ-calculus

A term in the pure λ-calculus are either variables, applications of two terms, or a λ-abstraction. A note on notation - M, N are used throughout this chapter as arbitrary λ-terms.

A variable is a symbol drawn from an infinite set of symbols, usually the English alphabet.

An application of two terms $(t\ u)$ is the application of the term t to the term u. Application is left associative. If t, u, v are terms, then $t\ u\ v$ is equivalent to $(t\ u)\ v$. Applications also have higher precedence (i.e. it binds tighter) than other terms, so $\lambda x.t\ u$ means $\lambda x.(t\ u)$ and not $(\lambda x.t)\ u$.

A λ-abstraction $\lambda x.M$ may be broken down into useful parts that will be given names, illustrated below.

$$\overbrace{\underbrace{\lambda x}_{binder} \, . \, \underbrace{M}_{body}}^{function}$$

We call $\lambda x.M$ the abstraction of t. We call λx a binder. We say that the variable x is bound to the body M. The body M is itself a λ-term. Any occurrence of x in M is called a binding.

The part called a function in the illustration is not a common name. However, it aids in the intuition of λ-calculus when it comes to implementing an interpreter. Specifically it calls to the intuition that λ as a standalone operator is a generator of functions, rather than the incorrect notion seen in popular web-based tutorials where the entire term is considered a function. The difference is admittedly subtle, but makes a world of difference when it comes to reasoning around the implications of λ-terms.

A λ-term is amenable to structural induction. By introducing named parts of the structure of a λ-term, it allows for an easy way to talk about such inductions.

The first subterm of a term is called the head. In an application $(x\ y)$, x is the first subterm, therefore it's the head. In the λ-abstraction $(\lambda x.((\lambda y.y)\ x)\ x)$, the first subterm is $((\lambda y.y)\ x)$. The head of a variable x is itself. This notion of a “head”ness of a term has to do with the notion of head-reduction, which are useful in the study of the theories of syntactic solvability of λ-calculus (Barendregt 1985, Part 1, Chapter 4 and Part 4). This is not explored here, however, the notion of the head is useful when discussing the various normal forms.

Contexts

A context is a λ term with some holes in it. A concrete definition of what a context is given below for λ-calculus:

- x is a context.
- $\square$ is a context.
- if $C_1[\,]$ and $C_2[\,]$ are contexts then so are $(C_1[\,]\ C_2[\,])$ and $\lambda x.C_1[\,]$.
- if $C[\,]$ is a context and M is an arbitrary λ-term, then $C[M]$ denotes the result of placing M in the hole. A free variable in M may no longer be free in $C[M]$.

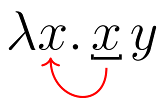
Example. Let $C[\,] \equiv \lambda x.\lambda y.\square$. If $M \equiv x$ then $C[M] \equiv \lambda x.\lambda y.x$.

3.2 Substitution

Given that the pure λ-calculus is a term rewriting system, substitution is an expected operation on the terms. Having defined the idea of a substitution context, a treatment on variables is needed before being able to discuss substitution in the pure λ-calculus.

A variable in the pure λ-calculus may be in one or two forms: a free variable or a bound variable. The notion of a free variable was briefly touched upon in the previous chapter. This section refines the ideas.

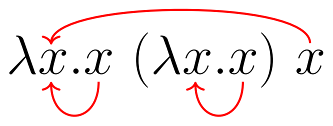
A variable that exists in a λ-abstraction may be free or bound. A bound variable is a variable that is contained in the context of a given binder. Revisiting the illustration from the previous chapter:



It’s easy to see that x is bound by the binder that the red arrow is pointing towards. y on the other hand is bound by nothing, therefore y is a free variable.

Let’s consider now the expression $\lambda x.(x\ (\lambda x.x)\ x)$. Parentheses have been inserted as to make clear the meaning of the expression. Which binders are the fourth and fifth x bound to?

The following illustration shows the relationship between binders and bindings.



We write $FV(M)$ as the set of free variables of an arbitrary λ-term M. It is defined thus:

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

where a variable is free in itself; $FV(M) \cup FV(N)$ is the union of the sets $FV(M)$ and $FV(N)$; and $FV(M)\backslash\{x\}$ is the set theoretic difference of $FV(M)$, obtained by removing x from $FV(M)$.

A substitution in the pure λ-calculus is a partial function from a variable to a term. We write a substitution as $\{M/x\}$ where M is an arbitrary valid term and x is a variable symbol. Alternately, we may also write a substitution using the meta-variable σ.

The application of a substitution σ to a term M is given by:

$$\begin{aligned} \sigma x &= u && \text{if } x \in Dom(\sigma) \\ \sigma x &= x && \text{if } x \notin Dom(\sigma) \\ \sigma(t\ u) &= (\sigma t)\ (\sigma u) && \\ \sigma(\lambda x.t) &= \lambda x.(\sigma t) && \text{if } \sigma \text{ avoids } x \end{aligned}$$

Being a map of variables to terms, a substitution σ may also contain free variables. We write $FV(\sigma)$ to mean the $FV(Range(\sigma))$, where $Range(\sigma)$ is the mapped-to term of each variable in $Dom(\sigma)$.

We now address the side condition to the final equation. "if σ avoids x" means $x \notin Dom(\sigma) \cup FV(\sigma)$.

Example. $\{M/x\}\lambda x.x$ is not $\lambda x.M$.

This is because the x (bolded for emphasis) under the λ is bound in $\lambda x.\mathbf{x}$. Therefore substitution may not occur.

Example. $\{y/x\}\lambda y.x$ is not $\lambda y.y$.

This is because the variable y is free in $\{y/x\}$, and it would become bound after substitution happens. Therefore substitution may not occur.

3.3 Rewrite Rules

There are three rewrite rules for λ-calculus, given as follows:

$$
\begin{aligned}
\{y/x\}\lambda x.t &\rightharpoonup \lambda y.\{y/x\}t && y \notin FV(t) && (\alpha)\\
(\lambda x.t)\ u &\rightharpoonup \{u/x\}t && \text{if } x \notin FV(t) \cup \{x\} && (\beta)\\
\lambda x.t\ x &\rightharpoonup t && x \notin FV(t) && (\eta)
\end{aligned}
$$

Figure 3.3.1: Rewrite rules for λ-calculus

Here, there are three rewrite rules. Of these, only the β rule is a reduction rule. The remaining rules are rewrite rules to facilitate conversion. Terms that are α-equivalent or η-equivalent may be converted from one to the other via the α or η rewrite rules.

Alpha Equivalence

The first rule is the α-rewrite rule. In order to understand the purpose of this rewrite rule, the notion of α-equivalence has to be introduced first. We say that two terms are α-equivalent if after applying a suitable instance of the α-rewrite rule to each term respectively, yields the same term. It should be noted that only variable-for-variable substitution is allowed.

Example. $\lambda x.x$ is α-equivalent to $\lambda z.z$.
We can see that the bound variable x has been renamed to z. Applying a substitution $\{z/x\}$ to $\lambda x.x$ will yield $\lambda z.z$. Therefore we say $\lambda x.x$ is alpha-equivalent to $\lambda z.z$.

Example. $\{(\lambda x.x)/y\}\lambda y.y$ does not trigger the α rewrite rule.
This is because $\lambda x.x$ is not a variable.

The notion of α-equivalence and consequently α-conversion plays a role in ensuring reduction happens in a sensible manner.

Beta Reduction

The purpose of the α-rewrite rule is to facilitate conversion between terms that are α-equivalent. This is required as the only reduction rule of the pure λ-calculus, the β rule requires a notion of bound variables.

A computation is said to have happened when an application of the β-reduction rule is performed. Observe that the β-rewrite rule has additional conditions attached to it. This condition is known as capture avoidance, a useful notion to avoid name clashes.

Example. Consider the term $(\lambda x.\lambda y.xy)\,y$. Reducing without capture avoidance would yield $\{y/x\}(\lambda y.xy) \rightarrow \lambda y.yy$. This is wrong. Instead, the correct thing to do would be to rename the bound variable in the left subterm first, yielding the following rewrite sequence:

Variant	BNF Describing the Variants
Normal Form (NF)	$n ::= x$ $\mid (x\ n)$ $\mid \lambda x.n$
Head Normal Form (HNF)	$n ::= x$ $\mid (x\ t)$ $\mid \lambda x.n$
Weak Normal Form (WNF)	$n ::= x$ $\mid (x\ n)$ $\mid \lambda x.t$
Weak Head Normal Form (WHNF)	$n ::= x$ $\mid (x\ t)$ $\mid \lambda x.t$

Table 3.1: Normal Forms of the pure λ-calculus. t is a meta-variable denoting an arbitrary term of the pure λ-calculus, as defined in Figure 3.1.1.

$$\begin{aligned}(\lambda x.\lambda y.x)\, y &\leftrightarrow_\alpha (\{a/y\}\lambda x.\lambda y.xy)\, y\\ &=_\alpha (\lambda x.\lambda a.xa)\, y\\ &\rightarrow_\beta \lambda a.ya\end{aligned}$$

Where does the new variable a come from? It comes from outside the system that we've defined here thus far - i.e. it's a meta-operation, called alpha-conversion which is written $\leftrightarrow_\alpha$.

Alpha conversion takes advantage of the fact that $\lambda y.xy$ and $\lambda a.xa$ are α-equivalent. By simply renaming y to a (i.e. applying the α rewrite rule), we have avoided a name clash. The meta-operation, alpha conversion is required for correct reduction.

Can the outer free variable y be renamed? No. There are no rules for renaming free variables.

3.4 Normal Forms

The concept of normal forms (NF) were introduced in the last chapter. The NF of λ-calculus are terms that has no reducible subterms. Table 3.1 summarises the variants of the normal forms of the pure λ-calculus in BNF.

For the purposes of evaluation of terms (i.e. for practical uses of λ-calculus), some other notions of normal forms have to be introduced. Different evaluation strategies will yield different kinds of normal forms. The section on evaluation and operational semantics details this fact.

A term is in HEAD NORMAL FORM (HNF) if the head is in head normal form. Note that a term in NF is a term in HNF. However, the converse is not necessarily true. A term may be in HNF but not in NF.

Example. $\lambda x.(x\,((\lambda y.y)\,x))$ is in head normal form, but is not in normal form.
The first subterm $(x\,((\lambda y.y)\,x))$ is in HNF. However its subterm $((\lambda y.y)\,x)$ is a redex and may be further reduced.

A term may be in a WEAK NORMAL FORM (WNF). A WNF differs slightly from HNF in that the body of a λ-term may be any term. However, the head and body of an application are in normal form, as defined in Table 3.1. A term in HNF is a term in WNF. However, the converse is not necessarily true. A term in WNF may not be in HNF and not be in NF.

Example. $(\lambda x.(\lambda y.y)\,x)$ is in weak normal form but not in head normal form and not in normal form.
The first subterm $(\lambda y.y)\,x$ is a redex and may be further reduced.

A term may also be in WEAK HEAD NORMAL FORM (WHNF). A term in WHNF is when it is not a redex. Terms often overlap in being in WNF and WHNF.

Example. $(\lambda x.(\lambda y.y)\,x)$ is in weak head normal form and in weak normal form, but not in head normal form and not in normal form.
This is because $(\lambda y.y)\,x$ is a redex and may be further reduced.

The purpose of introducing variations of normal forms for the pure λ-calculus has to do with evaluation. There are terms that cannot be reduced to a NF, but are reducible to a variant of a normal form.

Example. Let $\mathbf{\Omega} := (\lambda x.x\,x)\,(\lambda x.x\,x)$. Let $\mathbf{I} := \lambda x.x$. $\lambda x.\mathbf{I}\,x\,\mathbf{\Omega}$ may be reduced to $\lambda x.x\,((\lambda x.x\,x)\,(\lambda x.x\,x))$ which is in HNF, but not in NF.
Observe that $\mathbf{\Omega}$ reduces to itself. It will never be in normal form. The next section deals with termination and confluence, in which we shall explore $\mathbf{\Omega}$ in greater detail.

3.5 Confluence and Termination

Consider the following term:

$$\overbrace{(\lambda x.\underbrace{((\lambda y.xy)x)}_{r_1})a}^{r_2}$$

There are two redexes ready for reduction, labelled r_1 and r_2 respectively. Here, we have a choice of which redex to reduce first. If r_1 is chosen first, then the following reduction sequence occurs:

$$\begin{aligned}(\lambda x.((\lambda y.xy)x))a &\rightharpoonup (\lambda x.xx)a\\ &\rightharpoonup aa\end{aligned}$$

However, if r_2 were chosen, then the following reduction sequence occurs:

$$\begin{aligned}(\lambda x.((\lambda y.xy)x))a &\rightharpoonup (\lambda y.ay)a\\ &\rightharpoonup aa\end{aligned}$$

We say that the reduction of the term is confluent, because regardless of which path was taken, the final result is the same. More specifically, we say that β-reduction is confluent up to α-conversion.

$$\begin{tikzcd}
& \lambda x.((\lambda y.xy)x))a \arrow[dl, "r_1"'] \arrow[dr, "r_2"] & \\
(\lambda x.xx)a \arrow[dr] & & (\lambda y.ay)a \arrow[dl] \\
& aa &
\end{tikzcd}$$

The β-reduction rule of the pure λ-calculus is confluent, but it is not normalising. How is this possible? To investigate this further, we will have to refine our notions of what we have thus far called "confluence" of the β-reduction rule. To be really specific, the β-rule of the pure λ-calculus is locally confluent. Consider the reduction of the following term:

$$\begin{tikzcd}
& (\lambda x.xx)((\lambda y.y)a) \arrow[dl] \arrow[drr] & & \\
(\lambda x.xx)a \arrow[ddrr, bend right] & & ((\lambda y.y)a)\,((\lambda y.y)a) \arrow[dl, red] \arrow[dr, red] & \\
& ((\lambda y.y)a)a \arrow[dr, red] & & a((\lambda y.y)a) \arrow[dl, red] \\
& & aa &
\end{tikzcd}$$

Observe that the red arrows form the diamond property. However, the reduction of $(\lambda x.xx)((\lambda y.y)a)$ does not. The red diamond property is local to a portion of the graph. Hence we say that β-reduction is locally confluent. The reflexive transitive closure of the β-reduction, $\rightharpoonup_\beta^*$ is confluent however. Martin Lof [?] proved that one may replace $\rightharpoonup_\beta^*$ with $\rightharpoonup_\beta$. This also allows us to say that β-reduction is confluent.

The key insight with local confluence however, is that it allows for reductions to be non-terminating and yet be confluent. This allows the calculus to still be useful while allowing for recursion.

Example. Let $\mathbf{\Omega} := (\lambda x.x\,x)(\lambda x.x\,x)$. It is obvious that $\mathbf{\Omega}$ reduces to itself, therefore never terminates. Now, let $\mathbf{K} := \lambda x.\lambda y.x\,y$ and $\mathbf{I} := \lambda x.x$. The term $\mathbf{KI\Omega}$ may reduce to normal form under some evaluation strategies.

3.6 Evaluation and Operational Semantics

In λ-calculus, evaluation is done by applying the renaming and reduction rules of λ-calculus on an expression of λ-calculus terms.

The values of the pure λ-calculus are variants of its normal forms. The kind of normal form depends on the evaluation strategies used. Table 3.2 summarises the kinds of normal forms taken as value for each evaluation strategy.

Evaluation Strategies

Evaluation Strategy	Reduce Args?	Reduce Args First?	Reduces Under λ?	Values Are
Applicative	Yes	Yes	Yes	Normal Form
Normal	Yes	No	Yes	Normal Form
CBV	Yes	Yes	No	Weak Normal Form
CBN	Yes	No	No	Head Normal Form
Lazy	No	No	No	Weak Head Normal Form

Table 3.2: Summary of evaluation strategies of λ-calculus and the results

An EVALUATION STRATEGY is related to a reduction strategy. Following Felleisen et al (2009), this report treats an evaluation strategy as distinct from a reduction strategy. An evaluation strategy of λ-calculus revolves around how applications are reduced. A reduction strategy had previously been introduced - it refers to the choice of redex to reduce, and applies to all terms of the system.

This report considers five different evaluation strategies - call-by-value (CBV), call-by-name (CBN), call-by-need (Lazy), applicative order evaluation and normal order evaluation. Table 3.2 summarises the similarities and differences between the evaluation strategies.

To show each of the evaluation strategies, the following running example will be used, labelled with meta variables:

$$\underbrace{(\lambda x.\lambda y.x\ y)}_{t}\underbrace{((\overbrace{\lambda z.z}^{v})(\overbrace{\lambda a.a}^{w}))}_{u}\,b$$

Here, the expression is an application of two subterms, t and u. u itself is an application of two subterms, which we will call v and w. We may write the term as $t\,(v\,w)\,b$.

A normal order evaluation reduces the left-most outermost redex - $t\,(v\,w)$ first. The result is the application of $\{u/x\}$ to $\lambda y.x\,y$. An applicative order evaluation reduces the right-most inner most redex - $(v\,w)$ first. These two evaluation strategies then proceed with evaluations under the λ.

By contrast, CBV, CBN and Lazy evaluation orders do not reduce under the λ.

The following are reduction sequences for each of the evaluation strategies:

$$\begin{aligned}
(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))b &= \underline{(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))}b \\
&= (\lambda y.((\lambda z.z)(\lambda a.a))y)b \\
&= (\lambda y.\underline{((\lambda z.z)(\lambda a.a))}y)b \\
&= (\lambda y.(\lambda a.a)y)b \\
&= (\lambda y.\underline{(\lambda a.a)y})b \\
&= (\lambda y.y)b && \text{NF if } b \text{ is not present} \\
&= \underline{(\lambda y.y)b} \\
&= b
\end{aligned}$$

Figure 3.6.1: Normal order evaluation of the expression

$$\begin{aligned}
(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))b &= \underline{(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))}b \\
&= \underline{(\lambda x.\lambda y.x\ y)\underline{((\lambda z.z)(\lambda a.a))}}b \\
&= \underline{(\lambda x.\lambda y.x\ y)(\lambda a.a)}b \\
&= (\lambda y.(\lambda a.a)\ y)b \\
&= (\lambda y.\underline{(\lambda a.a)\ y})b \\
&= (\lambda y.y)b && \text{NF if } b \text{ is not present} \\
&= \underline{(\lambda y.y)b} \\
&= b
\end{aligned}$$

Figure 3.6.2: Applicative evaluation of the expression

$$
\begin{aligned}
(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))b &= \underline{(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))}b \\
&= \underline{(\lambda x.\lambda y.x\ y)\underline{((\lambda z.z)(\lambda a.a))}}b \\
&= \underline{(\lambda x.\lambda y.x\ y)(\lambda a.a)}b \\
&= (\lambda y.(\lambda a.a)\ y)b && \text{W(H)NF if } b \text{ is not present} \\
&= \underline{(\lambda y.(\lambda a.a)\ y)b} \\
&= (\lambda a.a)b \\
&= \underline{(\lambda a.a)b} \\
&= b
\end{aligned}
$$

Figure 3.6.3: Call-by-value evaluation of the expression

$$
\begin{aligned}
(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))b &= \underline{(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))}b \\
&= (\lambda y.((\lambda z.z)(\lambda a.a))y)b \\
&= (\lambda y.\underline{((\lambda z.z)(\lambda a.a))}y)b \\
&= (\lambda y.(\lambda a.a)y)b && \text{(W)HNF if } b \text{ is not present} \\
&= \underline{(\lambda y.(\lambda a.a)y)b} \\
&= (\lambda a.a)b \\
&= \underline{(\lambda a.a)b} \\
&= b
\end{aligned}
$$

Figure 3.6.4: Call-by-name evaluation of the expression

$$
\begin{aligned}
(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))b &= \underline{(\lambda x.\lambda y.x\ y)((\lambda z.z)(\lambda a.a))}b \\
&= (\lambda y.((\lambda z.z)(\lambda a.a))y)b \qquad \text{WHNF if } b \text{ is not present} \\
&= \underline{(\lambda y.((\lambda z.z)(\lambda a.a))y)b} \\
&= ((\lambda z.z)(\lambda a.a))b \\
&= \underline{((\lambda z.z)(\lambda a.a))b} \\
&= \underline{(\underline{(\lambda z.z)(\lambda a.a)})b} \\
&= \underline{(\lambda a.a)b} \\
&= b
\end{aligned}
$$

Figure 3.6.5: Call by need evaluation of the expression

Having shown the reduction sequences for each evaluation strategy, we may also define the operational semantics of each of the evaluation strategies. In practice, only the CBV and CBN strategies are used. Their operational semantics are given as follows

$$
\frac{}{x \Rightarrow x} \quad \frac{s \Rightarrow v}{s\ t \Rightarrow v\ t} \quad \frac{}{(\lambda x.t)v \Rightarrow \{v/x\}t}
$$

Figure 3.6.6: Operational Semantics of Pure λ-calculus under a CBN strategy

$$
\frac{}{x \Rightarrow x} \quad \frac{s \Rightarrow v}{s\ t \Rightarrow v\ t} \quad \frac{t \Rightarrow v}{s\ t \Rightarrow s\ v} \quad \frac{}{(\lambda x.t)v \Rightarrow \{v/x\}t}
$$

Figure 3.6.7: Operational Semantics of Pure λ-Calculus under a CBV strategy

3.7 Some Remarks on the Pure λ-Calculus

The pure λ-calculus is complicated. First, observe that the β-reduction rule is defined in the terms of a meta-operation of substitution. This necessitates definition of meta operations.

Second, for decades logicians and computer scientists were confused over the notion of evaluation strategies and reduction strategies. As recent as 1996, the notion of lazy evaluation was mistakenly described as "combining the advantages of normal order and applicative order evaluation" (Michaelson 1989, p.199), and that "[lazy evaluation and normal order evaluation] are often used interchangeably"(Abelson et al. 1996, p.542).

Chapter 4

Closure Calculus

Closure calculus (Jay 2018) is a term-rewriting system that began life as an exercise to re-state λ-calculus without requiring meta-theory while keeping the appealing properties of the pure λ-calculus. Its properties of progression and confluence has been proved (Jay 2017). There are two flavours of Closure Calculus, explained in the following sections. Closure Calculus draws from earlier works on explicit substitution(Abadi et al. 1990) as well as work on closure conversion (Reynolds 1972; Steele 1978; Appel 2007). Despite this, Closure Calculus is different from other lambda calculi with explicit substitution in that the only form of abstraction is a closure. Closure Calculus has all the desirable properties of λ-calculus: it allows for equational reasoning; it's a good theory of functions; and it is Turing-complete, allowing for arbitrary computations to be done.

The key differentiator of Closure Calculus from the pure λ-calculus is that does not require the meta-theory of substitution. Substitution is explicit in the terms of the language. This is also a differentiating point from other lambda calculi with explicit substitution. Where in other lambda calculi with explicit substitution uses multiple syntactic classes, Closure Calculus simply uses one syntactic class. Thus the meta-theory of α-equivalence is not required. The lack of meta-theory of substitution and α-equivalence allows for more efficient mechanisation of the calculus.

Closure Calculus comes in two flavours. The Closure Calculus with Names (CCN) contains variables with names. The Closure Calculus that is without names (CCDB) takes inspiration from de Bruijn indexing. CCDB comes with no meta theory, while CCN requires the meta-theory of variable equality.

This chapter explores Closure Calculus in the fashion set out by the chapters preceding this. First, the terms are defined. Then the notion of substitution and rewrite rules are defined. Next, the normal forms, confluence and termination properties of CCN and CCDB are described. Following this, an exploration of how evaluation of Closure Calculus terms may proceed. Last, some remarks on Closure Calculus are given.

4.1 Terms

The terms of both flavours of Closure Calculus are given as follows in BNF form.

$$
\begin{array}{lll}
s, t, u := & & \\
 & x & \text{(variable symbol)} \\
 & \mid t\,u & \text{(application)} \\
 & \mid t@u & \text{(tagged application)} \\
 & \mid I & \text{(identity operator)} \\
 & \mid \sigma :: x \mapsto t & \text{(extension)} \\
 & \mid \lambda[\sigma]x.t & \text{(abstraction)}
\end{array}
$$

(a) Terms of CCN

$$
\begin{array}{lll}
s, t, u := & & \\
 & J & \text{(zeroth index)} \\
 & \mid Rt & \text{(raise)} \\
 & \mid t\,u & \text{(application)} \\
 & \mid t@u & \text{(tagged application)} \\
 & \mid I & \text{(identity operator)} \\
 & \mid (u, s) & \text{(pair)} \\
 & \mid \lambda st & \text{(abstraction)}
\end{array}
$$

(b) Terms of CCDB

Figure 4.1.1: Terms of Closure Calculus

The main difference between CCN and CCDB is that there are variables in CCN and no notion of variables in CCDB.

CCN

The terms of CCN are introduced by contrasting against the terms of the more familiar pure λ-calculus.

A variable x and an application $t\,u$ serve the same purpose as they do in λ-calculus. As usual, a variable x is drawn from a countably infinite set. Like in λ-calculus, applications in CCN associates to the left.

Next we turn our attention to the tagged application, $t@u$. The tagged application plays a role in the reduction rules in that it differentiates redexes from non-redexes. A tagged application is never a redex, while a regular application is a redex. More on this will be explored in the section on reduction rules. Tagged applications are left-associative, but binds more tightly than regular application so that $a\,b@c\,d$ is equivalent to $(a\,(b@c))\,d$.

In contrasting against λ-calculus, an immediate stand out is the form for an abstraction. Where in λ-calculus an abstraction is written $\lambda x.M$, in CCN it is $\lambda[\sigma]x.M$. This form of abstraction is called a CLOSURE. The σ in $\lambda[\sigma]x.M$ is the environment of a closure, while x is the bound variable and the arbitrary term M is the body.

While the syntax of CCN does not preclude any other term taking the place of σ in an abstraction, the environment is usually of the form an extension or the identity operator I. The identity operator commonly serves as an empty environment. An extension has the form $\sigma :: x \mapsto t$. It is a term σ, concatenated with a mapping of a variable to another term. The extension serves a dual purpose - as the environment of a closure, and as a substitution. These are explain in the rules in the next section. The implications of not restricting the form that σ could take in $\lambda[\sigma]x.t$ will be discussed in Chapter 8.

We name specific structures of the λ-abstraction $\lambda[\sigma]x.M$ so that we may easily refer to the subterms, illustrated below.

$$\underbrace{\lambda[\overset{\mathit{environment}}{\sigma}]}_{\mathit{binder}}\overbrace{x\,.\,\underbrace{M}_{\mathit{body}}}^{\mathit{function}}$$

Whereas in the pure λ-calculus, the λ is an operator that generates a function, here a λ generates an environment with a function. This is commonly known as a closure.

CCDB

The terms of CCDB are slightly different to the terms of CCN.

In the stead of variables are the zeroth index J and the raising operator Rt. These operate in similar ways to the de Bruijn indexing in λ-calculus. Instead of having a separate syntactic class for the variable index, as per the usual de Bruijn indexing, the indices themselves are constructed of terms. J is the zeroth index, which is bound by the immediate enclosing λ. RJ would be the first index, bound by the second immediate enclosing λ, and so on and so forth.

Not having a separate syntactic class for variable indices has some implications - some will be explored in Chapter 8. The use of Rs and J in conjunction with the reduction rules (introduced later in the chapter) appear to be similar to lifting and lowering operations in traditional accounts of de Bruijn indexing. In traditional accounts of using de Bruijn indexing in pure λ-calculus, lifting and lowering are meta operations to avoid the α-conversion rule. In Closure Calculus, there are no rules resembling the α-conversion rule. Thus, there is no need for a separate, meta-operation for lifting and lowering. So what are R and J? They are terms that when put together form an index to the bound variable. CCDB has effectively no meta theory because of this.

The extension syntax in CCN is also replaced with a pair, written (u, s). s is usually a meta-variable standing in for an environment. Like the extension form in CCN, the pair has dual functions, serving both as an environment to an abstraction - e.g s in λst, and serving as an explicit substitution.

Notational Shorthand

Given M as some valid arbitrary subterm, the following pattern is very common in CCN:

$$\lambda[I]x.\lambda[I :: x \mapsto x]y.\lambda[I :: x \mapsto x :: y \mapsto y]z.\lambda[I :: x \mapsto x :: y \mapsto y :: z \mapsto z]a.M$$

The mapping $x \mapsto x$ in the environment of the closure is required for proper reduction to take place. Terms may have long environments which would make for unwieldy reading.

For the sake of formatting, the following shorthand may be used:

$$\lambda[I]x.\lambda[I,x]y,\lambda[I,x,y]z.M$$

The syntactic shorthand $[I,x]$ should be read as $[I :: x \mapsto x]$.

This shorthand only applies in terms where the environment solely consists of mappings of variables to themselves.

Example. $\lambda[I :: x \mapsto (\lambda[I]x.x) :: y \mapsto y]z.x@y$ will not be written in notational shorthand. This is because the environment $I :: x \mapsto (\lambda[I]x.x) :: y \mapsto y$ contains mappings of variables to non-variable terms.

In CCDB, numerous raise operators R may obscure the understanding of the term. Consider the term $\lambda I\lambda(J,I)\lambda(J,RJ,I)...RRRRJ...$. A much more readable term would be $\lambda[]\lambda[0]\lambda[0,1]...4...$. A natural number is a shorthand of combinations of the raise operator R and the zeroth index operator J. Specifically the natural number i is shorthand for R^iJ. Additionally, observe that the identity operator is replaced with the symbols [and]. The identity operator is implicit in these cases. $\lambda[1]M$ should be read as $\lambda(RJ,I)M$.

Contexts

There is little need to introduce the notion of substitution contexts for Closure Calculus. In the pure λ-calculus, the notion of a substitution context is necessary because substitution is a meta-operation. Having a notion of substitution context aids both in reading and discussing terms of pure λ-calculus. Substitution in Closure Calculus is part of the calculus. So the introduction of substitution contexts is for really just comprehension benefits. Further, introducing the term rewriting contexts for Closure Calculus allows a like-for-like comparison with the pure λ-calculus. The term rewriting contexts for Closure Calculus is given as follows:

- x is a context in CCN.
- J and Rt are contexts in CCDB.
- $\square$ is a context.
- If $C_1[\,]$ and $C_2[\,]$ are contexts then so is $C_1[\,]@C_2[\,]$ and $(C_1[\,]\,C_2[\,])$.
- If $C[\,]$ is a context, then so are $\sigma :: x \mapsto C[\,]$ and $\lambda[\sigma]x.C[\,]$.
- If $C[\,]$ is a context and M is an arbitrary term of Closure Calculus, then $C[M]$ denotes the result of placing M in the holes. A free variable in M may become bound in $C[M]$.

Example. Let $C[\,] \equiv \lambda[I]x.\lambda[I,x]y.\square$. If $M \equiv x$ then $C[M] \equiv \lambda[I].\lambda[I,x]y.x$. Observe the free variable x in M is now bound.

4.2 Substitution

Substitution in Closure Calculus is not a meta-operation as is in the pure λ-calculus. Instead a substitution is part of the object language[1] of Closure Calculus. Substitution proceeds from following the rewrite rules, introduced in the next section.

In CCN, a substitution is written $(\sigma :: x \mapsto t)$, where σ is a meta-variable representing another substitution[2], x is a meta-variable for variable symbols, and t is the meta-variable for an arbitrary term. Note here, the dual tasks performed by the extension syntax. An extension in CCN is both a substitution (i.e it's a map from variable to term) and an environment in a closure.

In CCDB, a substitution is written (u, s) where both u and s are meta-variables for valid terms. Like the extension of CCN, the pair in CCDB performs dual duties in acting as a substitution and acting as an environment in a closure.

Usual treatments of substitutions when discussing λ-calculi involve a discussion of free variables. Free variables in Closure Calculus are truly free. Consider the following term:

$$(\lambda[I]x.\boxed{\lambda[I]y.x})\,a$$

The reduction rules will be introduced in the next section, but the term above reduces to $\lambda[I]y.x$. This may come as a surprise to the reader familiar with the conventions of the pure λ-calculus. Why is x not bound by the preceding binder?

Consider a simple rule, which can be stated in English as "a free variable is a variable that is not bound by a preceding binder". Now consider an "equivalent" term in the pure λ-calculus

$$(\lambda x.\boxed{\lambda y.x})\,a$$

Is x free? It depends on the context in which we are discussing x. If we are to discuss the underlined subterm $\lambda y.x$ by itself, then x is free. However, it is not free in the term $\lambda x.\lambda y.x$. Variable freedom is dependent on context, hence it is not universally defined.

By contrast, a free variable is free everywhere in Closure Calculus. In the term $\lambda[I]x.\underline{\lambda[I]y.x}$, the underlined subterm $\lambda[I]y.x$ and in the full term $\lambda[I]x.\lambda[I]y.x$, the variable x is free. No contexts are needed when discussing free variables.

With such restrictions on free variables, how may one do meaningful computation?

This is done by means of specifying in the environment, a variable to be substituted upon:

[1]previously introduced in Notation Conventions of Chapter 1.

[2]usually - we may of course, construct perverse terms, a subject that will be explored in Chapter 8

$$(\lambda[I]x.\lambda[I\boxed{::x \mapsto x}]y.x)\,a\,b$$

This term reduces to a as expected.

There is value in defining such a universal notion of free variables. The first is that by removing meta-rules makes reasoning around the mechanics of the reduction far simpler. When an interpreter encounters a free variable, it is free. There is no need to figure out the context in which the interpreter is currently in. The second benefit is that Closure Calculus is closer to the operational semantics of modern computing machines than the pure λ-calculus is. This is more thoroughly explored in Chapter 9.

4.3 Rewrite Rules

The rewriting rules of Closure Calculus only consist of reduction rules. It is presented as such:

$$\begin{gathered}
x\,t \rightharpoonup x@t \\
(s@t)\,u \rightharpoonup (s@t)@u \\
(\lambda[\sigma]x.t)\,u \rightharpoonup (\sigma :: x \mapsto u)\,t \\
I\,u \rightharpoonup u \\
(\sigma :: x \mapsto t)\,x \rightharpoonup t \text{ , if } z = x \\
(\sigma :: x \mapsto t)\,z \rightharpoonup \sigma\,z \text{ , if } z \neq x \\
(\sigma :: x \mapsto t)\,(u@v) \rightharpoonup ((\sigma :: x \mapsto t)\,u)\,((\sigma :: x \mapsto t)\,v) \\
(\sigma :: x \mapsto t)\,(\lambda[\rho]z.t) \rightharpoonup \lambda[(\sigma :: x \mapsto t)\,\rho]z.t \\
(\sigma :: x \mapsto t)\,I \rightharpoonup I \\
(\sigma :: x \mapsto t)\,(\rho :: z \mapsto u) \rightharpoonup (\sigma :: x \mapsto t)\,\rho :: z \mapsto (\sigma :: x \mapsto t)\,v
\end{gathered}$$

(a) Reduction rules of CCN

$$\begin{gathered}
J\,t \rightharpoonup J@t \\
Rtu \rightharpoonup Rt@u \\
(r@t)u \rightharpoonup (r@t)@u \\
(\lambda st)u \rightharpoonup (u,s)t \\
It \rightharpoonup t \\
(u,s)J \rightharpoonup u \\
(u,s)(Rt) \rightharpoonup st \\
(u,s)(r@t) \rightharpoonup (u,s)r((u,s)t) \\
(u,s)\lambda r\,t \rightharpoonup \lambda((u,s)r)t \\
(u,s)I \rightharpoonup I \\
(u,s)(r,t) \rightharpoonup ((u,s)r,(u,s)t)
\end{gathered}$$

(b) Reduction rules of CCDB

Figure 4.3.1: Reduction rules of Closure Calculus

The rewrite rules for both variants of Closure Calculus are numerous, but straightforwards. In CCN, two rewrite rules require meta operations of variable equality in order to substitute terms. Variable equality is a straightforwards meta-theory and implementation is trivial. The more discerning reader might also prefer CCDB, where there are no meta-operations required in its reduction rules.

Also, observe that there are no rules that resembles the α-rewrite rule in the pure λ-calculus. This implies not having a formally-defined α-equivalence. Hence $\lambda[I]x.x$ and $\lambda[I : y \mapsto z]x.x$ and $\lambda[I]y.y$ are all considered different terms, while in the pure λ-calculus, $\lambda x.x$ and $\lambda y.y$ are considered the same.

There is no need for capture avoidance either. The interaction between the environment subterm and extensions ensures that names are preserved throughout reduction.

Example. Consider the term $(\lambda[I]x.\lambda[I,x]y.x)\ y$. The following reduction sequence ensues:

$$\begin{aligned}(\lambda[I]x.\lambda[I,x]y.x)\ y &\rightharpoonup (I :: x \mapsto y)(\lambda[I :: x \mapsto x]y.x)\\ &\rightharpoonup \lambda[((I :: x \mapsto y)I) :: x \mapsto ((I :: x \mapsto y)x)]y.x\\ &\rightharpoonup \lambda[((I :: x \mapsto y)I) :: x \mapsto y]y.x\\ &\rightharpoonup \lambda[I :: x \mapsto y]y.x\end{aligned}$$

Having seen an example of a reduction sequence in Closure Calculus, it is now time to revisit one of two things that makes Closure Calculus unique - tagged applications. Tagged applications delays applications until suitable substitutions may be made. More importantly, tagged applications allows for differentiation between redexes and non-redexes. The only way to eliminate a tagged application is through application of an extension.

Example. Consider the term $(\lambda[I]x.x@x)\,a$. The reduction sequence is as follows:

$$\begin{aligned}(\lambda[I]x.x@x)\,a &\rightarrow (I :: x \mapsto a)(x@x)\\ &\rightarrow ((I :: x \mapsto a)x)\,(I :: x \mapsto a))\\ &\rightarrow (a\,(I :: x \mapsto a))\\ &\rightarrow (a\,a)\\ &\rightarrow (a@a)\end{aligned}$$

Something to note is that normal applications between two variables will be reduced to a tagged application. A tagged application ensures binding is stable, since reduction of the body may expose variables that were previously hidden. Thus terms can almost always be reduced to a normal form.

4.4 Normal Forms

The normal form of Closure Calculus is as defined in the earlier chapters: terms that are no longer contain any redex in its subterms. Unlike in the pure λ-calculus, there is only one kind of normal form in Closure Calculus and is given as follows:

$$n := x \mid n@n \mid \lambda[n]x.n \mid I \mid n :: x \mapsto n$$

(a) Normal Forms of CCN

$$n := \; J \mid R\,n \mid n@n \mid \lambda\,n\,n \mid I \mid (n, n)$$

(b) Normal Forms of CCDB

Figure 4.4.1: Normal Forms of Closure Calculus

The reduction rules of Closure Calculus is such that terms are almost always in NF. However, it should be understood that Closure Calculus itself is not strongly normalising nor is it weakly normalising. As will be shown in the next section, there exist terms in Closure Calculus that will never terminate.

A convenient way to think about the normalisation property in Closure Calculus is that programs and inputs will always be in NF. However, applicative reduction of terms in NF may yield non-terminating terms.

4.5 Confluence and Termination

The reduction rules of Closure Calculus are confluent (Jay 2017). Reduction is confluent in the sense of local confluence. This means Closure Calculus supports recursive terms. We may also write recursive programs in normal form. The following sequence of examples explain the preceding two sentences.

Example. Let $\mathbf{\Omega} := (\lambda[I]x.x@x)\,(\lambda[I]x.x@x)$. This is a self-applying (or recursive) term. The reduction sequence is as follows:

$$\begin{aligned}(\lambda[I]x.x@x)\,(\lambda[I]x.x@x) &\rightarrow (I :: x \mapsto (\lambda[I]x.x@x))\,(x@x)\\ &\rightarrow ((I :: x \mapsto (\lambda[I]x.x@x))\,x)\,((I :: x \mapsto (\lambda[I]x.x@x))\,x)\\ &\rightarrow (\lambda[I]x.x@x)\,((I :: x \mapsto (\lambda[I]x.x@x))\,x)\\ &\rightarrow (\lambda[I]x.x@x)\,(\lambda[I]x.x@x)\\ &\rightarrow \ldots\end{aligned}$$

Having seen an example of a recursive term in Closure Calculus, what does it mean when I wrote "we may also write recursive programs in normal form? The following four examples build on one another.

Example. Let $\mathbf{\Omega} := (\lambda[I]x.x@x)\,(\lambda[I]x.x@x)$; $\mathbf{K} := \lambda[I]x.\lambda[I,x]y.x\,y$ and $\mathbf{I} := \lambda[I]x.x$. The term $\mathbf{KI\Omega}$ will never terminate. Now, define an alternative version $\mathbf{\Omega}' := (\lambda[I]x.x@x)@(\lambda[I]x.x@x)$. $\mathbf{\Omega}'$ is recursive and in NF as there are no redexes. The term $\mathbf{KI\Omega}'$ terminates.

Is $\mathbf{\Omega}$ the same as $\mathbf{\Omega}'$? This question is about a possible confusion between the usual term application and tagged term application. A tagged application is not the same as a usual term application. $\mathbf{\Omega}$ is not the same as $\mathbf{\Omega}'$. The use of tagged application in $\mathbf{\Omega}'$ shows that we may elect to have a recursive term in NF.

One might complain that $\mathbf{KI\Omega}$ is not a recursive program in the traditional sense in that a recursive program is a function preceded by a fix point combinator. So let's look at one with Curry's Y combinator.

Example. Let $\mathbf{Y} := \lambda[I]f.(\lambda[I,f]x.f@(x@x))@(\lambda[I,f]x.f@(x@x))$. Observe that $\mathbf{Y}$ is in NF. The reduction sequence of $\mathbf{Y}g$ is as follows (note that the $\mathbf{\Omega}$ placeholder is not the same one as in the example above):

$$\begin{aligned}
\mathbf{Y}g &\to (I :: f \mapsto g)(\lambda[I,f]x.f@(x@x))@(\lambda[I,f]x.f@(x@x)) \\
&\to^* (\lambda[I :: f \mapsto g]x.f@(x@x))\,(\lambda[I :: f \mapsto g]x.f@(x@x)) \\
&\equiv \mathbf{\Omega}\,\mathbf{\Omega} \\
&\equiv (\lambda[I :: f \mapsto g]x.f@(x@x))\,\mathbf{\Omega} \\
&\to (I :: f \mapsto g :: x \mapsto \mathbf{\Omega})(f@(x@x)) \\
&\to^* g\,(\mathbf{\Omega}\,\mathbf{\Omega}) \\
&\equiv g\,((\lambda[I :: f \mapsto g]x.f@(x@x))\,\mathbf{\Omega}) \\
&\equiv g\,(g\,(\mathbf{\Omega}\,\mathbf{\Omega})) \\
&\to \ldots
\end{aligned}$$

The reduction of $\mathbf{Y}g$ never terminates.

Remark. Having $\mathbf{Y}$ being in NF is not particularly useful. A recursive program like addition would need to be shown to be in NF. An obvious candidate is the usual modification of the Y combinator into the Z combinator. However, this does not work either:

Example. Let $\mathbf{Z} := \lambda[I]f.(\lambda[I,f]x.f@(\lambda[I,f,x]y.x@x@y))@(\lambda[I,f]x.f@(\lambda[I,f,x]y.x@x@y))$. The reduction sequence of $\mathbf{Z}g$ is as follows:

$$\begin{aligned}
\mathbf{Z}g &\to (I :: f \mapsto g)\ (\lambda[I,f]x.f@(\lambda[I,f,x]y.x@x@y))@(\lambda[I,f]x.f@(\lambda[I,f,x]y.x@x@y)) \\
&\to^* (\lambda[I :: f \mapsto g]x.f@(\lambda[I,f,x]y.x@x@y))\ (\lambda[I :: f \mapsto g]x.f@(\lambda[I,f,x]y.x@x@y)) \\
&\equiv \mathbf{\Omega}\,\mathbf{\Omega} \\
&\equiv (\lambda[I :: f \mapsto g]x.f@(\lambda[I,f,x]y.x@x@y))\,\mathbf{\Omega} \\
&\to (I :: f \mapsto g :: x \mapsto \mathbf{\Omega})(f@(\lambda[I,f,x]y.x@x@y)) \\
&\to^* g@\lambda[I :: f \mapsto g :: x \mapsto \mathbf{\Omega}]y.x@x@y
\end{aligned}$$

Observe that the result is in NF, albeit a particularly useless one. Recall that a tagged application may only be removed with an extension. The usual way to get an extension would be via a closure. However, the result is a naked tagged application. If instead of g, an abstraction $\lambda[M]x.N$ is used, then the reduction may further continue. However it is unlikely one would get the desired result, which is a recursive program in normal form.

Remark. It is easy to confuse Closure Calculus for a more traditional λ-calculus with an eager strategy.

Instead, let us consider a variant of the Z combinator, in which y is placed just after x.

Example. Let $\mathbf{Z} := \lambda[I]f.\lambda[I,f]x.(\lambda[I,f,x]y.f@(x@x@y))@(\lambda[I,f,x]y.f@(x@x@y))$. The reduction sequence of $\mathbf{Z}g$ is as follows:

$$
\begin{aligned}
\mathbf{Z}g &\rightarrow (I :: f \mapsto g)\ \lambda[I,f]x.(\lambda[I,f,x]y.f@(x@x@y))@(\lambda[I,f,x]y.f@(x@x@y)) \\
&\rightarrow^* (\lambda[I :: f \mapsto g)x.(\lambda[I,f,x]y.f@(x@x@y)))\ (\lambda[I :: f \mapsto g)x.(\lambda[I,f,x]y.f@(x@x@y))) \\
&\equiv \mathbf{\Omega}\ \mathbf{\Omega} \\
&\equiv (\lambda[I :: f \mapsto g)x.(\lambda[I,f,x]y.f@(x@x@y)))\ \mathbf{\Omega} \\
&\rightarrow^* \lambda[I :: f \mapsto g :: x \mapsto \mathbf{\Omega}]y.f@(x@x@y)
\end{aligned}
$$

Here we can see that $\mathbf{Z}g$ is indeed in normal form waiting for an input so that it may further reduce. If g were another function, then this would be a recursive program.

The Z combinator presented here is a recursive program in the sense that it's a function proceeded by a fix point operator. However, it is a defective one, and cannot fully represent primitive recursive functions.

This may be remedied by using a Closure Calculus version of Turing's Y combinator, written as follows:

$$
\begin{aligned}
\omega &:= \lambda[I]z.\lambda[I,z]f.\lambda[I,z,f]x.f@(z@z@f)@x \\
\mathbf{Y} &:= \lambda[I :: z \mapsto \omega]f.\lambda[I,z,f]x.f@(z@z@f)@x
\end{aligned}
$$

Now all recursive programs may be written in normal form!

4.6 Evaluation and Operational Semantics

In Closure Calculus, evaluation is done by applying reduction rules of Closure Calculus on an expression of Closure Calculus terms. It should be noted that values in Closure Calculus is exactly that of its normal form, given in Figure 4.4.1.

Closure calculus represents a freedom from the tyranny of evaluation strategies - one simply needs to reduce the term to normal form for evaluation. There is no need for a call-by-name, call-by-value or any other evaluation strategies that may change its confluence properties.

The operational semantics of both flavours of Closure Calculus are given as follows:

$$\frac{}{x \Rightarrow x} \qquad \frac{}{I \Rightarrow I} \qquad \frac{s \Rightarrow v_1 \qquad t \Rightarrow v_2}{s@t \Rightarrow v_1@v2} \qquad \frac{\sigma \Rightarrow v_1 \qquad u \Rightarrow v_2}{\lambda[\sigma]x.u \Rightarrow \lambda[v_1]x.v_2}$$

$$\frac{s \Rightarrow v_1 \qquad t \Rightarrow v_2 \qquad v_1\ v_2 \rightharpoonup u \qquad u \Rightarrow v}{s\ t \Rightarrow v} \qquad \frac{\sigma \Rightarrow v_1 \qquad u \Rightarrow v_2}{\sigma :: x \mapsto u \Rightarrow v_1 :: x \mapsto v_2}$$

Figure 4.6.1: Operational Semantics of CCN

$$\frac{}{J \Rightarrow J} \quad \frac{t \Rightarrow v}{Rt \Rightarrow Rv} \qquad \frac{}{I \Rightarrow I} \qquad \frac{s \Rightarrow v_1 \qquad t \Rightarrow v_2}{s@t \Rightarrow v_1@v_2} \qquad \frac{\sigma \Rightarrow v_1 \qquad u \Rightarrow v_2}{\lambda\sigma u \Rightarrow \lambda v_1 v_2}$$

$$\frac{s \Rightarrow v_1 \qquad t \Rightarrow v_2 \qquad v_1\ v_2 \rightharpoonup u \qquad u \Rightarrow v}{s\ t \Rightarrow v} \qquad \frac{\sigma \Rightarrow v_1 \qquad \rho \Rightarrow v_2}{\sigma, \rho \Rightarrow v_1, v_2}$$

Figure 4.6.2: Operational Semantics of CCDB

Like the reduction rules, while there are many evaluation rules to Closure Calculus, they are straightforwards with no meta theory requirement.

4.7 Some Remarks on Closure Calculus

This chapter on Closure Calculus would be very much shorter if only one variant of Closure Calculus were discussed. Given the structures of this chapter and the two chapters preceding it are very similar, we may compare the structures of the chapters and come to the conclusion that Closure Calculus is indeed simpler than the pure λ-calculus.

The lack of required meta-operations that makes Closure Calculus simpler than the pure λ-calculus. The lack of meta theory leads to fewer notions of normal forms and fewer notions of evaluation strategies. On a whole Closure Calculus is simpler than the pure λ-calculus. In the next part, we'll see how a simpler calculus leads to simpler implementations.

Is Closure Calculus Strongly Normalising?

No. Reduction of terms does not guarantee a normal form - there exists terms that never terminate. However, recursive programs can be written in normal form without resorting to variations such as head normal forms. This can lead to some amount of confusion.

Translating Pure λ-Calculus to Closure Calculus

There exists a straightforwards translation of the pure λ-calculus to CCN. However, the inverse is not true, as there can be terms defined in Closure Calculus that cannot be defined in λ-calculus. In this section, a method of converting from pure λ-calculus to CCN is defined.

The translation method is inductively defined:

$$
\begin{aligned}
&\text{let } r, s, t \in Terms_{\lambda};\\
&\text{let } u, v, w \in Terms_{CCN};\\
[\![x_{\lambda}]\!] &= x_{CCN}\\
[\![s\ t]\!] &= \begin{cases} [\![s]\!]@[\![t]\!] & \text{if } s\ t \text{ is under } \lambda\\ [\![s]\!]\ [\![t]\!] & \text{otherwise}\end{cases}\\
[\![\lambda x.t]\!] &= \lambda[I :: y \mapsto y]x.[\![t]\!]\ \forall y \in BV(t)
\end{aligned}
$$

Translating terms of the pure λ-calculus to that of Closure Calculus must ensure that reduction is preserved. This is to say that $(\lambda x.x)\,M$ which reduces to Mmust be translated to a term in Closure Calculus so that the result is the Closure Calculus equivalent, $[\![M]\!]$.

The translation of terms in pure λ-calculus to Closure Calculus is to be considered on two levels. They may be conceptually thought of as program and its application. These concepts will be explored further in the next chapter. Briefly, a program is a description of the computation (reductions in the context of lambda calculi) to be done. When translating programs of the pure λ-calculus, any application $(t\ u)$ are translated into tagged applications in CCN. When considering the applicative case - i.e. an application of a program to its input, then applications in λ-calculus may be replaced by applications in CCN.

Because free variables are universally free in CCN terms, care must be taken when translating abstractions from λ-calculus to Closure Calculus. The translation scheme defined yields larger CCN terms, but they still reduce as expected

Example. $[\![\lambda x.\lambda x.x]\!] = \lambda[I :: x \mapsto x]x.\lambda[I :: x \mapsto x]x.x$. This is contrary to the usual translations encountered in this chapter, which would read $\lambda[I]x.\lambda[I :: x \mapsto x]x.x$. Nonetheless, reduction proceeds as expected. The λ-calculus term, when applied to inputs, v and w, proceeds as follows:

$$
\begin{aligned}
(\lambda x.\lambda x.x)\,v\,w &\to (\lambda x.x)\,w\\
&\to w
\end{aligned}
$$

The translated terms would yield the following reduction sequence:

$$
\begin{aligned}
(\lambda[I :: x \mapsto x]x.\lambda[I :: x \mapsto x]x.x)\,v\,w &\to (I :: x \mapsto v)(\lambda[I :: x \mapsto x]x.x)\,w\\
&\to ((I :: x \mapsto x :: x \mapsto v)(\lambda[I :: x \mapsto x]x.x))\,w\\
&\to^{*} (\lambda[I :: x \mapsto (I :: x \mapsto x :: x \mapsto v)x])x.x)\,w\\
&\to (\lambda[I :: x \mapsto v]x.x)\,w\\
&\to^{*} w
\end{aligned}
$$

Translating from one language to another is the core of what programming language compilation is all about. The next chapter explores that notion in detail.

Part II

Implementation and Comparisons

Chapter 5

On Implementations and Fair Comparisons

The thesis of this report asserts that Closure Calculus is better than the pure λ-calculus. However, in comparing the lambda calculi, a number of subtle issues arise and must be considered. This chapter deals with issues surrounding the comparison and implementation of the lambda calculi.

Comparisons are done only on cases where complete evaluation is possible. The first section justifies such a comparison. Then, the intricacies of implementing an operational semantics is explored in the following two sections - first a general overview of implementing operational semantics, then a description of the implementations. Next, a consideration of δ-rules is given. Finally, the tests and benchmark suite are described, along with justifications thereof.

5.1 Why Compare Reductions to Normal Forms?

Although this report is mainly focused on comparing Closure Calculus to the pure λ-calculus, the comparison is informed by the desire to show Closure Calculus is the superior lambda calculus to be used as the theoretical basis of functional programming languages. Confusion is easy when discussing lambda calculi in the context of programming languages, so care is given to the rationale. A brief paragraph explaining the subtleties follows.

A program is a body of structured text comprising of symbols whose meanings (semantics) may be interpreted at whim of the language implementor. A lambda calculus is like a programming language - it is a body of structured text comprising of symbols. Its semantics however, is well defined. The notion of an operational semantics had been introduced in the previous chapters. When a programming language has a theoretical basis in a lambda calculus, this means the operational semantics of the programming language is an extension of the operational semantics of the lambda calculus.

Programming language implementors would be interested in the comparisons in this report insofar as operational semantics are concerned.

The pure λ-calculus has multiple operational semantics - call-by-name (CBN), call-by-need, call-by-value (CBV), applicative order and normal order were the ones introduced in this report. Each of these operational semantics defines a variant of a normal form as a value. For example, in the operational semantics associated with CBN, values are in head normal form.

Closure Calculus however, only has one operational semantics defined so far. The values in an evaluation context are normal forms in the strictest sense of the word. The notion of head reduction in Closure Calculus has not been studied. It may be argued that the virtues of Closure Calculus as a lambda calculus negates the need for head reduction. However, that will be left as future work. For now, let us delve into the finer points of making comparisons between the lambda calculi.

We say a pure λ-calculus term t_Λ is analogous with a Closure Calculus term t_{CC} if by applying the translation in Subsection 4.7 of Chapter 4 to t_Λ, we obtain t_{CC}. This is commonly called a simulation. However, we avoid the term simulation as that comes with implications that may not be justified. Further discussions on simulations are given in the next section.

Having introduced the notion of two terms being analogous, it is now shown that direct analogies are a little difficult as shown by the following example.

Example. Consider the reductions of the following term in pure λ-calculus $(\lambda x.\lambda y.x\,y)\;((\lambda z.z)\,(\lambda a.a))$, and its Closure Calculus translation, $(\lambda[I]x.\lambda[I,x]y.x@y)\;((\lambda[I]z.z)\,(\lambda[I]a.a))$. The subterms in red are programs that each expect two inputs. The subterm in blue is the first input.

The operational semantics of the pure λ-calculus with a CBV evaluation strategy is closest to the operational semantics of Closure Calculus so we compare them presently. Under a CBV evaluation strategy, the pure λ-calculus term reduces to $\lambda y.(\lambda a.a)\,y$ while the Closure Calculus term reduces to $\lambda[I :: x \mapsto (\lambda[I]a.a)]y.x@y$. These two terms are not analogous. A translation of the pure λ-calculus result to Closure Calculus does not yield the Closure Calculus result.

That pure λ-calculus terms are not analogous to Closure Calculus terms extends mostly on partially evaluated terms. Complete evaluation of the terms usually yield a value that is analogous in Closure Calculus. There are exceptions. These are discussed and justified in Chapter 8. Observe that the end result of complete evaluations are always in normal forms, so that the a translation of pure λ-calculus result will yield the Closure Calculus result.

It is on this basis that a fair comparison between the lambda calculi may be made. Happily, this plays well into the aims of programming language implementors. After all, the purpose of most programs are to completely evaluate its inputs.[1].

[1]For now, let's put aside considerations for Futamura projections.

5.2 On the Implementation of Operational Semantics in General

The modern computer are based on the von Neumann architecture. Without loss of generality, the von Neumann architecture is a realisation of a Universal Turing Machine (UTM) with some limitations imposed. A UTM is able to simulate any Turing Machine.

Informally we say two systems are equivalent if they can simulate each other. The pure λ-calculus and Turing machines simulate each other, as shown in the addendum of Alan Turing's seminal paper on computation (Turing 1937). This equivalence is usually called the Church-Turing thesis. Unfortunately, due to the informal nature of stating such an equivalence, it has led to some confusion about what the thesis is actually about. Jay and Vergara (2014) provides a good overview of the confusion and the consequence thereof. A brief treatment of the confusion is provided in the section of the appendix titled On the Confusion of the Church-Turing Thesis. For the rest of this report, we will assume that the Church-Turing thesis holds, at least up to computation of numerical functions.

Programming languages provide operational semantics such that the desired behaviours of the language may be correctly simulated by a computer. To proceed, I shall have to be clearer on what some words mean.

Let there be two systems of computation, $\mathcal{V}$ and Λ, representing a von Neumann machine with random access memory (as modern computers are) and a λ-calculus respectively. The term of Λ represented in $\mathcal{V}$ and vice-versa is called a REPRESENTATION. An encoding ρ is an injective function used to encode terms in $Dom(\Lambda)$ to terms in $Dom(\mathcal{V})$, written $\rho : \Lambda \to \mathcal{V}$. An encoding is usually in form of a scheme, to be instantiated on specific terms.

Encoding static terms of Λ to its representation is trivial. Using the pure λ-calculus as an example Λ, we may elect to minimally represent the term $\lambda x.x$ with 4 bits of machine memory: 0010 as shown by Tromp (2018).

Static terms are uninteresting. Interesting things happen when a term is applied to another. To fully simulate a system we must also simulate the behaviour of the reduction rules. More specifically, we must also encode the operational semantics of Λ in $\mathcal{V}$. The encoded representation is called an EXECUTION MODEL of Λ on machine $\mathcal{V}$.

A full description of machine $\mathcal{V}$ may be too complex for the purposes of creating an execution model of Λ. For example, we would not want to be discussing implementing a λ-calculus in terms of which logic gate to activate. So instead, we abstract over the finer grained details to arrive at a description of a machine with just enough details to implement the execution model of Λ. This is called an ABSTRACT MACHINE.

Function application in Λ is instantaneous. However, with the advent of stored-program computers, the notion of a function is somewhat broken. Accordingly we may now separate the notion of a function into its static component and its dynamic component. Its static component is what is written, while its dynamic component is what happens when a function is applied. In modelling a λ-calculus as a programming language, we can then split the reduction phases into two: compile phase and runtime phase.

Example. Consider the term $\lambda x.((\lambda y.y)x)$. It is ready to take an input at run time. While nothing is wrong with directly encoding the term into a von Neumann machine, it would be very inefficient to reduce everything at run time. Instead, we would like to reduce to normal form at compile time, so that at run time, minimal amounts of work would be done. A very good compiler would reduce the term, even under the λ, such that at run time, $\lambda x.x$ is the program being run. Such a compiler is quite hard to write, owing to ill-defined notions of reduction on open terms.

Hence we will need to have the term reducer at both compile time and at run time. A RUNTIME SYSTEM is a collection of functions that allow for the operational semantics of the language to be used at run time. This usually includes memory allocation functions and garbage collection functions.

Having introduced to the the notion of implementing an operational semantics by simulation into a von Neumann machine, it should be noted that implementations of operational semantics is still very much an open research topic - there is no generally agreed-upon best way of implementing an operational semantics. The choice of implementing the operational semantics of the lambda calculi in this report is geared towards being able to make a fair statement about the performance characteristics of the lambda calculi.

5.3 On the Implementation of the Lambda Calculi

The calculi are implemented in an imperative programming language - Go. A primer in the programming language is given in the Appendix. Thus this section concerns the design of the interpreters.

The usual choice to implement operational semantics of lambda calculi are functional programming languages like Haskell or OCaml. Go was chosen as its syntax and semantics are closer to the underlying semantics of the underlying machine. Thus an implementation of an operational semantics in form of an interpreter is a realistic representation of an implementation of the operational semantics on a von Neumann machine. Further, Go allows for careful management of the data structures used such that the interpreters are not biased one way or another.

The interpreters are designed so that the data structures used are shared as much as possible. This ensures any differences between the performance of the interpreters will not be due to differing designs. As such, the data structures representing a term of a lambda calculus are share between the interpreters. The following section describe first the data structures that are used, followed by a section describing management of memory. Then a description of the metadata that is tracked by the interpreters is given. Finally, the interpreters are described.

The source code may be found at `https://github.com/chewxy/cccombos`. All subdirectories mentioned in this report is relative to the root directory that is the repository. The subdirectories `cbv`, `cbn`, `ccn` and `ccdb` contain the respective interpreters. In each interpreter's subdirectory, there are one or more `prelude` subdirectories. These `prelude` subdirectories define common terms for the purposes of the benchmarks and tests.

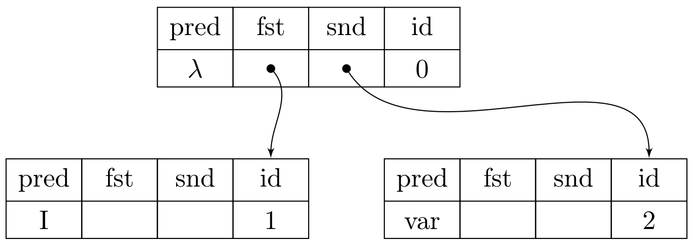
Representing Terms

A term is represented by a struct. The definition can be found in `internal/termso1/manager.go`. Alternative representations are discussed in the Appendix. It's defined as follows.

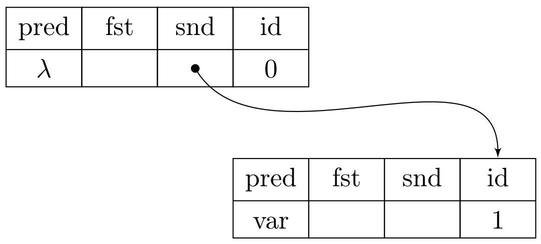
```
type Term struct {
        pred  Predicate
        fst   Ptr
        snd   Ptr
        id    Ptr
}
```

A `Predicate` is a label denoting the kind of term. The complete listing of predicates can be found in `internal/termso1/predicate.go`.

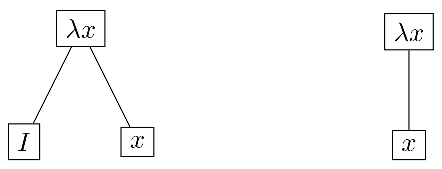
`fst` and `snd` are pointers to other terms, while `id` is the given unique identity for a term. The `id` field is part of the required book-keeping when implementing an interpreter, and is not strictly part of the term. All fields are represented by 4 bytes. We may visualise a CCN term, $\lambda[I]x.x$ as such:



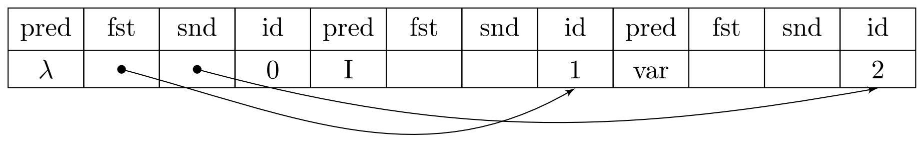
Conversely, the pure λ-calculus term $\lambda x.x$ is visualised as such:



These structures corresponds well to natural visualisations of the terms as trees:



In practice the objects are allocated as closely as possible for performance purposes, so they actually look more like this (for the CCN term $\lambda[I]x.x$):



pred	fst	snd	id	pred	fst	snd	id	pred	fst	snd	id
λ	•	•	0	I			1	var			2

The observant reader will note that there are no names in the `Term` struct. A λ binds a name. Without a name, how can there be computation?

The spartan nature of the `Term` struct was chosen by design. The rationale for the design shall presently be given, followed by an explanation of handling of names.

First, consider the evaluation of the a term of the pure λ-calculus: $(\lambda x.M)y$. Assuming the names are normalised such that β-reduction may proceed (that is to say, all checks and α-conversions have been done), full β-reduction requires traversing the entirety of M, in order to replace all matching x in M with y. Indeed, when evaluating a term of a λ-calculus, the majority of operations are tree-traversal operations. Hence, an easily traversable data structured is preferred for performance sake.

Next, consider the depth-first traversal of any given term. Every algorithm that traverses a tree data structure depth first uses a stack. The stack may be implicit, as is the case when recursively traversing the structure. When using recursive functions to traverse a tree depth first, the stack provided by the programming language runtime is used. The stack may not be available to the programmer for manipulation, but it is nonetheless there.

When traversing a term at depth N + 1, an interpreter would have to hold the data of N previous levels in its stack. We are hence presented with two design choices. One, allocate the object representing a term on the heap, and keep a pointer to it on the stack. Two, allocate a sufficiently small object on the stack. The benefits of the former is that we are able to store more (pointers to) terms on the stack, allowing for longer computations. However, this comes at a cost: allocating objects on the heap creates plenty of garbage. Most garbage collecting algorithms scan for pointers on the stack in order to mark objects as being alive or dead. Dead objects are scheduled for collection. This imposes a large overhead at runtime. The latter design choice sidesteps this issue by not having to allocate any objects on the heap. The cost of doing so is that we may not be able to store as many terms on the stack as we would have had the first design choice been chosen.

There exists a tension between performance and maximal complexity of programs that are able to be run. This tension is hard to resolve. In most compilers, an analysis pass is performed to analyse whether an object is suitable to be destined for the stack or heap. This is called escape analysis[2].

Building a compiler with many analysis phases is not in scope of the project, hence a decision was made to optimise to use the stack more. This decision is further bolstered by the fact that modern computers have very large stacks - much more than any program that can realistically be written directly in the calculi. Further, the `Term` struct takes 2 machine words on a modern machine, while a pointer to the heap takes 1 machine word. The decision is obvious.

So how are the names managed?

Names are managed through a another structure, the `Datum` structure, defined as follows:

```
type Datum struct {
        Name, MetaName string
}
```

The use of the `Datum` struct, the `Term` struct are very much tied into the memory manager. The following section on the memory manager clarifies the use of the `Datum` structure.

The Memory Manager

The memory manager is a key component in the design of the interpreters of the calculi. Go is a programming language with managed memory. In general, managed memory makes for a safer programming language experience. Like most programming languages with managed memory, Go's garbage collector algorithm is non-deterministic at compile time. Therefore, the garbage collector may interfere with benchmarking the performance of the various calculi. We wish to avert that, therefore, the use of a manual memory manager is required. Happily, the design with the memory manager reinforces the design for representing terms.

The `Manager` struct, found in `internal/termso1/manager.go` represents the memory manager. A brief description of the fields is presented after the

[2]At the time of writing, Rust has one of the most advanced escape analysis in production-ready programming languages, albeit very much human aided (in the sense that one needs to "fight the borrow checker"). Interestingly the GHC compiler for Haskell does very minimal escape analysis, preferring to allocate large portions of objects on the heap. Unsurprisingly, the garbage collector of GHC is one of the primary complaints against it.

definition.

```
type Manager struct {
        terms    []Term
        data     []Datum
        freelist map[Ptr]struct{}
}
```

The major design pattern for the memory manager is an arena-based management with a very simple allocation strategy. The common name for the design and data access strategy is a Structure of Arrays. This design splits a term into two parts (`Term` and `Datum`) based on use frequency. Observe that in reductions of terms, the name is not often accessed or used. So the `Datum` structure represents less-frequently-used data. This way, the reductions are optimized for performance.

`terms` is an array of `Term` objects, serving as the arena of memory dedicated to storing `Term` objects. The `id` field of a `Term` object is simply the index of the object within the memory manager. For every `Term` object, there is a corresponding `Datum` object that is created and stored in the `data` field. One may think of that as a separate region of memory that is earmarked for use specifically to objects of the `Datum` type. Hence the datum of the ith `Term` object is also the ith `Datum` object. Last, the `freelist` field is a set of `Term` objects that are free for use. Further explanations on how the memory manager works will clarify the use of `freelist` .

The role of the memory manager is to manage the `Term` objects. The memory manager allocates `Term` objects. When it needs to allocate a new `Term` object, it first checks to see if there are any already-allocated objects that are available for use in the `freelist`. It also manages the freeing of objects. Instead of returning memory to the system, the memory manager clears the fields of a `Term` object, adds the `id` of the free object back into the `freelist`. This way, the majority of the work done for allocating and freeing objects in memory is constant, allowing for a fairer comparison between the calculi.

Metadata

One of the comparisons to be done is the number of reduction operations and meta-operations each calculus performs. To do that, we would need to keep count of each operation. Given that this counting is global (i.e. all interpreters in this project have this requirement), the data structure for the bookkeeping of counting operations is shared across all interpreters. It's defined in `internal/meta/meta.go`. A brief description of each field follows the definition:

```
type Metadata struct {
        // Counting
        RuleMatch [16]int
        MetaOp    [maxMeta]int // cannot be summed!
        Rules int

        // Logging
        Logger        *log.Logger
        RestrictDepth int
        depth         int
}
```

`RuleMatch` is an array of integers. The index of the array indicates the rule. The values indicate the number of times a rule has fired. Because no calculus has more than 16 rules, a finite array of 16 is able to account for all the rules of all the calculi considered. `MetaOp` is another array of integers, whose purpose is to keep count the number of meta-operation rules fired. The counting of meta-operations overlap, and hence the counts cannot be summed. The following are the meta-operations that are being counted:

- `VarEq` - Variable equality checks
- `Sub` - Substitutions
- `TermEq` - Term equality
- `Fresh` - Fresh variables generated
- `Capture` - Variable capture checks
- `UnionFV` - Unions of free variable sets

Additionally `Rules` indicate how many reduction rules an interpreter has. The CCN interpreter has 10 rules, CCDB has 11, while the pure λ-calculus interpreters has only 2 or 3 rules. This is required because the `RuleMatch` field is a fixed-size array that is fixed at compile time. The size is fixed to be larger than the number of rules for either calculi. As an implementation detail, 16 was chosen to fit within the memory lines of the processor, leading to fewer cache misses, and hence higher performance.

The remaining three fields, `Logger`, `RestrictDepth` and `depth` concern the logging of evaluations. The `Logger` field points to an output file to write the logs to, while `RestrictDepth` controls the logging up to a certain depth of evaluation. The `depth` field stores the current depth of the evaluation.

The Interpreters

Now we are ready to discuss the interpreters. Each calculus has its own interpreter, each in their own package. All interpreters share a common structure - that is, a composition of `Metadata` and `Manager`. There are also, for each interpreter, configuration related fields. All interpreters are defined in a fashion similar to this:

```
type Interpreter struct {
        Metadata
        Manager

        // other fields for configuration of the interpreter
}
```

The structural operational semantics of each calculus provides the rules for interpretation. One may think of the structural operational semantics as a very concise definition of the interpreter. The structural operational semantics for the calculi, previously introduced in the previous chapters are repeated here for comparison:

$$\frac{}{x \Rightarrow x} \qquad \frac{}{I \Rightarrow I} \qquad \frac{s \Rightarrow v_1 \quad t \Rightarrow v_2}{s@t \Rightarrow v_1@v2} \qquad \frac{\sigma \Rightarrow v_1 \quad u \Rightarrow v_2}{\lambda[\sigma]x.u \Rightarrow \lambda[v_1]x.v_2}$$

$$\frac{s \Rightarrow v_1 \quad t \Rightarrow v_2 \quad v_1\ v_2 \rightharpoonup u \quad u \Rightarrow v}{s\ t \Rightarrow v} \qquad \frac{\sigma \Rightarrow v_1 \quad u \Rightarrow v_2}{\sigma :: x \mapsto u \Rightarrow v_1 :: x \mapsto v_2}$$

Figure 5.3.1: Operational Semantics of CCN

$$\frac{}{J \Rightarrow J} \quad \frac{t \Rightarrow v}{Rt \Rightarrow Rv} \qquad \frac{}{I \Rightarrow I} \qquad \frac{s \Rightarrow v_1 \quad t \Rightarrow v_2}{s@t \Rightarrow v_1@v_2} \qquad \frac{\sigma \Rightarrow v_1 \quad u \Rightarrow v_2}{\lambda\sigma u \Rightarrow \lambda v_1 v_2}$$

$$\frac{s \Rightarrow v_1 \quad t \Rightarrow v_2 \quad v_1\ v_2 \rightharpoonup u \quad u \Rightarrow v}{s\ t \Rightarrow v} \qquad \frac{\sigma \Rightarrow v_1 \quad \rho \Rightarrow v_2}{\sigma, \rho \Rightarrow v_1, v_2}$$

Figure 5.3.2: Operational Semantics of CCDB

$$\frac{}{x \Rightarrow x} \quad \frac{s \Rightarrow v}{s\ t \Rightarrow v\ t} \quad \frac{t \Rightarrow v}{s\ t \Rightarrow s\ v} \quad \frac{}{(\lambda x.t)v \Rightarrow \{v/x\}t}$$

Figure 5.3.3: Operational Semantics of Pure λ-Calculus (CBV)

$$\frac{}{x \Rightarrow x} \quad \frac{s \Rightarrow v}{s\ t \Rightarrow v\ t} \quad \frac{}{(\lambda x.t)v \Rightarrow \{v/x\}t}$$

Figure 5.3.4: Operational Semantics of call-by-name λ-calculus

The interpreter rules are implemented as methods on the `Interpreter` object. Implementation is rather straightforwards. The rules are implemented as a switch table, switching on the given predicate. Each inference rule's premise is the switch condition, and the conclusion of each inference rule is the body.

Upon first glance, it would appear that the operational semantics for CCN and CCDB are far more complex than that of the pure λ-calculus. What is left out of the pure λ-calculus' operational semantics is the meta-operations required. An honest implementation would count both parts of the interpreter, for meta-operations too take computational resources.

5.4 δ-Terms and δ-Rules

While both the pure λ-calculus and Closure Calculus are Turing complete - in the sense that both calculi are able to define computable functions on natural numbers - modern computers have built-in support for numbers and their operations. For the purposes of benchmarks, the calculi are extended with δ-rules to take advantage of these built-in operations.

New terms are added to each of the calculi, given as follows:

$$
\begin{array}{lr}
s, t, u := \ldots & \text{(previous definitions from each individual calculus)} \\
\quad \mid \mathbb{N} & \text{(natural numbers)} \\
\quad \mid \texttt{True} \mid \texttt{False} & \text{(booleans)} \\
\quad \mid t + u \mid t - u \mid t \times u \mid t \div u \mid t \ \texttt{mod} \ u & \text{(arithmetic)} \\
\quad \mid t < u \mid t \leq u \mid t > u \mid t \geq u \mid t = u & \text{(comparisons)} \\
\quad \mid \texttt{if} \ t \ \texttt{then} \ u \ \texttt{else} \ s & \text{(conditional)}
\end{array}
$$

All δ-terms are representable by a `Term`. New `Predicate`s are defined for each of the δ-term. For values (natural numbers and booleans), the `Fst` field of a `Term` is used to store the value. The natural numbers are implemented as finite, 32-bit unsigned integers. Booleans are also represented by 32-bit unsigned integers where 1 represents True and 0 represents False.

Additional reduction rules, the δ-rules are defined and given here as operational semantics as follows ($[\![\cdot]\!]$ indicates the results of the built in operations - arithmetic operations return natural numbers and comparison operations return boolean values):

$$
\frac{a, b \Rightarrow \mathbb{N}}{a + b \Rightarrow [\![a + b]\!]} \qquad
\frac{a, b \Rightarrow \mathbb{N}}{a - b \Rightarrow [\![a - b]\!]} \qquad
\frac{a, b \Rightarrow \mathbb{N}}{a \times b \Rightarrow [\![a \times b]\!]} \qquad
\frac{a, b \Rightarrow \mathbb{N} \qquad b > 0}{a \div b \Rightarrow [\![a \div b]\!]}
$$

$$
\frac{a, b \Rightarrow \mathbb{N} \qquad b > 0}{a \ \texttt{mod} \ b \Rightarrow [\![a \ \texttt{mod} \ b]\!]} \qquad
\frac{t \Rightarrow \texttt{True}}{\texttt{if} \ t \ \texttt{then} \ u \ \texttt{else} \ s \Rightarrow u} \qquad
\frac{t \Rightarrow \texttt{False}}{\texttt{if} \ t \ \texttt{then} \ u \ \texttt{else} \ s \Rightarrow s}
$$

$$
\frac{a, b \Rightarrow \mathbb{N}}{a < b \Rightarrow [\![a < b]\!]} \qquad
\frac{a, b \Rightarrow \mathbb{N}}{a \leq b \Rightarrow [\![a \leq b]\!]} \qquad
\frac{a, b \Rightarrow \mathbb{N}}{a > b \Rightarrow [\![a > b]\!]} \qquad
\frac{a, b \Rightarrow \mathbb{N}}{a \geq b \Rightarrow [\![a \geq b]\!]}
$$

These structural operational semantics are also implemented in the same way as the operational semantics of each calculus - as a switch table.

5.5 The Test and Benchmark Programs

The following programs were created as unit tests to ensure that there is a correspondence between the values of λ-calculus and the values of Closure Calculus. As such, programs and data may be listed down in the pure λ-calculus form. Conversions to Closure Calculus follows algorithm outlined in Chapter 4. Items marked with a * are also used in benchmarks.

1. $(\lambda x.x)\, a$
2. $(\lambda x.x\ y)\, a$
3. $(\lambda x.\lambda y.x)\, a$
4. $(\lambda x.\lambda y.y)\, a$
5. $(\lambda x.\lambda x.x)\, a$
6. $(\lambda a.\lambda a.a)\, a$
7. SKK*
8. Addition with Scott Numerals *
9. `fib` - nth Fibonacci number (with δ-rules for natural numbers) *
10. `primes` - Prime Filter (with δ-rules of natural numbers)*
11. `tak` - Tak function (with δ-rules of natural numbers)*

A brief description of the programs and their reasoning follows.

The first six programs and input pairs were used as sanity checks to ensure that each interpreter reduces the terms correctly as expected. Programs 1 to 4 are basic reduction exercises. Programs 5 and 6 test an interpreter's ability to handle confusion of names, either via meta-operations (pure λ-calculus) or the calculus itself (Closure Calculus). Name clashes are further tested in programs 7 to 11, where repeat variable names are used frequently.

SKK is a program that is best expressed in combinatory logic: $SKba$, where their combinators are written in the calculi. Addition is self-explanatory. It's a program that apply to the Scott encoded numerals. These are basic operations, to be contrasted against the more complex programs described below. Benchmarking basic programs give a baseline to compare and understand the results.

`fib` calculates the nth number of the Fibonacci sequence, where n is the parameter denoting the input to the program. Support for a computer representation of natural numbers in form of 32-bit unsigned integer is provided by means of extending the calculi with δ-rules. Fibonacci functions are used for benchmarks because it tests the ability of an interpreter to handle large amounts of closures.

`primes` is an implementation of the Sieve of Eratosthenes. Specifically, the function used for benchmarking is the version taken directly from the Haskell homepage (Haskell Language Authors 2014), translated into pure λ-calculus and Closure Calculus. This program is good for benchmarking as it is a good proxy for real-life programs.

`tak` is an implementation of the Tak function. The Tak function is traditionally used to benchmark handling of recursion (Knuth 1991, 1997).

A Note On The λ-encodings

The benchmarks involve data structures that have to be encoded into the lambda calculus under test. Here a rationale is given for the choice.

Lists and tuples are encoded such that they are identified by their right folds. The Church encoding, the Scott encoding, and the Parigot encoding of lists were also considered. However, either choice would bias towards one calculus or another. For example, comparing the performance of an interpreter using four encoding styles (Church, Scott, Parigot, right folds), the pure λ-calculus interpreter would perform better on the Church and Parigot encodings, while the Closure Calculus interpreter would perform better on the Scott encoded list. Thus, the "middle ground" was chosen as a fair comparison can then be had. More on the encoding of lists is explored in Chapter 8.

While the encodings for lists and tuples offer many choices, the choices of encoding a fix point combinator is somewhat limited.

For the pure λ-calculus benchmarks, the usual λ-encoding of the Y combinator, $\lambda p.(\lambda x.f\,(x\,x))(\lambda x.f\,(x\,x))$ is used, while the Y_2 combinator is used for Closure Calculus. The pure λ-calculus equivalent of the Y_2 combinator works as well. However, it would present an unbalanced view of the results as it would only work under one evaluation strategy. This is to say, the usual encoding of the Y combinator yields fewer reduction steps than the pure λ-calculus version of the Y_2 combinator. It is tempting to make the fix point combinator a δ-term with a corresponding δ-rule. This would not yield an interesting result, as most of the reductions are due to reductions while recursing.

How the Benchmark Programs Are Derived

This section discusses how each of the benchmark programs came to be. The running example for this section will be the `fib` program since it is the simplest.

First, the program is written in Haskell:

```
fib :: (Num t, Num a, Eq a) => a -> t
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Observe that this function is recursive, but was written without the use of a fixpoint operator. So this program is converted into one which explicitly uses a fixpoint operator. Instead of using the built in version of `fix`, write `y` such that it resembles the usual encoding of the Y combinator in λ-calculus (`newtype Mu` is required to sidestep some type checking issues):

```
newtype Mu a = Mu ( Mu a -> a)
y f = (\h -> h $ Mu h) (\x -> f . (\(Mu g) -> g) x $ x)

fibF :: (Eq a , Num a , Num t) => (a -> t) -> a -> t
fibF rec n = if n == 0 then 0
                else if n == 1 then 1
                else rec (n -1) + rec (n - 2)

fib :: (Num t , Num a , Eq a) => a -> t
fib = y fibF
```

This is sufficiently close enough to the pure λ-calculus version:

$$\begin{aligned}
\mathbf{Y} &:= \lambda f.(\lambda x.f\,(x\,x))\,f.(\lambda x.f\,(x\,x))\\
\mathbf{fibF} &:= \lambda rec.\lambda n.if\ n = 0\ then\ \lambda x.0\\
&\qquad\qquad else\ (if\ n = 1\ then\ \lambda x.1)\\
&\qquad\qquad else\ \lambda x.(rec\,((n-1) + (rec\,(\,n-2)))\\
\mathbf{fib} &:= (\mathbf{Y}\,\mathbf{fibF})
\end{aligned}$$

Finally this is translated to Closure Calculus, with $\mathbf{Y_2}$ being previously defined in Chapter 4:

$$\begin{aligned}
\mathbf{fibF} &:= \lambda[I]rec.\lambda[I, rec]n.if\ n = 0then\ \lambda[I]x.0\\
&\qquad\qquad else\ (if(\ n = 1\ then\ \lambda[I]x.1)\\
&\qquad\qquad else\ \lambda[I, rec]x.rec@(n-1) + (rec@(n-2))))@x\\
\mathbf{fib} &:= (\mathbf{Y_2}\,\mathbf{fibF})
\end{aligned}$$

The other programs are translated in a similar fashion. The next section presents the results and analysis.

Chapter 6

Closure Calculus is Simpler

This chapter compares Closure Calculus to the pure λ-calculus on their simplicity. This is done by first by comparing the relative simplicity of the lambda calculi. Then this abstract notion is made concrete - a comparison of the implementations of both calculi is then done. A brief introduction to the metrics of complexity is first given, then it is shown that Closure Calculus is simpler to implement.

6.1 On the Simplicity of the Lambda Calculi

$$\begin{aligned} (\lambda x.t)\ u &\rightharpoonup \{u/x\}t \\ &\quad \text{if } u\ \notin FV(t) \cup \{x\} \end{aligned}$$

(a) Pure λ

$$\begin{aligned} x\,t &\rightharpoonup x@t \\ (s@t)\,u &\rightharpoonup (s@t)@u \\ (\lambda[\sigma]x.t)\,u &\rightharpoonup (\sigma :: x \mapsto u)\,t \\ I\,u &\rightharpoonup u \\ (\sigma :: x \mapsto t)\,x &\rightharpoonup t \text{ , if } z = x \\ (\sigma :: x \mapsto t)\,z &\rightharpoonup \sigma\,z \text{ , if } z \neq x \\ (\sigma :: x \mapsto t)\,(u@v) &\rightharpoonup ((\sigma :: x \mapsto t)\,u)\,((\sigma :: x \mapsto t)\,v) \\ (\sigma :: x \mapsto t)\,(\lambda[\rho]z.t) &\rightharpoonup \lambda[(\sigma :: x \mapsto t)\,\rho]z.t \\ (\sigma :: x \mapsto t)\,I &\rightharpoonup I \\ (\sigma :: x \mapsto t)\,(\rho :: z \mapsto u) &\rightharpoonup (\sigma :: x \mapsto t)\,\rho :: z \mapsto (\sigma :: x \mapsto t)\,v \end{aligned}$$

(b) CCN

$$\begin{aligned} J\,t &\rightharpoonup J@t \\ Rtu &\rightharpoonup Rt@u \\ (r@t)u &\rightharpoonup (r@t)@u \\ (\lambda st)u &\rightharpoonup (u,s)t \\ It &\rightharpoonup t \\ (u,s)J &\rightharpoonup u \\ (u,s)(Rt) &\rightharpoonup st \\ (u,s)(r@t) &\rightharpoonup (u,s)r((u,s)t) \\ (u,s)\lambda r\,t &\rightharpoonup \lambda((u,s)r)t \\ (u,s)I &\rightharpoonup I \\ (u,s)(r,t) &\rightharpoonup ((u,s)r,(u,s)t) \end{aligned}$$

(c) CCDB

Figure 6.1.1: Reduction Rules

Figure 6.1.1 collates the reduction rules of the pure λ-calculus, CCN and CCDB. At first blush, the reduction rules of the pure λ-calculus is far simpler than that of either flavours of Closure Calculus. It is true and obvious that 1 rule is fewer than 10 or 11 rules. A claim to be made here is that the reduction rules of CCN and CCDB are in fact simpler than that of the pure λ-calculus. This is a contentious claim, and we must proceed with care on the reasoning, laid out below.

Observe that the side condition - "if $x \notin FV(t) \cup \{x\}$" - hides much of the complexity of the pure λ-calculus. When viewed from an implementation point of view, the reduction rules of the pure λ-calculus appears incomplete, and offers much latitude to the implementor.

Recall that the pure λ-calculus was introduced in Chapter 3 with three rewrite rules, one of which is the β reduction rule. When a β-reduction is to occur, a check is made to see if there would be a name clash. If there is, then an alpha conversion needs to be applied on the bound variables. This process of determining if two terms are α-equivalent is quadratic(Morazán and Schultz 2008). Mechanical reasoning of when to apply the alpha conversion is also quadratic.

A common method to ensure that the side condition is fulfilled is through applying alpha conversion on all bound variables on all subterms before β-reduction ensues, so $(\lambda x.(\lambda x.x)\,x)\,(\lambda x.x)$ becomes $(\lambda a.(\lambda b.b)\,a)\,(\lambda c.c)$ before reduction occurs. This method is due to Barendregt (1985) who introduced his variable convention to ensure clarity. While this simplifies the mechanised reasoning (i.e it is applied universally to all term so that the term undergoing reduction has only fresh variables), the fact remains that a quadratic process is still required for testing of α-equivalence. Furthermore in implementing this, a like-for-like analysis with Closure Calculus is no longer possible as Closure Calculus requires no additional phases.

Another common solution to the meta-theory problem follows the plan outlined in the previous paragraph. However, instead of alpha conversion, terms are converted into terms of the pure λ-calculus with de Bruijn indices. This conversion process makes reductions much easier. However, this complicates the implementation as the pure λ-calculus with de Bruijn indices brings its own meta-theory.

There exists other methods to reduce or remove the required meta-theory, such as using higher order abstraction syntax. However, those only serve to complicate the implementations. The abundance of choice in implementing the meta-operations, combined with the underspecificity of the meta-operations often lead to incorrect implementations.

All this and we still have not considered the data structures and algorithms required to implement the side condition. For example, the implementor may be obliged to keep track of which variables are bound when traversing a term, in order to find out which variable may be replaced by a fresh variable. This requires data structures that are more complicated than the λ-terms themselves! A complete description of the pure λ-calculus that takes into account the subtleties of the implementation details would undoubtedly be quite complex. The pursuit of such a complete description has been the mainstay of academia for years.

By contrast, the side conditions in the reduction rules of Closure Calculus are either trivial (variable comparison) or non-existent. The reduction rules may be implemented in a straightforwards fashion, leaving no choice of implementation for meta-operations. There is no need to keep track of free or bound variables. The result is an implementation that is far simpler than an implementation of the pure λ-calculus.

It is in this sense that the reduction rules of Closure Calculus is said to be simpler than that of the pure λ-calculus.

6.2 Complexity Metrics

The first result of this report is shown by comparing the complexity of an interpreter for Closure Calculus compared to an interpreter for the pure λ-Calculus. The most common measure of complexity is the notion of cyclomatic complexity (McCabe 1976).

Cyclomatic complexity is simply a count of linearly independent paths a program may take. Cyclomatic complexity is highly dependent on the control flow of a program. The more different paths a program may take, the more complex it is.

Another complexity metric that is quite commonly used is the COCOMO metrics (Boehm 1984). It is a model of estimating monetary costs, effort as measured by time requirements and resource as measured by human resources. Accordingly, the costs of software development is proportional to its complexity, thus by estimating the costs, one may estimate the complexity of a program. The latest model is derived from statistical regressions on 161 software development projects in the early 2000s. The basic COCOMO metrics are the only one suitable for use in the context of this work. It provides a rough order of magnitude estimates of software costs, which is enough.

Last, due to the use of an imperative programming language with few advanced features, we may rely on induction on the lines of code in the program. The longer a program is, the more complex it is.

The complexity metrics are calculated using an open source software called `scc` (Boyter 2018). `scc` automatically calculates these metrics and are used by corporations like Intel to manage complexity.

6.3 Closure Calculus Is Simpler To Implement

We may use the traditional measures of software complexity precisely because the programming language used is an imperative, non-functional programming language that does not obscure complexity. Thus the syntactic complexity of the program is proportional to the actual complexity of the program. Note that there is a combinatorial explosion of implementation choices when it comes to the pure λ-calculus. The implementation chosen for this report is one of the simplest possible implementation.

The following table presents the complexity metrics of the interpreters, following the complexity metrics explained in the previous chapter. The structure of the source code is laid out in separate files such that it is easier to include or exclude files when accounting for specific metrics of complexities. Explanations of the observations follow.

Interpreter	Lines of Code (Total)	Meta Theory LoC	Cyclomatic Complexity (Total)	Meta Theory CC	Cost	Time	People
CBN	226	116	30	17	$21,119	3.5 months	0.7
CBV	227	116	30	17	$21,147	3.5 months	0.7
Norm	231	116	30	17	$21,259	3.5 months	0.7
CCN	159	4	5	0	$18,325	2.9 months	0.5
CCDB	155	0	5	0	$18,325	2.9 months	0.5

It is quite obvious that an interpreter for Closure Calculus, regardless of flavour, is simpler than an interpreter for the pure λ-calculus. This is true across all the metrics.

According to the COCOMO metrics, it would take less time, less money and fewer people to build an interpreter for either flavour of Closure Calculus than any flavour of the pure λ-calculus. This is attributable to the cyclomatic complexity and the lines of code for each interpreter, which we discuss next.

The pure λ-calculus interpreters require around 230 lines of code to implement. Further analysis indicates that more than half those lines (116) are attributable to meta-theory requirements. The cyclomatic complexities may also be split into two separate metrics. Again, the cyclomatic complexities of the meta-theory component (17) dominates the cyclomatic complexities of the interpreters (13).

The Closure Calculus interpreters require fewer lines in total to implement. However, just comparing interpreters without the meta-theory, it can be seen that the pure λ-calculus interpreters take fewer lines (110-115 lines vs 155-159 lines) to implement. This ratio is similar to the ratios of lines of code found in the Haskell and OCaml implementations, which have meta-theoretic considerations obscured. However it would be dishonest to conclude that the pure λ-calculus is simpler. Without the meta-theory considerations the pure λ-calculus interpreters would simply not work correctly!

However the most surprising of the metrics is if we compare the cyclomatic complexity of only the interpreters for the pure λ-calculus and the cyclomatic complexity of the interpreters of Closure Calculus. Further analysis into this reveals that much of the complexity is attributable to the α-conversion rule. The α-conversion rule requires nested conditionals in order to correctly perform the meta-operation of substitution. This leads to a combinatorial increase in code paths, leading to higher cyclomatic complexity, which is a measure of how many paths a program may take.

There is debate whether to consider the α-conversion rule to be a meta-theory. If so, then the cyclomatic complexity of all the interpreters modulo meta-theory is 5. However, as most literature on the pure λ-calculus considers α-conversion to be part of the rewrite rules, this report treats it as a rewrite rule as well.

Chapter 7

Closure Calculus Terms Reduce to Normal Forms Faster

This chapter considers the comparison of the Closure Calculus interpreters with the pure λ-calculus interpreters. The comparison is done on the performance on reductions to normal form. First, an overview of the performance metrics is given. This is followed by a listing of benchmark programs and the results of a count of operations and meta operations. Then results on time and memory usage for the interpreters are given. Finally, this chapter closes with further analyses on the two sets of results, showing that the lack of meta-operations is the cause of the superior performance of the Closure Calculus interpreter.

7.1 Performance Metrics

Performance of the calculi is considered on three levels: the number of interpreter reduction operations, the time taken per operation and memory statistics. The tools used are built in to the Go programming language itself. The benchmark tools have been battle tested in large corporations. First I give a description of interpreter reduction operations, followed by a description of how time per operation is calculated, then a description of the memory statistics is given.

Interpreter reduction operation counts is straightforwards: it's simply a count of operations per reduction. However, a finer grained breakdown is given. The argument that is made is that the pure λ-calculus has fewer reductions in total, but has more meta-operations. These meta-operations take up computing resources too! Thus we must separately count the meta-operations alongside the number of reductions. Closure calculus has many more reduction rules than the pure λ-calculus, but little to no meta-operations required.

Given the operational differences between an interpreter of the pure λ-calculus and Closure Calculus, what might we say about the absolute time taken to compute a function? This is why absolute time measures need to also be taken into account.

When the program is compiled with the benchmarking turned on, each benchmark function (called an "operation") has an associated counter. Every time a function is called, the counter is incremented. An overall statistics of time taken per operation is the computed. The less time it takes to complete an operation, the faster it is.

Further, we may also instrument the operation with statistics on memory. Two statistics are provided for memory: amount of memory per operation, and the number of allocations required per operation. Performance is usually inversely proportional to these statistics.

Last, we may profile the program to figure out which exact component of an operation is taking the most computational resources. This would be useful in explaining the benefits and deficits of either calculi.

7.2 Closure Calculus Has Fewer Meta Operations

The previous chapter sets up the notion of performance metrics for comparison between the two calculi. One of the performance metrics is the operations count. To that end the interpreters have an optional flag that counts the reduction operations. While the previous chapter lists many tests to be performed, this section catalogues three of the simpler tests which are easily verifiable with pen and paper in order to facilitate discussions.

Given that Closure Calculus is aggressively normalising, and that these programs are relatively well behaved, the normal order interpreter for the pure λ-calculus is used. The following table presents the results. Explanations of the tests and observations follow.

Program and input	Interpreter	Reductions	Var Eq	Subs	Fresh	Capture
$(\lambda x.x\,y)u$	Norm	1	2	3	0	0
$(\lambda[I]x.x@y)u$	CCN	6	2	0	0	0
$(\lambda I(0@1))2$	CCDB	6	0	0	0	0
$(\lambda x.\lambda y.x)u$	Norm	1	4	2	1	1
$(\lambda[I]x.\lambda[I,x]y.x)u$	CCN	5	1	0	0	0
$(\lambda x.\lambda y.x)y$	Norm	1	7	2	2	2
$(\lambda[I]x.\lambda[I,x]y.x)y$	CCN	5	1	0	0	0
$(\lambda I\lambda(0,I)1)2$	CCDB	5	0	0	0	0
$\mathbf{SK}u\ v$	Norm	5	46	31	14	10
$\mathbf{SK}u\ v$	CCN	33	14	0	0	0
$\mathbf{SK}2,3$	CCDB	33	0	0	0	0

Table 7.1: Reduction results for various program in λ-Calculus

How the operations are counted is exposited in the following example, using the first program-input pair as the term to be reduced.

Example. Counting the reductions and meta-operations for $(\lambda x.x\,y)\,u$, $\lambda[I]x.x@y$ and $\lambda I(0@1)\,2$ respectively. Reduction counts will be in red, substitution counts will be in blue and variable equality will be in green. The counts are placed in running with the reduction steps. $\{u/x\}(x\,y)$ 1 0 0 reads: "after the reduction that yields the result $\{u/x\}(x\,y)$, 1 reduction, 0 substitutions and 0 variable equality checks would have been done". The following table shows the counts for each reduction step for each of the calculi.

Norm		CCN		CCDB	
		$(\lambda[I]x.x@y)u$		$(\lambda I(0@1))2$	
$(\lambda x.x\,y)u$		$\rightarrow (I :: x \mapsto u)(x@y)$	1 0 0	$\rightarrow (2,I)(0@1)$	1 0 0
$\rightarrow \{u/x\}(x\,y)$	1 0 0	$\rightarrow ((I :: x \mapsto u)\,x)((I :: x \mapsto u)\,y)$	2 0 0	$\rightarrow (2,I)0\,(2,I)1$	2 0 0
$\rightarrow (\{u/x\}x\ \{u/x\}y)$	1 1 0	$\rightarrow u((I :: x \mapsto u)\,y)$	3 0 0	$\rightarrow 2\,(2,I)\,0$	3 0 0
$\rightarrow (u\,y)$	1 3 2	$\rightarrow u(I\,y)$	4 0 1	$\rightarrow 2\,(I\,0)$	4 0 0
		$\rightarrow u\,y$	5 0 2	$\rightarrow 2\,0$	5 0 0
		$\rightarrow u@y$	6 0 2	$\rightarrow 2@0$	6 0 0

Fresh and Capture are additional operations that are also counted but not explained in the example above. Briefly, Fresh indicates the count of how many times a fresh variable is required in the road to reducing to a normal form. Capture indicates how many times a variable capture check was performed. Fresh and Capture are overlapping metrics - the counts may overlap with other counts of operations. Care must be taken when reasoning around aggregates of the metrics. However, substitutions and variable equality are non-overlapping metrics. These will be the main tools of reasoning.

The first set of examples is used to show a basic overview of how the counts of operations and meta-operations differ between the calculi.

The second set of examples, $(\lambda x.\lambda y.x)\,u$, $(\lambda x.\lambda y.x)y$ and its CCN equivalent results in the same equivalent NF but one application requires a more meta-operational checks to be performed while the other does not. Given that there can be no confusion in the unnamed version of Closure Calculus, only one the metrics of example, $(\lambda I \lambda(0, I)1)\,2$ is shown.

The last set of examples is more indicative of real-life programs that the two toy sets of examples above. $SKuv$ was chosen because it is small enough for the reader to verify manually while still showing a large difference in the performance of the pure λ-calculus and Closure Calculus.

Having fully introduced all the metrics and the classes of examples, let's proceed with the analysis of the reduction characteristics of the various calculi.

Overall the observation to be made is that the pure λ-calculus requires fewer dramatically reduction operations but requires more meta-operations. However, with the exception of CCDB, observe that if we deem reductions and substitutions to have the same costs, the sum of costs of operations and substitution is roughly of the same magnitude. For example, for the first set of tests, the sum of operations is 4, 6, 6 for Norm, CCN, and CCDB respectively. For the last set of tests, $SKuv$ the sum of operations is 36, 33, 33 for Norm, CCN and CCDB respectively.

7.3 Closure Calculus is Faster; Uses Less Memory

The previous section compares the number of operations and meta-operations each calculi. This section compares the performance of the calculi in terms of absolute time. This is done by benchmarking program-input pairs, listed in the previous chapter. Benchmarking toy examples aren't particularly useful, so the programs that are benchmarked here are more indicative of real world workloads. Only SKK is retained from the previous section as it gives insights to the impact of meta-operations to performance.

The following table shows the results of the benchmarks. Only CCN and CBN are compared. Descriptions of the metrics and results follow.

Benchmark	Ops (ns/op)		Memory (B/Op)		Allocs/Op	
	CCN	CBN	CCN	CBN	CCN	CBN
SKK	943	1792	22	128	0	9
Addition (Scott Numerals)	3218	34728	176	7067	0	265
	Ops (ms/op)		Memory (MB/Op)		Allocs/Op	
fib 20	185.76	2547.04	132.62	373.71	0	12762168
primes 30	172.85	4408.15	150.28	677.61	0	9176241
tak 28 12 7	2362.17	12711.02*	686.96	1042.26*	2	N/A

These benchmarks are done with a MacBook with an Intel Core i5 CPU @ 3.1GHz; 32GiB of RAM running on MacOS 10.13.6. The Go version used to compile the interpreter is go1.10 darwin/amd64. A note on the tak 28 12 7 benchmark: A rudimentary garbage collector had to be implemented for the pure λ-calculus interpreter, otherwise the benchmark would take too much memory and wouldn't complete. Thus the results have to be treated with care.

The metrics that are being benchmarked are Ops, Memory and Allocs.

Ops is the average time taken per operation. An operation is the totality of functions under benchmark. In the benchmarks, the interpreter's `Interpret` method is the function under benchmark. Time taken to prepare the program and inputs is not taken into account. A concrete example would be that the program and input $SKb\ a$, called SKK in the benchmarks table, took on average 943 nano seconds to complete when represented as Closure Calculus. The equivalent representation in the pure λ-calculus, interpreted by a CBN interpreter took on average 1792 nanoseconds to complete.

Memory is the amount of memory used by the program for the duration of the operation. Returning to the SKK benchmark, we see that when represented as Closure Calculus, it uses a mere 22 bytes per operation whereas it uses 128 bytes per operation for the CBN interpreter.

Last we consider Allocs. Allocs counts the number of times the Go runtime system has to allocate memory per operation. The runtime may need to allocate memory in order to facilitate dynamic data types such as slices, whose sizes are unknown until runtime. The CCN interpreter rarely needs to allocate memory. However, we can expect all pure λ-calculus interpreters to have some amount of allocations. This is because lists of free variables have to be determined at runtime in order to perform a β-reduction correctly.

The numbers speak for themselves. The CCN interpreter is a magnitude faster than the CBN interpreter on average. The CCN interpreter uses less memory than the CBN interpreter.

The hypothesis is that the meta-operations do not have the same costs as the operations. The next section confirms this.

7.4 Further Analysis

We may do further analysis by instrumenting the program with the profiling tools that come with the language. Reviewing the profiles, we find the CBN λ-calculus interpreter, spends most of its time performing substitution. In particular, capture avoiding substitution has been shown to be particularly expensive. For the naive Closure Calculus interpreter, most of the time is spent in reduction functions. Figures 7.4.1 and 7.4.2 shows the profiling report for the SKK benchmark. Despite being a simple program, the profile showing a record of what is happening at runtime is illuminating.

```
Showing top 10 nodes out of 91
  flat  flat%   sum%    cum   cum%
 290ms 14.87% 14.87% 1390ms 71.28%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).substitute
 150ms  7.69% 22.56%  150ms  7.69%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Free
 140ms  7.18% 29.74% 1180ms 60.51%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).substituteUnderLambda
 130ms  6.67% 36.41%  170ms  8.72%  github.com/chewxy/cccombos/internal/termso1.(*Manager).binop
 110ms  5.64% 42.05%  230ms 11.79%  github.com/chewxy/cccombos/internal/termso1.(*Manager).App
 110ms  5.64% 47.69%  220ms 11.28%  runtime.mallocgc
 100ms  5.13% 52.82%  260ms 13.33%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Cleanup
  80ms  4.10% 56.92%   90ms  4.62%  github.com/chewxy/cccombos/internal/termso1.(*Manager).NameOf
  60ms  3.08% 63.59%   60ms  3.08%  memeqbody
  50ms  2.56% 66.15%   50ms  2.56%  runtime.memequal
```

Figure 7.4.1: Profiler Reports for the CBN λ-calculus interpreter

```
Showing top 10 nodes out of 23
  flat  flat%   sum%    cum   cum%
 510ms 17.11% 17.11%  530ms 17.79%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Free
 420ms 14.09% 31.21% 1450ms 48.66%  github.com/chewxy/cccombos/cc/naive01.(*Interpreter).reduceAppSigmaX
 390ms 13.09% 44.30%  520ms 17.45%  github.com/chewxy/cccombos/internal/termso1.(*Manager).binop
 200ms  6.71% 51.01%  440ms 14.77%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Sigma
 190ms  6.38% 57.38%  200ms  6.71%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Get
 180ms  6.04% 63.42% 1750ms 58.72%  github.com/chewxy/cccombos/cc/naive01.(*Interpreter).reduceApp
 140ms  4.70% 68.12%  580ms 19.46%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Cleanup
 130ms  4.36% 72.48%  130ms  4.36%  github.com/chewxy/cccombos/internal/termso1.(*Manager).alloc
 120ms  4.03% 76.51%  160ms  5.37%  github.com/chewxy/cccombos/internal/termso1.(*Manager).NameOf
 110ms  3.69% 80.20%  200ms  6.71%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Closure
```

Figure 7.4.2: Profiler reports for the CCN interpreter

The reports show the top 10 functions that spend most time executing. It is sorted by the *total* time spent executing a function. This is misleadingly called **`flat`**. The other metric, **`cum`**, denotes the total cumulative time spent in the function. This includes the time spent when control has left the function. For example, allocating a slice may trigger a garbage collection cycle. That time is included in the cumulative time, but not the flat time. Both views are important in understanding the performance characteristics of a program.

In both profile reports, the functions are highlighted according to the use case of the function. Functions highlighted yellow are functions directly involved in reduction of terms. Functions highlighted red are functions related to the meta theory operations. Functions highlighted blue are supporting functions needed by the runtime of each individual calculus. Last, functions highlighted grey are functions from the Go runtime.

First observation that can be made is that there are no functions from the Go runtime in the report for the CCN interpreter. That the functions from the Go runtime show up in the profile for the CBN interpreter is telling. Closure calculus has no requirement for us to track free variables. Hence there is no need to acquire and pass around sets of free variables. On the other hand, the pure λ-calculus requires us to keep track of sets of free variables. These sets are implemented as slices in Go, which, due to its dynamic nature, requires assistance from the garbage collector from time to time. `runtime.mallocgc` is precisely the function that is called when dealing with dynamically sized sets of free variables.

Upon further inspection, we see that this train of thinking is correct. If the profiles are sorted by cumulative time spent, then we see `fv` and `fresh` being in the top 10:

```
Showing top 10 nodes out of 91
  flat  flat%   sum%    cum   cum%
 0.02s  1.03%  2.56%  1.47s 75.38%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).Interpret
 0.03s  1.54%  4.10%  1.45s 74.36%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).eval
 0.01s  0.51%  4.62%  1.41s 72.31%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).beta
 0.29s 14.87% 19.49%  1.39s 71.28%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).substitute
 0.14s  7.18% 26.67%  1.18s 60.51%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).substituteUnderLambda
 0.10s  5.13% 31.79%  0.26s 13.33%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Cleanup
 0.04s  2.05% 33.85%  0.26s 13.33%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).fv
 0.11s  5.64% 39.49%  0.23s 11.79%  github.com/chewxy/cccombos/internal/termso1.(*Manager).App
 0.11s  5.64% 45.13%  0.22s 11.28%  runtime.mallocgc
     0     0% 45.13%  0.18s  9.23%  github.com/chewxy/cccombos/lc/eager01.(*Interpreter).fresh
```

Figure 7.4.3: Profiler reports for the CBN interpreter, sorted by cumulative time spent

By contrast, the cumulative profile for CCN is as follows:

```
Showing top 10 nodes out of 23
  flat  flat%   sum%    cum   cum%
  80ms  2.66%  4.32% 1940ms 64.45%  github.com/chewxy/cccombos/cc/naive01.(*Interpreter).Interpret
 190ms  6.31% 10.63% 1780ms 59.14%  github.com/chewxy/cccombos/cc/naive01.(*Interpreter).reduceApp
 440ms 14.62% 25.25% 1520ms 50.50%  github.com/chewxy/cccombos/cc/naive01.(*Interpreter).reduceAppSigmaX
 140ms  6.31% 31.56%  750ms 24.92%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Cleanup
 530ms 17.61% 49.17%  580ms 19.27%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Free
 160ms  5.32% 54.49%  410ms 13.62%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Sigma
 310ms 10.30% 64.78%  410ms 13.62%  github.com/chewxy/cccombos/internal/termso1.(*Manager).binop
  90ms  2.99% 67.77%  260ms  8.64%  github.com/chewxy/cccombos/cc/naive01.(*Interpreter).A
 220ms  7.31% 75.08%  230ms  7.64%  github.com/chewxy/cccombos/internal/termso1.(*Manager).NameOf
 160ms  5.32% 80.40%  210ms  6.98%  github.com/chewxy/cccombos/internal/termso1.(*Manager).Get
```

Figure 7.4.4: Profiler reports for the CCN interpreter, sorted by cumulative time spent

These patterns in both reports are as expected. The majority of the cumulative time is spent on reducing terms. However, observe that the total time spent in the `substitute` function in the CBN profile (highlighted in green). Other than the runtime support functions, the `substitute` function is where the program spends the most time on. This, coupled with the fact that `memeqbody` and `runtime.memequal` shows up in the top 10 functions where the program spends the most time in confirms that it is passing around a set of free variables that is the cause of the slow performance when compared to the CCN interpreter.

When the same kind of profiling is done on more complicated programs, similar patterns show up. This allows us to come to a conclusion that this is congruent with the intuition that the simplicity of Closure Calculus leads to a more performant programming language.

This has some implications on implementing a programming language. The next part of this report addresses that.

Part III

Implications

Chapter 8

Interesting Artefacts of Closure Calculus

The nature of Closure Calculus upends some familiar understandings in pure λ-calculus. This chapter discusses with some such findings. They may be useful for programming language implementors.

First, a discussion of Closure Calculus and natural numbers is given. Encoding of natural numbers in the pure λ-calculus are not analogous[1] in Closure Calculus.

Then an introduction to List and Pair is given. This is a precursor to the next section where the Church-encoded Pred, previously deemed unimplementable is defined.

The sections that follow then gives more concrete advice to programming language implementors. These are artefacts that would sit well in the toolbox of programming language implementors.

Evaluation functions with local variables are introduced. An example of simulating Combinatory Logic in Closure Calculus is given.

This is followed by an example of functions that extract the body of an abstraction. Last, a quick guide is given on how using the Identity operator in place of an identity function would yield fewer reduction steps.

8.1 Natural Numbers

In general, Closure Calculus lends itself really well to encodings that embed values.

What I mean by "embed values" can be readily seen in Figure 8.1.1. Observe that in the Scott and Parigot encodings of natural numbers, the underlined terms in the bodies are numbers are "embedded" whole in the term. The term

[1]defined in Section 5.1 of Chapter 5

$$\begin{aligned}\mathbf{0} &:= \lambda f.\lambda x.x\\ \mathbf{Succ} &:= \lambda n.\lambda f.\lambda x.f\,(n\,f\,x)\\ \mathbf{1} &:= \mathbf{Succ}\ \mathbf{0}\\ &\to^* \lambda f.\lambda x.f\,x\\ \mathbf{2} &:= \mathbf{Succ}\ \mathbf{1}\\ &\to^* \lambda f.\lambda x.f\,(f\,x)\\ &\ldots\end{aligned}$$

(a) Church Encoding

$$\begin{aligned}\mathbf{0} &:= \lambda f.\lambda x.f\\ \mathbf{Succ} &:= \lambda n.\lambda f.\lambda x.x\,n\\ \mathbf{1} &:= \mathbf{Succ}\ \mathbf{0}\\ &\to^* \lambda f.\lambda x.x\,\underline{\mathbf{0}}\\ \mathbf{2} &:= \mathbf{Succ}\ \mathbf{1}\\ &\to^* \lambda f.\lambda x.x\,\underline{\mathbf{1}}\\ &\ldots\end{aligned}$$

(b) Scott Encoding

$$\begin{aligned}\mathbf{0} &:= \lambda f.\lambda x.x\\ \mathbf{Succ} &:= \lambda n.\lambda f.\lambda x.f\,n\,(n\,f\,a)\\ \mathbf{1} &:= \mathbf{Succ}\ \mathbf{0}\\ &\to^* \lambda f.\lambda x.f\,\underline{\mathbf{0}}\,x\\ \mathbf{2} &:= \mathbf{Succ}\ \mathbf{1}\\ &\to^* \lambda f.\lambda x.f\,\underline{\mathbf{1}}\,(f\,\underline{\mathbf{0}}\,x)\\ &\ldots\end{aligned}$$

(c) Parigot encoding

Figure 8.1.1: Number Encodings in the pure λ-calculus

representing 2 contains the term representing 1. The term representing 1 contains the term representing 0. This cannot be seen in the Church encoding. Instead, the Church encoding may be viewed as an iterator that generates new numbers.

Like the pure λ-calculus, Closure Calculus has no notion of natural numbers. Instead, natural numbers may be encoded in the terms of Closure Calculus.

Using the Scott encoding, the Peano axioms may be stated as such:

$$\begin{aligned}\mathbf{0} &:= \lambda[I]f.\lambda[I :: f \mapsto f]x.f\\ \mathbf{Succ} &:= \lambda[I]n.\lambda[I :: n \mapsto n]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n\end{aligned}$$

The natural numbers may then be inductively defined:

$$\begin{aligned}
&\mathbf{1} := \mathbf{Succ}\ \mathbf{0}\\
&\quad = (\lambda[I]n.\lambda[I :: n \mapsto n]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n)\ \mathbf{0}\\
&\quad \to \lambda[I :: n \mapsto \mathbf{0}]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n\\
&\mathbf{2} := \mathbf{Succ}\ \mathbf{1}\\
&\quad = (\lambda[I]n.\lambda[I :: n \mapsto n]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n)\ \mathbf{1}\\
&\quad \to \lambda[I :: n \mapsto \mathbf{1}]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n\\
&\quad ...
\end{aligned}$$

These results are indeed strange, especially when one is familiar with the λ-calculus version of the Scott numerals.

Running the resulting terms through the conversion process as described in the previous section yields different terms from what is expected:

$$\begin{aligned}
[\![\mathbf{0}_\lambda]\!] &= [\![\lambda f.\lambda x.f]\!]\\
&\to^* \lambda[I]f.\lambda[I :: f \mapsto f]x.f\\
&\equiv \mathbf{0}_{cc}\\
[\![\mathbf{1}_\lambda]\!] &= [\![\lambda f.\lambda x.x\,\mathbf{0}_\lambda]\!]\\
&\to^* \lambda[I]f.\lambda[I :: f \mapsto f]x.x@\mathbf{0}_{cc}\\
&\not\equiv 1_{cc}\\
[\![\mathbf{2}_\lambda]\!] &= [\![\lambda f.\lambda x.x\,\mathbf{1}_\lambda]\!]\\
&\to^* \lambda[I]f.\lambda[I :: f \mapsto f]x.x@[\![\mathbf{1}_\lambda]\!]\\
&\not\equiv \mathbf{2}_{cc}\\
&...
\end{aligned}$$

Scott encoded numbers in the pure λ-calculus are not analogous in Closure Calculus. What to make of this? Here, we should follow the definition of a natural number as defined by Peano. The number 1 is defined as the succession of 0. The result of applying the successor function to 0 is $\lambda[I :: n \mapsto 0]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n$. Therefore we call this term the canonical representation of a Scott encoded number 1.

It is important to observe that the canonical representation is but a representation. Numbers are represented differently in different number systems - the most common of which is the positional base-10 number system that we're familiar with. In λ-calculus itself, the common encoding of natural numbers are the Church numerals. Alternate encodings, like the Scott encoding, or the Parigot encoding exists. They represent the same abstract notion of numbers in different terms.

Closure Calculus clarifies the difference between encoding and representation. Different, non-equivalent terms may be used to represent the same number under the same encoding scheme. Using a different term $\lambda[I]f.\lambda[I :: f \mapsto$

$f]x.x@0$ or indeed $\lambda[I]f.\lambda[I]x.x@0$ as a Scott-encoded number 1 does not break any functions that expects a Scott-encoded number.

More importantly, note that the three terms are all in normal form:

$$\begin{aligned}
&\lambda[I :: n \mapsto 0]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n \\
&\lambda[I]f.\lambda[I :: f \mapsto f]x.x@0 \\
&\lambda[I]f.\lambda[I]x.x@0
\end{aligned}$$

Are there benefits in separating the concept of encoding and representation? We may think of this as a stepping stone to a constructing a notion of semantic equality of terms. Equally, we may consider then that Closure Calculus is capable of maintaining multiple representations of the same term. This allows programming language implementors to choose the best representation for the given task.

8.2 List and Pair

Encodings of lists are fixtures of introductory lambda calculus courses. A list of numbers, $[1, 2, 3]$ is also commonly written $Cons\ 1\ (Cons\ 2\ (Cons\ 3\ Nil)))$, where $Cons$ and Nil are the constructors of a list data type. The pure λ-calclus approach require that $Cons$ and Nil be meta-variables, standing in for some λ-term. The usual accounts of list encodings can be faithfully translated to Closure Calculus. A brief overview is presented in Figure 8.2.1.

$$\begin{aligned}
\mathbf{selFst} &:= \lambda[I]x.\lambda[I :: x \mapsto x]y.x \\
\mathbf{selSnd} &:= \lambda[I]x.\lambda[I :: x \mapsto x]y.y \\
\mathbf{True} &:= \mathbf{selFst} \\
\mathbf{False} &:= \mathbf{selSnd} \\
\mathbf{falseGen} &:= \lambda[I]f.f@\mathbf{False}
\end{aligned}$$

(a) Common Terms

$$\begin{aligned}
\mathbf{Pair} &:= \lambda[I]x.\lambda[I, x]y.\lambda[I, x, y]f.f@x@y \\
\mathbf{fst} &:= \lambda[I]f.f@\mathbf{selFst} \\
\mathbf{snd} &:= \lambda[I]f.f@\mathbf{selSnd} \\
\mathbf{Nil} &:= \lambda[I]x.\mathbf{Pair}@\mathbf{tTrue}@\mathbf{True}@x \\
\mathbf{Cons} &:= \lambda[I]x.\lambda[I, x]xs.\mathbf{Pair}@\mathbf{False}@(\mathbf{Pair}@x@xs) \\
\mathbf{head} &:= \lambda[I]x.\mathbf{fst}@(\mathbf{snd}@x) \\
\mathbf{tail} &:= \lambda[I]x.\mathbf{snd}@(\mathbf{snd}@x) \\
\mathbf{isNil} &:= \mathbf{fst}
\end{aligned}$$

(b) Church-encoded list

$$\begin{aligned}
\mathbf{Nil} &:= \lambda[I]f.\lambda[I, f]x.f \\
\mathbf{Cons} &:= \lambda[I]x.\lambda[I, x]xs.\lambda[I, x, xs]f.\lambda[I, x, xs, f]g.g@x@xs \\
\mathbf{head} &:= \lambda[I]xs.xs@\mathbf{Nil}@(\lambda[I]x.\lambda[I, x]xs.x) \\
\mathbf{tail} &:= \lambda[I]xs.xs@\mathbf{Nil}@(\lambda[I]x.\lambda[I, x]xs.xs) \\
\mathbf{isNil} &:= \lambda[I]xs.xs@\mathbf{True}@\mathbf{falseGen}
\end{aligned}$$

(c) Scott Encoded list

Figure 8.2.1: List and Pairs Encodings

The list and pair encodings are rather straightforwards. It must be rather strange to see it included in a chapter on the interesting artefacts of Closure Calculus. Nonetheless, they are introduced here as the notion of lists and pairs will help with the later sections. In particular, a Church-encoded *pred* can be derived using the Church pair. Further, explorations on encoding schemes yield some interesting results, which are good to juxtapose against more traditional encodings.

8.3 On The Church-Encoded *pred*

In the original paper outlining Closure Calculus, there was a claim made that the predecessor function for Church encoded numbers may not be implemented in Closure Calculus for the want of reduction under the λ. However, there is another way to define *pred* without first having to undergo anaesthesia at the dentist.

We start by following the traditional thinking around reversible computing: we may represent the result as a pair of values - the left projection of the pair encodes the history or environment of function that was applied while the right projection gives the resulting value. *pred* is the dual to *succ*. Hence to encode the history of applying *succ*, we merely need to store the last number. We write the pair as $(prev, curr)$.

Define a function **bind1** $:= \lambda[I]p.\mathbf{Pair}@(\mathbf{snd}@p)@(\mathbf{succ}@(\mathbf{snd}@\mathbf{p}))$. This function takes a pair p and observes the right projection, which we have defined as *curr*. Then it creates a new pair using the **Pair** function, only now *curr* is placed in the left projection and the successor of *curr* is placed in the right projection. It should be noted that this is a matter of definition. If we had defined the pair to be $(curr, prev)$ then the orders would be reversed, and **fst** would be used in place of **snd** in **bind1**[2].

Next, define **unit0** $:= (\mathbf{Pair}\,\mathbf{0}\,\mathbf{0})$. **unit0** is a pair of two zeroes, the unit value of the function we're interested in. Observe that **unit0** is a standard application, not a tagged application. It is the resulting NF that we're interested in. In full it's written $\lambda[I :: x \mapsto \mathbf{0} :: y \mapsto \mathbf{0}]f.f@x@y$.

Thus *pred* can then be defined as $\lambda[I]n.\mathbf{fst}@(n@\mathbf{bind1}@\mathbf{unit0})$. This definition of *pred* takes advantage of the fact that Church numerals are iterators - an encoding of a natural number n simply means apply a function f n times. Examples follow the definition of common terms.

$$\begin{aligned}
\mathbf{0} &:= \lambda[I]f.\lambda[I,f]x.x \\
\mathbf{succ} &:= \lambda[I]n.\lambda[I,n]f.\lambda[I,n,f]x.f@(n@f@x) \\
\mathbf{1} &:= \mathbf{succ}\ \mathbf{0} \\
&\rightarrow^* \lambda[I :: n \mapsto \mathbf{0}]f.\lambda[I,n,f]x.f@(n@f@x) \\
\mathbf{2} &:= \mathbf{succ}\ \mathbf{1} \\
&\rightarrow^* \lambda[I :: n \mapsto \mathbf{1}]f.\lambda[I,n,f]x.f@(n@f@x) \\
&\ldots
\end{aligned}$$

[2]The name curious name **bind1** is derived from the usual signature of a monadic bind ala Haskell, where the second argument is a function with a signature `(a -> m b)`

Example. **pred 1** ≡ **pred** (**succ 0**) gives the following reduction sequence:

$$\begin{aligned}
\textbf{pred}\ (\textbf{succ 0}) &\to^* \textbf{fst}\ ((\lambda[I :: n \mapsto \textbf{0}]f.\lambda[I, n, f]x.f@(n@f@x))\ \textbf{bind1 unit0}) \\
&\to^* \textbf{fst}\ ((\lambda[I :: n \mapsto \textbf{0} :: f \mapsto \textbf{bind1}]x.f@(n@f@x))\ \textbf{unit0})) \\
&\to^* \textbf{fst}\ ((I :: n \mapsto \textbf{0} :: f \mapsto \textbf{bind1} :: x \mapsto \textbf{unit0})(f@(n@f@x)) \\
&\to^* \textbf{fst}\ (\textbf{bind1}\ (\textbf{0 bind1 unit0})) \\
&\equiv\ \ \textbf{fst}\ (\textbf{bind1}\ ((\lambda[I]f.\lambda[I, f]x.x)\ \textbf{bind1 unit0})) \\
&\to^* \textbf{fst}\ (\textbf{bind1}\ ((\lambda[I :: f \mapsto \textbf{bind1}]x.x)\ \textbf{unit0})) \\
&\to^* \textbf{fst}\ (\textbf{bind1 unit0}) \\
&\to^* \lambda[I]f.\lambda[I, f]x.x \\
&\equiv \textbf{0}
\end{aligned}$$

Example. **pred 2** ≡ **pred** (**succ 1**) gives the following reduction sequence:

$$\begin{aligned}
\textbf{pred}\ (\textbf{succ 1}) &\to^* \textbf{fst}\ ((\lambda[I :: n \mapsto \textbf{1}]f.\lambda[I, n, f]x.f@(n@f@x))\ \textbf{bind1 unit0}) \\
&\to^* \textbf{fst}\ ((\lambda[I :: n \mapsto \textbf{1} :: f \mapsto \textbf{bind1}]x.f@(n@f@x))\ \textbf{unit0})) \\
&\to^* \textbf{fst}\ ((I :: n \mapsto \textbf{1} :: f \mapsto \textbf{bind1} :: x \mapsto \textbf{unit0})(f@(n@f@x)) \\
&\to^* \textbf{fst}\ (\textbf{bind1}\ (\textbf{1 bind1 unit0})) \\
&\equiv\ \ \textbf{fst}\ (\textbf{bind1}\ ((\lambda[I :: n \mapsto \textbf{0}]f.\lambda[I, n, f]x.f@(n@f@x))\ \textbf{bind1 unit0})) \\
&\to^* \textbf{fst}\ (\textbf{bind1}\ ((\lambda[I :: n \mapsto \textbf{0} :: f \mapsto \textbf{bind1}]x.f@(n@f@x))\ \textbf{unit0})) \\
&\to^* \textbf{fst}\ (\textbf{bind1}\ ((I :: n \mapsto \textbf{0} :: f \mapsto \textbf{bind1} :: x \mapsto \textbf{unit0})(f@(n@f@x)))) \\
&\to^* \textbf{fst}\ (\textbf{bind1}\ (\textbf{bind1}\ (\textbf{0 bind1 unit0}))) \\
&\to^* \textbf{fst}\ (\textbf{bind1}\ (\textbf{bind1 unit0})) \\
&\equiv \textbf{fst}\ (\lambda[I :: x \mapsto \textbf{1} :: y \mapsto \lambda[I :: n \mapsto \textbf{1}]f.\lambda[I, n, f]x.f@(n@f@x)]f.f@x@y) \\
&\to^* \lambda[I :: n \mapsto \textbf{0}]f.\lambda[I, n, f]x.f@(n@f@x) \\
&\equiv \textbf{1}
\end{aligned}$$

8.4 Evaluation Functions With Local Variables

In Closure Calculus it is easy to clearly define evaluation functions with local variables. A usual evaluation function has this form:

$$\lambda \overbrace{[\rho]}^{Env} x. \overbrace{(\sigma :: y \mapsto t)}^{Locals} @ \overbrace{x}^{B}$$

Here, we may think of the evaluation function as having three components - the derived *Env*, local variables *Locals* and the body *B*. The use of *Locals* makes defining an evaluation function easy. The following subsection shows an example of applying this concept to give an alternative simulation of Combinatory Logic in Closure Calculus.

Encoding a TRS in Closure Calculus

The combinatory logic (CL) is a first order TRS. The signature Σ of CL are given as such:

$$\Sigma = \{S, K, I, App\}$$

Observe that unlike λ-calculus, there is no notion of term-level variables. The rules are given as such:

$$\begin{aligned} App(App(App(S\ x)\ y)\ z) &\rightarrow App(App(x\ z)App(y\ z)) \\ App(App(K\ x)\ y) &\rightarrow x \\ App(I\ x) &\rightarrow x \end{aligned}$$

Some common combinators are the B, C, M combinators, defined as such:

$$\begin{aligned} B &:= S(KS)K && \text{(bluebird)} \\ C &:= S(BBS)(KK) && \text{(cardinal)} \\ M &:= SII && \text{(mockingbird)} \end{aligned}$$

One way to simulate CL in Closure Calculus is similar to the usual way that it is simulated in the pure λ-calculus:

$$\begin{aligned} \mathbf{S} &:= \lambda[I]x.\lambda[I,x]y.\lambda[I,x,y]z.(x@z)@(y@z) \\ \mathbf{K} &:= \lambda[I]x.\lambda[I,x]y.x \\ \mathbf{I} &:= \lambda[I]x.x \end{aligned}$$

Given these, the Y combinator is simply defined as $\lambda[I]x.\mathbf{S}@\mathbf{I}@\mathbf{I}@x$. The B combinator may be defined as $\lambda[I]x.\mathbf{S}@(\mathbf{K}@\mathbf{S})@\mathbf{K}@x$.

The alternate way to simulate CL is given by embedding the definitions of S, K, I in an extension. Call this function *eval*[3].

$$\mathbf{eval} := \lambda[I]x.(I :: s \mapsto \mathbf{S} :: k \mapsto \mathbf{K})@x$$

It is able to evaluate any arbitrary expression in combinatory logic, written in a form where *App* is replaced with @. The so-called bird combinators (Smullyan 2012) can be defined as such:

$$\mathbf{B} := s@(k@S)@k \qquad \text{(bluebird)}$$
$$\mathbf{C} := s@(\mathbf{B}@\mathbf{B})@S)@(K@K) \qquad \text{(cardinal)}$$
$$\mathbf{M} := s@I@I \qquad \text{(mockingbird)}$$

The M Combinator is perhaps the more interesting example. Observe that I is not actually in **boldface**. Nor is there a mapping from a variable to an identity mapping. Instead, the I is the identity operator from CCN itself. This is how it reduces:

$$\begin{aligned}
(\mathbf{eval}\ \mathbf{M}\ z) &= (\lambda[I]x.(I :: s \mapsto \mathbf{S} :: k \mapsto \mathbf{K})@x)\ (s@I@I)\ z \\
&\to ((I :: x \mapsto (s@I@I)\ (I :: s \mapsto \mathbf{S} :: k \mapsto \mathbf{K})@x)\ z \\
&\to^* ((I :: s \mapsto \mathbf{S} :: k \mapsto \mathbf{K} :: x \mapsto (s@I@I))\ (s@I@I))\ z \\
&\to^* ((\lambda[I :: x \mapsto I]y.\lambda[I, x, y]z.(x@z)@(y@z))\ I)\ z \\
&\to (\lambda[I :: x \mapsto I :: y \mapsto I]z.(x@z)@(y@z))\ z \\
&\to^* z@z
\end{aligned}$$

The M combinator opens up the power of full recursion. The Y combinator is defined as $S(K(M))(S(S(KS)K)(K(M))$. Encoding it and reduction is left as an exercise to the reader.

Via CL, there may exist a route to defining Closure Calculus in the terms of Closure Calculus.

[3]An alternative to this *eval* is $\mathbf{eval} := \lambda[I :: s \mapsto \mathbf{S} :: k \mapsto \mathbf{K}]x.I@x$. The results are the same.

8.5 Extracting the Body of an Abstraction:

While not intensional in the traditional sense, Closure Calculus allows for extraction of the body of an abstraction. Consider the following program and application in CCN:

$$
\begin{aligned}
\mathbf{F} &:= \lambda[I]x.a;\\
\mathbf{T} &:= \lambda[\mathbf{F}]y.z@y;\\
(\mathbf{T}\,b) &= (\lambda[\mathbf{F}]y.z@y)\,b\\
&\rightarrow (\mathbf{F} :: y \mapsto b)(z@y)\\
&\rightarrow (\mathbf{F} :: y \mapsto b)\,z)\,((\mathbf{F} :: y \mapsto b)\,y)\\
&\rightarrow (\mathbf{F}\,z)\,((\mathbf{F} :: y \mapsto b)\,y)\\
&\rightarrow ((I :: x \mapsto z)\,a)\,((\mathbf{F} :: y \mapsto b)\,y)\\
&\rightarrow (I\,a)\,((\mathbf{F} :: y \mapsto b)\,y)\\
&\rightarrow a\,((\mathbf{F} :: y \mapsto b)\,y)\\
&\rightarrow a\,b\\
&\rightarrow a@b
\end{aligned}
$$

This could of course go really wrong:

$$
\begin{aligned}
\mathbf{F} &:= \lambda[I]x.a;\\
T &:= \lambda[\mathbf{F}]y.z@c;\\
(T\,b) &= (\lambda[\mathbf{F}]y.z@c)\,b\\
&\rightarrow (\mathbf{F} :: y \mapsto b)(z@c)\\
&\rightarrow (\mathbf{F} :: y \mapsto b)\,z)\,((\mathbf{F} :: y \mapsto b)\,c)\\
&\rightarrow (\mathbf{F}\,z)\,((\mathbf{F} :: y \mapsto b)\,c)\\
&\rightarrow ((I :: x \mapsto z)\,a)\,((\mathbf{F} :: y \mapsto b)\,c)\\
&\rightarrow (I\,a)\,((\mathbf{F} :: y \mapsto b)\,c)\\
&\rightarrow a\,((\mathbf{F} :: y \mapsto b)\,c)\\
&\rightarrow^{*} a\,a\\
&\rightarrow a@a
\end{aligned}
$$

The structure of the extension provides enough guarantees that if a variable name matches, then its substitution is produced. This allows for a well formed macro system to be constructed.

8.6 Concerning Identity

The identity function is heavily used in functional programming. In λ-calculus, the identity abstraction is written $\lambda x.x$. Converting the identity abstraction to

Closure Calculus yields $\lambda[I]x.x$ and λIJ.

However, it should be noted both flavours of Closure Calculus come equiped with the identity operator. They may be used instead of defining an identity abstraction. Consider the *Pred* function for Scott encoded numbers:

$$
\begin{aligned}
\mathbf{id} &:= \lambda[I]x.x \\
\mathbf{Pred} &:= \lambda[I]n.n@\mathbf{0}@\mathbf{id}; \\
\mathbf{Pred}\,\mathbf{1} &= (\lambda[I]n.n@\mathbf{0}@\mathbf{id})\mathbf{1} \\
&\to (I :: n \mapsto \mathbf{1})(n@\mathbf{0}@\mathbf{id}) \\
&\to ((I :: n \mapsto \mathbf{1})\,(n@\mathbf{0}))\ ((I :: n \mapsto \mathbf{1})\,\mathbf{id}) \\
&\to (((I :: n \mapsto \mathbf{1})\,n)\,(I :: n \mapsto \mathbf{1})\,0)\ ((I :: n \mapsto \mathbf{1})\,\mathbf{id}) \\
&\to (\mathbf{1}\,(I :: n \mapsto \mathbf{1})\,\mathbf{0})\ ((I :: n \mapsto \mathbf{1})\,\mathbf{id}) \\
&\to^{*} (\mathbf{1}\,\mathbf{0})\ ((I :: n \mapsto \mathbf{1})\,\mathbf{id}) \\
&\equiv ((\lambda[I :: n \mapsto \mathbf{0}]f.\lambda[I :: n \mapsto n :: f \mapsto f]x.x@n)\,\mathbf{0})\ ((I :: n \mapsto \mathbf{1})\,\mathbf{id}) \\
&\to^{*} (\lambda[I :: n \mapsto \mathbf{0} :: f \mapsto \mathbf{0}]x.x@n)\ ((I :: n \mapsto \mathbf{1})\,\mathbf{id}) \\
&\to^{*} (\lambda[I :: n \mapsto 0 :: f \mapsto \mathbf{0}]x.x@n)\ \mathbf{id} \\
&\to^{*} (I :: n \mapsto \mathbf{0} :: f \mapsto \mathbf{0} :: x \mapsto \mathbf{id})\ (x@n) \\
&\to^{*} \mathbf{0} \\
\mathbf{Pred_2} &:= \lambda[I]n.n@\mathbf{0}@I; \\
\mathbf{Pred_2}\,\mathbf{1} &= (\lambda[I]n.n@\mathbf{0}@I)1 \\
&\to^{*} (\lambda[I :: n \mapsto \mathbf{0} :: f \mapsto \mathbf{0}]x.x@n)\ ((I :: n \mapsto \mathbf{1})\,I) \\
&\to (\lambda[I :: n \mapsto \mathbf{0} :: f \mapsto \mathbf{0}]x.x@n)\ I \\
&\to (I :: n \mapsto \mathbf{0} :: f \mapsto \mathbf{0} :: x \mapsto I)\ (x@n) \\
&\to^{*} \mathbf{0}
\end{aligned}
$$

Reductions using the I term results in a shorter reduction sequence. This is a good form of optimisation for compilers as well.

Chapter 9

Discussions

The results presented in the previous chapters - implementing an interpreter for both calculi - yields some important implications for developing compilers for functional programming languages. This chapter discusses these implications.

Two results follow from the previous chapters. The first is that reductions in Closure Calculus is faster than reductions in the pure λ-calculus.

Second, is that the interpreter for Closure Calculus is simpler than that of the pure λ-calculus. This is across all metrics of complexity - an interpreter of Closure Calculus takes fewer lines of code to implement, and has a lower cyclomatic complexity than an interpreter for the pure λ-calculus.

This has some implication on programming language implementors, the main stakeholders in this report. So the first section of this chapter is dedicated to the implications.

There may of course be some potential objections, which are laid out in the second section.

In order to answer these objections, a survey of compilation methodologies for functional programming language is presented. Finally, the objections are answered

A response to these objections would require a cursory look at how industrial functional programming languages are compiled.

9.1 What Does This Mean For Programming Language Theory

While this report concerns the comparison of the pure λ-calculus to Closure Calculus, the learnings transfer to programming language theory as well. Much of the report concerns the benchmarking of term reductions in both lambda calculi as if they were programming languages. This was not accidental. Rather, the goal is to persuade stakeholders that the qualities of Closure Calculus makes it ideal as a theoretical basis for functional languages.

9.2 Three Potential Objections

We begin by addressing potential objections that may arise from the previous chapters.

The first objection that may arise is that typical interpreters of the pure λ-calculus are not implemented the way it was implemented for this report.

The second objection might be that industrial programming languages do not use the pure λ-calculus.

A third and final objection might be that the implementation are of untyped calculi, which has little utility in real-life programming languages.

9.3 A Survey of Compilation Methodologies for Popular Programming languages

In order to properly respond to the potential objections, we must take a detour to look at the compilation methodologies of two popular functional programming languages. First we will look at the compilation methodologies of Haskell, then the same is done for Ocaml.An outline suffices for the considerations.

Haskell first parses the source code into an AST called HsSyn. The source code, represented as HsSyn goes through a series of checks before being desugared into GHC Core. GHC Core is a very simple typed λ-calculus based on System F_ω. After being translated into GHC Core, some work is done to simplify the terms and prepare it for conversion into the language of the Spineless Tagless G-machine (STG).

STG is the abstract machine that Haskell runs on (Peyton Jones and Salkild 1989; Peyton Jones 1992). In the STG, Core terms are translated into STG bindings. STG bindings has the following form:

$$f = \underbrace{\underbrace{\{v_1, v_2, \dots v_n\}}_{\texttt{free vars}} \lambda \overset{\texttt{update flag}}{\overline{\pi}} x.e}_{\texttt{lambda-form}}$$

From an implementation point of view, f is a pointer to a heap allocated closure. The closure is a `lambda-form`, containing a list of free variables, an update flag π and a body, e, which is any valid STG expression. Each of the free variables in the list of free variables is a pointer to a memory location.

Now let us consider another form of terms readers would by now be familiar with - the shorthand notation for a CCN term:

$$\lambda[I, \overbrace{v_1, v_2, \dots v_n}^{\texttt{"free" vars}}].x.t$$

Observe that the lambda form and the abstraction form of Closure Calculus are very similar! We will return to discussing this form in the later sections.

For now, let us turn our attention to the compilation phases of OCaml.

Ocaml compiles to machine code first by parsing the source code and preprocessing it using `Camlp4`. After typechecking and inferencing is done, the source code is desugared into λ-form. Like GHC Core, the λ-form of OCaml is a small λ-calculus. However, unlike GHC Core, it is untyped. Simplification of terms is performed and then a closure conversion is performed.

A closure is defined as follows (unrelated fields have been removed for readability):

```
and ufunction = {
  label  : function_label;
  arity  : int;
  params : (Backend_var.With_provenance.t * value_kind) list;
  return : value_kind;
  body   : ulambda;
  env    : Backend_var.t option;
}
```

Observe that this is very similar to the alternative representation of a term as shown in the Alternative Representations of Terms section in the Appendix, which bears repeating here with fields renamed to match the definition of `ufunction`:

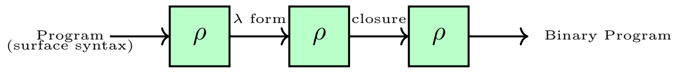
```
type Closure struct {
        env   Term       // env
        param Variable   // bound variable
        body  Term       // body

        label string // meta name for nicer pretty printing
}
```

The `ufunction` data type is simply an uncurried version of the `Closure` data type - an optimisation that is fairly trivial to implement.

The key insight however, is that Ocaml, like Haskell, relies on conversion to closures in its compilation phase.

In fact, a cursory analysis of all compiled functional programming languages show that at some point in their compiler phases, closure conversion is required. These closure conversion phases usually sit close to the end of the compilation pipeline, merely a few steps away from compilation to binary. However, before the closure conversion phases, the surface syntax of the languages are first translated into some form of non-closure λ-calculus (GHC Core for Haskell, and λ-form in Ocaml). Thus the compilation process for functional programming languages may be generalised to a process that looks as follows (ρ denotes an arbitrary translation process):



9.4 Answering the Objections

We are now ready to answer the objections that may arise.

Objection 1 - An Intentionally Naïve Pure λ-Calculus Interpreter was Used for Comparison

The first objection is that the comparison may be unfair, for typical interpreters of the pure λ-calculus are not implemented the way it was implemented for this report. Recall that the interpreters for this report are implemented as tree-walking interpreters. Substitutions are replaced directly in the term itself. Usual interpreters of the pure λ-calculus use a variety of augmentations, such as de Bruijn indices and environment-based abstract machines. Three answers are given for this objection.

The first answer is that using augmented or multi-pass interpreters is akin to comparisons against a different lambda calculus. Secondly, an argument could be made that these augmentation only serve to make an interpreter more complex - the point of the exercise is to compare the simplest possible interpreters. Lastly, using augmented interpreters makes "comparing the performance characteristics of the calculi" a meaningless statement.

Consider the following reductio ad absurdum argument: imagine that there exists an ideal interpreter for each calculus. Each of these interpreters involve many translation steps that translate the syntax of the calculus to machine language. All optimisations that can be performed are performed with the aim of generating the fastest binary program. Given that the interpreters are ideal translators, any program written in either calculus will converge to the same binary program. Therefore any benchmark is useless. Instead, the benchmarks should be done on the most naïve form - if both interpreters are equally naïve in implementation then we may conclude that the differences are not due to optimisations.

Objection 2 - Industrial Programming Languages Do Not Use The Pure λ-Calculus

The second objection is that industrial programming languages surely do not use the pure λ-calculus, so a comparison with the pure λ-calculus is moot. However, this is easily answered by the fact that all functional programming languages are rooted somewhat in the pure λ-calculus, in that few changes are necessary in order to translate it to a slightly different calculus like System F. Further, the pure λ-calculus is universally understood by functional programming language implementers. A comparison of Closure Calculus to the pure λ-calculus is both instructive and illuminating of what is missing.

Objection 3 - Untyped Calculi Have Little Utility in Real Life

The answer to the third objection - that untyped calculi are of little utility in real life - is answered by OCaml, which uses an untyped λ-calculus under the hood. If it is of utility to a popular industrial functional programming language, surely the third objection is based on a false premise.

The answers to the objection points to where Closure Calculus will excel: being an intermediate representation (IR) for a higher level programming language.

9.5 On The Oddity of $x \mapsto x$

Now we return to a discussion of Closure Calculus terms. It is remarked upon in the original paper outlining Closure Calculus and in Section 4.2 of Chapter 4 that the requirement to add a $x \mapsto x$ to the environment of an abstraction term was a little odd.

To a programming language implementor however, the reasoning is obvious. In practice all variables are associated with a context. A scope or context is usually implemented as mappings of labels (variables) to memory locations that contains a value.

Two observations can be made. First, the memory must be allocated for the value that the variable represents[1]. The second observation is more subtle: that variables play dual roles: first as a label for lookup within a context, second as a memory location. The subtlety is highlighted when phrased thus: There is an abstract component to variables and a concrete component to variables. Thus the context/environment/scope is a bridge from the abstract (label) to the concrete (memory location).

Recall from Chapter 4 that the extension form serves a dual purpose - as the environment to a closure and as a substitution. The particular genius of this purposeful conflation of notions allows for a principled transition from abstract to concrete. A comparison is warranted.

Let us consider once again, the following pure λ-calculus term:

$$\lambda x.\lambda y.x$$

An abstract Haskell-like or OCaml-like implementation would perform a closure conversion, yielding a data structure that looks like this:

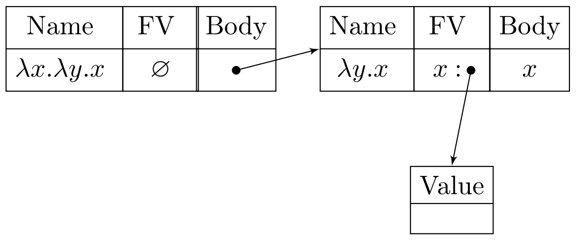


Figure 9.5.1: An abstract data structure representation of $\lambda x.\lambda y.x$

[1]A well-known programming language implementor once confided in me that "reality hates sum types"

Observe that the memory location that the free variable x is pointing to is unfilled. However, a fact must not go unnoticed: the memory is allocated to hold a value when it arises. We will return to this fact shortly.

Now, let's say at run time, the term $\lambda x.\lambda y.x$ was applied to a value, 100. The runtime system creates fills in the value in the pre-allocated memory. The resulting concrete data structure looks like this:

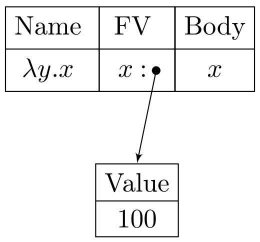


Figure 9.5.2: Result of $((\lambda x.\lambda y.x)\ 100)$ represented in the same format as above

The process of filling the pre-allocated memory with the value bound to x in the FV field is a meta-operation - an operation outside the defined confines of the calculus. It took very many years of analysis before such meta-theory could be formalised (Appel 2007).

The conflation of the two roles that variables play - one as a label and one as a pointer to memory - shows up immediately in Closure Calculus. $x \mapsto x$ is a syntactic representation of a mapping from the abstract (a label) to the concrete (a memory location). To make things work correctly, the value is itself a label. An illustration is warranted, so let's return to the example above, but this time, with Closure Calculus terms.

The term under consideration is $\lambda[I]x.\lambda[I, x]y.x$. The following data structure represents it well:

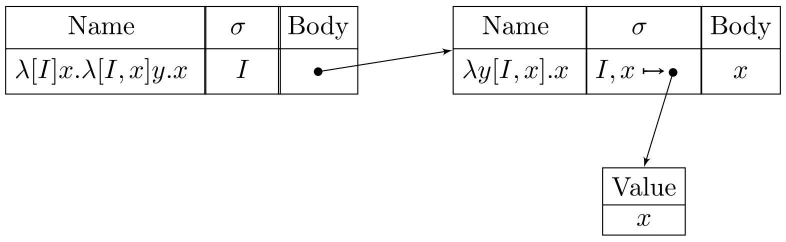


Figure 9.5.3: Abstract data structure representation of $\lambda[I]x.\lambda[I, x]y.x$

Upon application to a value, 100, the resulting data structure is similar to the one above.

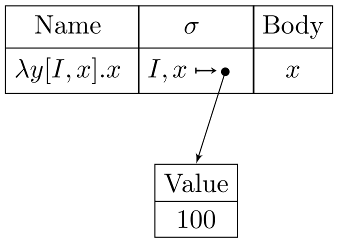


Figure 9.5.4: Result of $((\lambda[I]x.\lambda[I,x]y.x)\,100)$

The rule that allowed for a principled reduction of terms is

$$(\sigma :: x \mapsto t)(\rho :: z \mapsto u) \rightharpoonup (\sigma :: x \mapsto t)\rho :: z \mapsto (\sigma :: x \mapsto t)u$$

It does not privilege either the blue or red subterms on the left hand side. However, in a typical reduction, the blue subterm corresponds to a substitution and the red subterm corresponds to a closure environment. Mapping a variable to itself is therefore the most natural thing to do. In short, Closure Calculus presents a principled way of mapping abstract terms to concrete values in memory locations. This clarity would undoubtedly lead to better methods of understanding memory use.

Chapter 10

Conclusion

Closure Calculus is a better lambda calculus than the pure λ-calculus. It is simpler than the pure λ-calculus, leading to simpler implementations. Closure Calculus is also more performant than the pure λ-calculus.

Comparisons of Closure Calculus against the pure λ-calculus was done by implementing an interpreter for each of the two flavours of Closure Calculus (CCN and CCDB) and two interpreters for the pure λ-calculus, representing the pure λ-calculus with a Call-By-Value evaluation strategy and the Call-By-Name evaluation strategy respectively. To compare simplicity of implementation, the following metrics were used: Lines of Code, Cyclomatic Complexity and the COCOMO metric. To compare performance of programs in either calculi, execution time and memory were compared.

The comparisons show that Closure Calculus is simpler to implement when compared to the pure λ-calculus. This is attributable to the fact that the meta-theory requirements for Closure Calculus are either trivial (variable equality in CCN) or non-existent (CCDB).

When comparing absolute time, Closure Calculus terms reduce to normal form faster than an equivalent pure λ-calculus term. Reduction to normal forms also use less memory than reduction of an equivalent term in the pure λ-calculus. This contributes to the reduction speed of the pure λ-calculus.

Being simpler and faster than the pure λ-calculus, it can be said that Closure Calculus is better than the pure λ-calculus.

This has implications on programming language design. Programming language implementors should seriously consider using Closure Calculus as a theoretical basis for functional programming languages as doing so allows for functional programming languages that are simpler to implement, and yet have faster execution of programs.

Bibliography

Abadi, M., Cardelli, L., Curien, P.-L. and Lèvy, J.-J. 1990, 'Explicit Substitutions', *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages, San Francisco, California*, ACM, pp. 31–46.

Abelson, H., Sussman, G. J. and with Julie Sussman 1996, *Structure and Interpretation of Computer Programs*, 2nd edn., MIT Press/McGraw-Hill, Cambridge.

Appel, A. W. 2007, *Compiling with Continuations*, Cambridge University Press.

Archambault-Bouffard, V. and Monnier, S. ????, 'Implementation of Explicit Substitutions: From $\lambda\sigma$ to the Suspension Calculus', .

Baader, F. and Nipkow, T. 1999, *Term rewriting and all that*, Cambridge university press.

Barendregt, H. 1985, *The Lambda Calculus: Its Syntax and Semantics*, Studies in logic and the foundations of mathematics, North-Holland.

Bezem, M., Klop, J. W. and de Vrijer, R. 2003, 'Term rewriting systems by terese. number 55 in cambridge tracts in theoretical computer science', .

Bezem, M., Klop, J. W. and de Vrijer, R. 2006, 'Terese lite: Excerpts from the book term rewriting systems by terese', .

Boehm, B. W. 1984, 'Software engineering economics', *IEEE Trans. Softw. Eng.*, vol. 10, no. 1, pp. 4–21.
URL: *http://dx.doi.org/10.1109/TSE.1984.5010193*

Boyter, B. 2018, 'scc', .
URL: *https://github.com/boyter/scc*

Church, A. 1932, 'A Set of Postulates for the Foundation of Logic', *Annals of Mathematics*, vol. 33, no. 2, pp. 346–366.

Church, A. 1936, 'An Unsolvable Problem of Elementary Number Theory', *American Journal of Mathematics*, vol. 58, no. 2, pp. 345–363.

DeRemer, F. and Kron, H. 1975, 'Programming-in-the large versus programming-in-the-small', *SIGPLAN Not.*, vol. 10, no. 6, pp. 114–121.
URL: *http://doi.acm.org/10.1145/390016.808431*

Felleisen, M., Findler, R. B. and Flatt, M. 2009, *Semantics Engineering with PLT Redex*, 1st edn., The MIT Press.

Go Authors, T. 2018, 'The go programming language specification', .
URL: *https://golang.org/ref/spec*

Haskell Language Authors, T. 2014, 'Haskell language', .
URL: *https://www.haskell.org/*

Jay, B. 2017, 'Intensional-computation, Repository of Proofs in Coq', .
URL: *https://github.com/Barry-Jay/Intensional-computation*

Jay, B. 2018, 'Closure Calculus', .

Jay, B. and Vergara, J. 2014, 'Confusion in the church-turing thesis', .

Kleene, S. C. and Rosser, J. B. 1935, 'The Inconsistency of Certain Formal Logics', *Annals of Mathematics*, vol. 36, no. 3, pp. 630–636.

Klop, J. W. 1990, *Term rewriting systems*, Centrum voor Wiskunde en Informatica.

Knuth, D. E. 1991, 'Textbook examples of recursion', *Artificial Intelligence and Mathematical Theory of Computation. Papers in Honor of John McCarthy*, pp. 207–229.

Knuth, D. E. 1997, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

McCabe, T. J. 1976, 'A complexity measure', *IEEE Transactions on software Engineering*, , no. 4, pp. 308–320.

Mellies, P.-A. 1995, 'Typed λ-calculi with Explicit Substitutions May Not Terminate', Dezani-Ciancaglini, M. and Plotkin, G. (eds.) *Typed Lambda Calculi and Applications*, Springer Berlin Heidelberg, pp. 328–334.

Michaelson, G. J. 1989, 'An introduction to functional programming through lambda calculus', *International computer science series*, .

Morazán, M. T. and Schultz, U. P. 2008, 'Optimal Lambda Lifting in Quadratic Time', Chitil, O., Horváth, Z. and Zsók, V. (eds.) *Implementation and Application of Functional Languages*, Springer Berlin Heidelberg, pp. 37–56.

Munoz, C. A. 1996, 'Confluence and Preservation of Strong Normalisation in An Explicit Substitutions Calculus', *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, IEEE Computer Society, pp. 440–7.

Perlis, A. J. 1982, ‘Special feature: Epigrams on programming’, *SIGPLAN Not.*, vol. 17, no. 9, pp. 7–13.
URL: *http://doi.acm.org/10.1145/947955.1083808*

Peyton Jones, S. L. 1992, ‘Implementing Lazy Functional Languages on Stock Hardware: the Spineless Tagless G-machine’, *Journal of functional programming*, vol. 2, no. 2, pp. 127–202.

Peyton Jones, S. L. and Salkild, J. 1989, ‘The spineless tagless g-machine’, *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA ’89, ACM, New York, NY, USA, pp. 184–201.
URL: *http://doi.acm.org/10.1145/99370.99385*

Pike, R. 2012, ‘Go at google: Language design in the service of software engineering’, .

Reynolds, J. C. 1972, ‘Definitional Interpreters for Higher-order Programming Languages’, *Proceedings of the ACM Annual Conference - Volume 2*, ACM ’72, ACM, pp. 717–740.

Smullyan, R. 2012, *To Mock a Mocking Bird*, Knopf Doubleday Publishing Group.
URL: *https://books.google.com.au/books?id=NyF1kvJhZbAC*

Steele, G. L., Jr. 1978, ‘Rabbit: a Compiler for Scheme’, Tech. rep.

Thompson, S. 1991, *Type Theory and Functional Programming*, Addison Wesley.

Tromp, J. 2018, ‘Functional bits : Lambda calculus based algorithmic information theory’, .

Turing, A. M. 1937, ‘On computable numbers, with an application to the entscheidungsproblem’, *Proceedings of the London mathematical society*, vol. 2, no. 1, pp. 230–265.

Wright, A. K., Felleisen, M. et al. 1994, ‘A Syntactic Approach to Type Soundness’, *Information and computation*, vol. 115, no. 1, pp. 38–94.

Appendix

A Primer on Go

This section introduces the Go programming language in a succinct manner. Readers familiar to the language or readers disinterested in the details may opt to skip it. The section proceeds as follows: first, sytactic idiosycracies are introduced. This is followed by the introduction of features that allow for constructing algorithms. Last, features that allow for constructing data structures are introduced.

Go is an open source, strongly-typed, garbage collected programming language from Google for the purposes of systems programming. The language was specifically designed for software development in the large (as defined by DeRemer and Kron (1975))(Pike 2012). This to say, Go is well suited to large teams of people working on one program. Furthermore, unlike most major programming languages, Go has a finalized specification (Go Authors 2018). The stability is ideal for comparisons of performance.

Syntactically, `//` denotes the start of a comment in Go source code. In this report it may be used to also add commentaries to programs written in Go.

Go programs are organized in packages. Packages contain program entities such as variables or types.

An identifier is a name given to a program entity. Identifiers take two forms: exported or unexported. Exported identifiers start with a capital letter while unexported identifier start with a small letter. Exported identifiers are accessible from outside the package it was defined in while unexported identifiers are local to the package it was defined in. A list of predeclared identifiers is exempt from this rule.

Go supports the usual notion of computational variables and constants. Functions are defined by the `func` keyword. The `func` keyword is also used to define methods, which are functions associated with an object.

Example. I is a function that takes an int and returns an int.

```
func I(a int) int {
        return a
}
```

Conditionals are supported by means of an `if` statement. When there are many conditions, a switch table is used. These are useful for implementing operational semantics.

Example. The following snippet implements the Call-By-Value operational semantics of the pure λ-Calculus with a switch table. It is not unlike pattern-matching guards used in functional programming languages:

```
switch at := a.(type) {
case Variable:
        return a
case Lam:
        return a
case App:
        at.B = interpreter.evalCBV(at.B)
        at.A = interpreter.evalCBV(at.A)
        return at
}
```

There is a minimalistic type system for the language. Type definitions serve dual roles in defining a type, and defining its memory layout. This is useful for having fine grain control over structuring of data.

The primary way of structuring data in Go is with a `struct`. The `struct` has been briefly introduced in the previous sections without explanations. A struct defines the layout of memory in the computer. It is comprised of a sequence of fields, each of which has a name and a type. A struct may be empty, in which no memory is allocated to it. A struct may also have an embedded field - a field with no explicit name. The embedded field is specified as a type name. An example is more illumating, so one follows.

Example. `foo` is a type definition with the layout defined by a struct.

```
type foo struct {
        bar
        A int
        b float64
}
```

Several observations can be made. `foo` is an unexported type. This means that `foo` may not be accessed from outside the package it was defined in. In practice this means external functions may not create values of type `foo`. However, values of type `foo` may still be created within the local package and used by external functions.

Values of type `foo` are comprised of some value of type `bar`, a value of type `int`, called `A` and a value of type `float64`, named `b`. Each value of type `foo` is arranged in the order it was defined in computer memory. If `bar` takes 4 bytes, while both `int` and `float64` each takes 8 bytes, then we may visualise the layout as such[1]:

bar	A	b

Now we turn our attention to the fields themselves. `bar` is an embedded field in `foo`. It is unexported, so it may not be accessed from outside the package it was defined in. The same is true for `b`. Meanwhile, `A` is exported. It is accessible from outside the package. For example, an external function may be written to change the value of `A` directly. Exported fields of unexported struct types are generally considered bad form but allows for some programs to be written which would otherwise been unable to be written.

Another important data structure that comes built in to Go is a slice. But first, the array type must be defined. An array is a contiguous block of memory with N elements of a type, where the size of each element is determined by the size of the type. An array is written `[N]T` where `N` is a meta-variable standing for a natural number and `T` is a meta-variable standing for a type. The size of an array type is static and is part of the type definition. A value with type `[2]int` does not have the same type as a value with type `[5]int`.

Having introduced the array type, we can focus on the slice. A slice type is like an array type, except its size is dynamic. It is written `[]T`, where `T` is a meta-variable standing for a type. Being dynamically sized means we do not have to worry about allocating or deallocating memory, as it is handled by the language itself.

Go also provides built in support for hashmaps. A hashmap is a data structure for implementing associative arrays. A hashmap type is written `map[T]U`, where `T` and `U` are meta variables for types of the key and value respectively.

We may define new types in Go that has the layout of already-defined types. This is particularly useful for using types as a guard against bad values.

Example. In the following snippet, `foo` is a new type but has the layout of an `int` type. `bar` is another new type that has the layout of the `foo` type.

```
type foo int
type bar foo
```

A value of type `foo` does not have the same type as a value of type `bar`. They may not be used interchangeably.

Last but not least, Go has the notion of pointers. A pointer points to a location in memory. Most types are addressable, that is, we may take the address of the value of that type. A pointer is written `*T`, where `T` is a meta-variable standing for a type. We may take the address of a variable by writing `&x`, where `x` is an addressable identifier. A variable holding a pointer type may be dereferenced by writing `*x`.

[1]This visualization features proportional sizing of fields. For the rest of the report, visualisations of structs will not take into account the size of fields

There are many more features of the Go programming language. However, this is all the basics necessary to understand the implementation of the calculi.

Alternative Representations of Terms

This section discusses the potential alternative representation of terms in the interpreters implemented. First a brief description of Go's interface types is given, then the alternative representation is given. Last a rationale is given on why the alternative representation is not suitable for the purposes of this report.

Interface Types, A Brief (de)Tour

A brief detour is necessary as a key concept required is that of an interface type in Go. Interface types are the only way of supporting type polymorphism in Go.

Briefly, an interface type is a type defined by a set of methods. So long as a type T implements all the methods of an interface type `I`, then we say `T` implements `I`.

Let there be an interface type `Stringer` and three other types `T` , `U` and `V`, which are defined in the following snippet:

```
type Stringer interface {
        String() string
}
type T string
type U int
type V bool

func (u U) String() string { return "" }
func (v V) String() string { return "" }
func (v V) Greet() string  { return "hello" }
```

This program snippet says that `T` is internally represented as a string, while `U` is internally represented as an integer and `V` is internally represented as a boolean. The types `U` and `V` each has a method called `String()`, which returns a value of type `string` (an empty string). The type `T` does not have any methods defined on it. Additionally, note that `V` has a method `Greet`, which also returns a string (‘`hello`’).

`U` and `V` each implements `Stringer`. `T` does not.

In the following snippet, only a value of type `U` or a value of type `V` may inhibit the variable `foo`.

```
var foo Stringer
foo = U(1)        // OK
foo = T("hello")  // compilation error
foo = V(true)     // OK
```

Notice that `foo` may take values from more than one type. This idea is very similar to typeclasses in Haskell, whose primary motivation was to provide similar kinds of ad-hoc type polymorphism.

Now, let's say `foo` is inhabited by a value of type `V`. Trying to access the `Greet` method of type `V` will cause a compilation error. This is because the type `Stringer` only has one method it knows about. This kind of type polymorphism on its own is often valuable enough. However, there may be times we would need to know which concrete type inhabits the variable `foo`. We would need to query the structure of the type.

We query the structure of the value of `foo` by means of a `switch` statement as follows:

```
switch bar := foo.(type){
        case U:
                // do something
        case V:
                // do something
        default:
}
```

The `default` keyword is a special syntax to allow us to define the behaviour of a program when the concrete type is an unhandled case.

Recall that the interface types are fulfilled by any type that has a method. This allows for highly extensible software libraries. However, one would need to deal with unhandled cases. One does this by defining the behaviour under the default case.

Although there exists a calculus that allows programmers to safely add cases to existing functions (i.e. the Pattern Calculus), language features like these may not be as beneficial as thought for software engineering in the large[2].

Alternative Representation of Terms

Having described the interface type and ways of querying it, we may now proceed with exploring the alternative representation of terms.

We start by first describing a type for `Term`, as follows:

```
type Term interface {
        isTerm()
}
```

As interface types support ad hoc polymorphism, any term may implement the set of methods defining the type. The use of an unexported function limits where a `Term` may be defined to only within a local package. This solves the issue of having potentially unhandled cases.

Then we describe the various form a term may take. Using CCN as an example, we can describe the six forms a term may take as follows:

[2]This was explained to me after I had put a change request to extend Go with Pattern Calculus-esque case extensions.

```
type Variable string

type App struct{ A, B Term }

type Tag App

type I struct{}

type Ext struct {
        Ext      Term // Parent env
        Variable      // Explicit Sub
        Term          // Explicit Sub
}

type Closure struct {
        Ext      Term // env
        Variable      // bound variable
        Term          // body

        MetaName string // meta name for nicer pretty printing
}
func (t Variable) isTerm()            {}
func (t App) isTerm()                 {}
func (t Tag) isTerm()                 {}
func (t Ext) isTerm()                 {}
func (t Closure) isTerm()             {}
func (t I) isTerm()                   {}
```

This corresponds well to the BNF of CCN. By defining a method called `isTerm` to each of the types defined here, we may now have represented the terms of the pure λ-calculus as the interface type `Term`. An interpreter is simply a function that queries the internal structure of a `Term` to decide which function to transition to next.

As befitting a programming language designed around software development in the large, this representation of terms is very easy to understand. In fact, this would be the canonical way to implement a λ-calculus in Go.

So Why Not Use This Representation?

Two main objections arise. Both objections to using this representation revolve around the runtime system of Go. We wish to compare interpreters for the pure λ-calculus and Closure Calculus. Our comparison is done by means of benchmarking and profiling of the interpreter.

Thus the first objection is that we want our runtimes for the interpreters to be as pure as possible - in that the runtime system of Go should interfere as little as possible. The use of interface types leverages the runtime system of Go quite a bit. In the runtime profiles, they show up as `runtime.convT2I` or `runtime.convT2E`. The additional noise in the data decreases confidence in the analyses.

The second objection arises from the consequence of relying on the runtime system of Go. Go is a garbage collected programming language. Use of interface types causes heap allocations which may trigger a garbage collection cycle. This would lead to non-determinism of program execution, affecting the statistics collected. Clearly this won't do.

With these two objections, it is now clear why the representation as presented in Chapter **??** was chosen. It was chosen to minimise work done by the Go runtime system, such that maximal amounts of work is done by the interpreters instead. This is done so that a fair and objective comparison between the implementations of the calculi may be made.

On the Confusion of the Church-Turing Thesis

This section discusses the confusion of the Church-Turing thesis in brief. A significantly more detailed exposition is to be found in (Jay and Vergara 2014).

Briefly, the Church thesis states the following[3]:

$$\forall f : \mathbb{N}_\Lambda \rightarrow \mathbb{N}_\Lambda,\ f \text{ is effectively calculatable}$$

Where $\mathbb{N}_\Lambda$ is a natural number encoded as a λ-term. This statement says that effectively calculable, and hence general recursive.

Turing's thesis may be stated in the brief as the following[4]:

$$\forall f : \mathbb{S} \rightarrow \mathbb{S},\ f \text{ is computable}$$

This statement states that a function from a string $\mathbb{S}$ to $\mathbb{S}$ is considered computable if there exists a Turing Machine may compute it.

Turing had focused the majority of his paper on computable numbers. This was necessary in order to use a Gödelian approach to proving that the Entscheidungsproblem cannot be solved. He first demonstrated that one may construct a Turing Machine to compute any function of natural numbers, albeit as encoded in what we would call bit-strings in modern times.

Kleene later equated Turing's notion of computability with both Church's notion of effectively calculatability and Gödel's general recursivity in his Theorem XXX.

However, consider a Universal Turing Machine may simulate any other Turing machine. So what of the following functions?

$$g : \mathbb{P} \rightarrow \mathbb{P},\ \text{where } \mathbb{P} : \mathbb{S} \rightarrow \mathbb{S}$$

Herein lies the problem with having a confusion. If we are to accept that Church's thesis is equivalent to Turing's thesis, then we must accept that g are computable by λ-calculus. It isn't. To wit, given two closed terms of the pure λ-calculus in NF, it is not possible to define an equality relation. λ-calculus simply does not provide enough intensionality to be able to realize as many functions of g as the Turing model of computation can. This is a particularly important factor limiting the space of designing programming languages.

Does it really matter? In some sense, no. Recalling that all functions may be encoded as a Gödel number, consider the cardinality of all natural numbers, $\aleph_0$. It is much smaller than the cardinality of is power set $2^{\aleph_0}$, of which $\mathbb{P}$ lies in. A pessimistic viewpoint would be to merely consider Church's thesis to be equivalent to Turing's thesis, and shutting off any more explorations in to a $2^{\aleph_0}$

[3]a more formal notion of Church's thesis is the following statement: $\forall f : \mathbb{N} \rightarrow \mathbb{N}, \exists e \forall i \exists k$ such that $T(e, i, k) \wedge U(k, i)$, where T is Kleene's T predicate and U is a function that will provide an answer if T determines that the function halts. i is the input and k is a computable function encoded as a Gödel number and e is the index of the halting state. Ironically this is not expressible in λ-calculus, but is expressible in the Turing model of computation.

[4]Turing was more descriptive but less rigorous than Church was. The statement is a very rough formalisation of Turing's thesis

space. A more optimistic viewpoint would be to acknowledge the very minor deficiencies of λ-calculus, and set forth to discover better calculi. Better calculi informs better, safer programming language paradigms. This is a good thing to hold on to when faced with doubt in holding on to the optimistic view.