Catarina da Costa Boucinha

Operational Semantics for Linear Languages

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
June, 2012
Catarina da Costa Boucinha

Operational Semantics for Linear Languages

Thesis submitted to the Faculdade de Ciências of Universidade do Porto to obtain Masters degree in Computer Science

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
June, 2012
To my parents and all my friends...
Acknowledgments

First of all I would like to thank my supervisor Sandra Alves for all her help throughout the writing of this thesis, especially bearing with my constant nervousness and always helping to calm me down. This thesis would have never been possible without her help and support. I would also like to thank Mário Florido for his help, always with helpful suggestions and advice.

I would like to thank all my friends, especially Eva Maia, for always getting me to go further than what I though I could go, for all the advice, knowledge and friendship shared throughout the years, and of course for all the coffee breaks. And to Marcela Oliveira, for all the support and her caring shoulder, for being my family when I lived in Porto and for putting up with my stressed person every single day. I thank all my friends from Ovar, who made my weekends a thing to look forward to and who always listen to my computer “geek” talk. Diana, Luís, Marilia, Nuno and Sofia my wild side thanks you with all its heart. I would also thank my friends who made every train, subway or bus ride an adventure, an epic odyssey of stories that I will remember forever.

And since I would have a very long list of names if I were to mention you all, I would like to thank all my friends that shared with me the adventures that I like to call “the story that I will tell my grandchildren”, the sad and happy moments. I would like to thank the ones who shared their knowledge, how to be a better person, how the world works; the ones that sang with me, that grew with me, that traveled the world with me; and the ones who stood on a stage with me and that showed me that I had more courage than what I though.

Last but never least, I would like to thank my family, specially my mother and father, without whom I would have never made it this far. For their constant support through the good and bad decisions that I made, for always working so hard to give me the opportunity to reach this far in the academic world, for their constant advice (even when I did not want to ear it) that helped me be the person I am today, and finally for listening to my never ending constant talking with all the patience in the world.
Abstract

The goal of this thesis is to study the operational semantics of the linear language, System $\mathcal{L}_{rec}$, in order to explore its use as an intermediate language. System $\mathcal{L}_{rec}$ is a linear $\lambda$-calculus extended with numbers, pairs and an unbound linear recursor, with a close-reduction strategy. It is compatible with the notion of linear function with a minimal extension to the linear $\lambda$-calculus, keeping the system Turing-complete.

The contributions of this thesis are:

- Implementations of interpreters call-by-name for the PCF language, System $\mathcal{L}_{rec}$, and its extension with built-in naturals, $\mathcal{L}_{rec}^N$ and an interpreter call-by-value for System $\mathcal{L}_{rec}$;

- Implementations of call-by-name abstract machines for the PCF language, the System $\mathcal{L}_{rec}$ and $\mathcal{L}_{rec}^\text{nat}$;

- A compilation from PCF to System $\mathcal{L}_{rec}$ and $\mathcal{L}_{rec}^\text{nat}$;

- Several benchmarks on the number of reductions for its different machines.
Resumo

Esta tese tem como objectivo estudar as semânticas operacionais de uma linguagem linear, Sistema $L_{rec}$, de maneira a explorar o seu uso como uma linguagem intermédia. O Sistema $L_{rec}$ é um $\lambda$-calculus extendido com números, pares e um recursor linear não ligado, com uma estratégia de redução fechada. É compatível com a noção de função linear com uma extensão mínima ao $\lambda$-calculus linear, mantendo o sistema Turing completo.

As contribuições desta tese são:

- Implementações de interpretadores de chamada-por-nome para a linguagem PCF, Sistema $L_{rec}$ e uma extensão deste com naturais built-in, $L_{nat}^{rec}$, e um interpretador de chamada-por-valor para o Sistema $L_{rec}$;

- Implementação de máquinas abstractas de chamada-por-nome para a linguagem PCF, para o Sistema $L_{rec}$ e $L_{nat}^{rec}$;

- Uma compilação de PCF para o Sistema $L_{rec}$ e $L_{nat}^{rec}$;

- Várias conclusões sobre o número de reduções para as diferentes máquinas.
Contents

Abstract ............................................. 9

Resumo .............................................. 11

List of Tables ....................................... 17

List of Figures ...................................... 19

1 Introduction ...................................... 21
   1.1 Motivation and Related Work .................. 22
   1.2 Overview of the Thesis ........................ 23

2 Background ....................................... 25
   2.1 Lambda-Calculus ............................... 25
      2.1.1 Syntax .................................. 25
      2.1.2 Variables and Substitution ............... 26
      2.1.3 The $\beta$-reduction .................... 28
      2.1.4 Reduction Strategies ..................... 28
      2.1.5 Linear Calculi ........................... 30
   2.2 The Simple Typed Lambda Calculus .......... 31
   2.3 Operational Semantics and Abstract Machine 33
2.3.1 Call-by-Name and Call-by-Value .................................. 33
2.3.2 Krivine Machine ...................................................... 33

3 PCF .......................................................... 37
   3.1 Syntax .......................................................... 37
   3.2 Implementation .................................................. 39
      3.2.1 Call-by-Name Interpreter ................................. 40
      3.2.2 Stack Machine .............................................. 42

4 System $\mathcal{L}_{rec}$ ........................................ 45
   4.1 Syntax .......................................................... 45
      4.1.1 Variables and Substitution ............................... 46
      4.1.2 Types for System $\mathcal{L}_{rec}$ ......................... 49
      4.1.3 Evaluation Strategies ................................... 51
   4.2 Implementation .................................................. 52
      4.2.1 Call-by-Name Interpreter ................................. 53
      4.2.2 Call-by-Value Interpreter ................................. 55
      4.2.3 Stack Machine .............................................. 55

5 Compiling ......................................................... 61
   5.1 Compiling ........................................................ 61
   5.2 Comparing Results .............................................. 63
      5.2.1 System $\mathcal{L}_{rec}$ with built-in naturals .......... 66
      5.2.2 Comparing Results with $\mathcal{L}_{rec}^{nat}$ .......... 68

6 Conclusion ......................................................... 73

A Code .......................................................... 75
A.1 PCF Language ......................................................... 75
A.2 $\mathcal{L}_{rec}$ Language ............................................. 76
A.3 Compilation ........................................................ 78
A.3.1 $\mathcal{L}_{rec}^{\text{nat}}$ Code ..................................... 82

References .............................................................. 87
## List of Tables

2.1 Normal Forms, where $M_i \in \Lambda$ (Definition 2.1.1) ........................................ 29
2.2 Krivine Stack Machine ......................................................................................... 33
3.1 PCF Call-by-Name Evaluation ............................................................................. 39
3.2 PCF Stack Machine ............................................................................................ 42
4.1 Type System for System $\mathcal{L}_{\text{rec}}$ ............................................................. 50
4.2 CBN evaluation for System $\mathcal{L}_{\text{rec}}$ ............................................................ 51
4.3 $\mathcal{L}_{\text{rec}}$ Stack Machine ............................................................................... 56
5.1 PCF compilation into $\mathcal{L}_{\text{rec}}$ .................................................................... 62
5.2 Results Comparison ........................................................................................... 66
5.3 $\mathcal{L}_{\text{rec}}^{\text{nat}}$ Call-by-Name Evaluation ......................................................... 67
5.4 $\mathcal{L}_{\text{rec}}^{\text{nat}}$ Stack Machine ............................................................................ 67
5.5 PCF compilation into $\mathcal{L}_{\text{rec}}^{\text{nat}}$ ................................................................. 68
5.6 Results Comparison with $\mathcal{L}_{\text{rec}}^{\text{nat}}$ ......................................................... 69
5.7 Results Comparison Between Compilations ....................................................... 69
List of Figures

3.1  **PCF Implementation** ............................................. 40

4.1  System $L_{rec}$ Implementation .................................... 52

5.1  Abstract time measure results ...................................... 70

5.2  Abstract time measure results for the encodings ............... 71
Chapter 1

Introduction

The goal of this thesis is to study operational semantics for the linear language, $L_{rec}$, and to compare it with standard operational semantics for functional languages based on $\lambda$-calculus. System $L_{rec}$ [Alves 11] is defined as a linear $\lambda$-calculus extended with numbers, pairs and an unbounded linear recursor, which gives a Turing-complete system. It is used, in this thesis, as a base for creating an operational semantics for linear languages. We use System $L_{rec}$ because we believe it will show better results as an intermediate code than the other linear languages due to its unbounded recursor.

We will compare it with PCF, an also Turing-complete programming language with ground types, roughly close to $\lambda$-calculus with a fixed point operator, which makes it the ideal programming language to compare with System $L_{rec}$.

The PCF language (Programming Language for Computable Functions) was introduced by Plotkin [Plotkin 77], as a programming language for computational functions based on LCF [Scott 93]. It is a variant of the typed $\lambda$-calculus with the addition of ground types int and bool and a restrict relation of conversion.

In order to obtain a Turing-complete system, PCF extends the simply typed $\lambda$-calculus with a fixed point operator. However, it is based on the existence of a non-linear condition, which discards the possibility of infinite computation on the branches. On the other hand, System $L_{rec}$ uses an unbound recursor with a built-in test over pairs that allows the encoding of finite iterations and minimization, obtaining a Turing-complete linear $\lambda$-calculus.

Starting from these two languages, the contributions of this thesis are:

- An implementation of call-by-name interpreter for the PCF language, and also
1.1 Motivation and Related Work

Syntactically linear languages have several implementation advantages: knowing that a function is linear is a property that a compiler can take advantage to optimize code, as well as in program analysis. The linear $\lambda$-calculus is a subset of the $\lambda$-calculus in which no variable occurs more than once in any subterm of any term, and for any function the argument occurs in the body of the function. Although it has very nice computational properties, it has a very limited computational power: every term reduces in time linear to its size.

There are several works that explore linearity in the $\lambda$-calculus, many of those derived from Girard’s Linear Logic [Girard 87] (a logic where hypothesis are looked as resources, which are consumed by proofs, contrary to classical logic where hypothesis can be used as many times as needed). In particular an extension of the linear $\lambda$-calculus with bounded iterator of closed functions proved to capture exactly the class of primitive recursive functions [Lago 09]. In [Alves 10] another extension of the linear $\lambda$-calculus with bounded iteration was presented, that imposed a closed-on-reduction condition on iterated functions which proved to be computationally equivalent to
Gödel’s System T [Girard 89] (Gödel’s System T is a simply typed \( \lambda \)-calculus with numbers and booleans, a condition and a bounded recursor). We point out to [Asperti 98, Girard 98, Asperti 02, Lafont 04, Terui 01, Baillot 04] for the definition of several other linear calculi for capturing specific complexity classes. There is also previous work that uses linear types to characterize computations with time bounds [Hofmann 99].

### 1.2 Overview of the Thesis

The rest of the thesis is organized in the following way:

**Chapter 2: Background** In this chapter, a background is given in \( \lambda \)-calculus, Simple Typed \( \lambda \)-calculus and Operational Semantics, explaining its basic concepts and notions. Also, a brief explanation of the Krivine abstract machine for the \( \lambda \)-calculus, which uses a call-by-name evaluation.

**Chapter 3: PCF Language** Here we give the syntax and type system for the PCF language, an abstract stack-machine for a call-by-name evaluation of the language, and we explain the implementation of the evaluation and of the machine.

**Chapter 4: System \( \mathcal{L}_{\text{rec}} \)** A syntax and type system for System \( \mathcal{L}_{\text{rec}} \) is given in Chapter 4 along with an abstract stack-machine, also with a call-by-name evaluation. In the end of this chapter, an implementation of System \( \mathcal{L}_{\text{rec}} \) is also given.

**Chapter 5: Compiling** In this chapter we show the encoding from PCF to System \( \mathcal{L}_{\text{rec}} \), its implementation and the comparison of the results obtained. Also, a variant of System \( \mathcal{L}_{\text{rec}} \) with built-in naturals is given, with its implementation and a comparison the results obtained previously against this new implementation.

**Chapter 6: Conclusions** In this final chapter we state the conclusions of this work along with the outline of future work.
1.2. OVERVIEW OF THE THESIS
Chapter 2

Background

2.1 Lambda-Calculus

In this section we briefly describe the λ-calculus, defined by Alonzo Church [Church 32] in 1932, which is a Turing-complete computational model that has a very simple syntax (variables, function abstraction and function application) and a main rewrite rule (β-reduction), based on the notion of substitution.

2.1.1 Syntax

The λ-calculus models the definition and application of functions, based on the notion of substitution. The application \( MN \) represents the application of the function represented by \( M \) to the argument represented by \( N \). The abstraction \( \lambda x.M \) represents the function \( f \) such that \( f(x) = M \). The application of this function \( f \) to \( N \) results in the substitution of \( x \) for \( N \) in \( M \), i.e. \( f(N) \).

**Definition 2.1.1. (λ-calculus)** Let \( \mathbb{V} \) be an infinite set of variables. The set of λ-terms \( \Lambda \), is build from the set \( \mathbb{V} \) using application and abstraction in the following way:

\[
x \in \mathbb{V} \quad \Rightarrow \quad x \in \Lambda
\]

\[
(\text{Application}) \quad M, N \in \Lambda \quad \Rightarrow \quad (MN) \in \Lambda
\]

\[
(\text{Abstraction}) \quad M \in \Lambda, x \in \mathbb{V} \quad \Rightarrow \quad (\lambda x.M) \in \Lambda
\]

**Notation:** We use the symbol \( \equiv \) to denote syntactic equality between terms.
Since we consider that the application is left associative, and the abstraction is right associative, we will use the following abbreviations:

- \((M_1M_2 \ldots M_n) \equiv (\ldots (M_1M_2) \ldots M_n)\)

- \((\lambda x_1x_2 \ldots x_n. M) \equiv (\lambda x_1.(\lambda x_2.(\ldots (\lambda x_n.M)\ldots)))\)

Which means that the term \((\lambda x.(\lambda z.((xz)(\lambda y.(yx)))))\) can be written as \((\lambda xz.xz(\lambda y.yx))\).

### 2.1.2 Variables and Substitution

We will distinguish the type of variable occurrences, formalizing the concepts of free and bound variables of a term.

**Definition 2.1.2. (Free Variables)** Let \(M \in \Lambda\), the set \(FV(M)\) of free variables of \(M\) is inductively defined as follows:

- \(FV(x) = \{x\}\)
- \(FV(MN) = FV(M) \cup FV(N)\)
- \(FV(\lambda x.M) = FV(M) \setminus \{x\}\)

**Definition 2.1.3. (Bound Variables)** Let \(M \in \Lambda\), the set \(BV(M)\) of bound variables of \(M\) is inductively defined as follows:

- \(BV(x) = \emptyset\)
- \(BV(MN) = BV(M) \cup BV(N)\)
- \(BV(\lambda x.M) = BV(M) \cup \{x\}\)

A \(\lambda\)-term is closed if and only if \(FV(M) = \emptyset\).

Note that the sets of free and bound variables of a term are not necessarily disjoint, for example, \(x\) occurs both free and bound in the term \(x(\lambda xy.x)\).

**Definition 2.1.4. (Substitution)** The result of substituting the free occurrences of \(x\) by \(L\) in \(M\) (denoted by \(M[L/x]\)) is defined as:
\[
x[L/y] = \begin{cases} 
L & \text{if } x \equiv y \\
x & \text{otherwise}
\end{cases}
\]

\[
(MN)[L/y] = (M[L/y])(N[L/y])
\]

\[
(\lambda x.M)[L/y] = \begin{cases} 
(\lambda x.M) & \text{if } x \equiv y \\
(\lambda x.M[L/y]) & \text{otherwise}
\end{cases}
\]

One problem that can arise from variable substitution is the capture of free variables. As an example, consider the \(\lambda\)-term \((\lambda xy.x)\). We would expect to obtain \(N\) when applied to two arguments \(N\) and \(P\). However, if \(N \equiv y\) then we would get \((\lambda xy.x)[N/x][P/y] \equiv P\), because the free occurrence of \(x\) becomes a bound occurrence of \(y\). To avoid this problem, the substitution \(M[N/x]\) should only be permitted if \(x\) does not occur free in any subterm of \(M\) of the form \(\lambda y.P\), and \(y \in FV(N)\), in which case we say that \(x\) is substitutable by \(N\) in \(M\).

This condition is ensured if the set of bound variables of \(M\) is disjoint from the set of free variables of \(N\):

\[
BV(M) \cap FV(N) = \emptyset
\]

We can always rename the bound variables in \(M\) to ensure that this condition is guarantied. This change in the bound variables is called \(\alpha\)-conversion.

**Definition 2.1.5. (\(\alpha\)-conversion)** A change of a bound variable \(x\) in a term \(M\) is the substitution of all subterms of \(M\) of the form \((\lambda x.N)\) by \(\lambda y.(N[y/x])\), where \(y\) does not occur in \(N\).

This change of bound variables does not alter the function, in fact, it preserves the meaning of the term. This notion is called \(\alpha\)-congruence.

**Definition 2.1.6. (\(\alpha\)-congruence)** The terms \(M\) and \(N\) are \(\alpha\)-congruent, (notation \(M \equiv_\alpha N\)), if \(N\) can be obtained from \(M\), by a series of changes of bound variables, and vice-versa.

For example:

\[
\lambda x. xy \equiv_\alpha \lambda z. zy \not\equiv_\alpha \lambda y. yy.
\]

Assume from now on, that the sets of free and bound variables are always disjoint, which is known as the Barendregt’s name convention [Barendregt 97]. Therefore any substitution \(M[N/x]\) is valid. Also, do not differentiate terms that are \(\alpha\)-congruent (for instance \(\lambda x. x \equiv_\alpha \lambda y. y\)).
2.1.3 The $\beta$-reduction

We will now describe a reduction relation on $\Lambda$ [Barendregt 84], that together with $\Lambda$, defines the reduction system for which some properties will be discussed.

Definition 2.1.7. ($\beta$-reduction) The following contraction rule defines the notion of $\beta$-reduction on $\Lambda$:

$$\beta : (\lambda x. M) N \rightarrow M[N/x], \ M, N \in \Lambda$$

A $\lambda$-term of the form $(\lambda x. M) N$ is called a $\beta$-redex and $M[N/x]$ is its $\beta$-contractum.

A $\lambda$-term $M$ is reduced to $N$ in one $\beta$-reduction step, and is written as $M \rightarrow^1_\beta N$, if $N$ can be obtained by substituting on $M$ a $\beta$-redex by its $\beta$-contractum. We write $M \rightarrow^*_\beta N$, for the reflexive and transitive closure of $\rightarrow^*_{\beta}$.

Definition 2.1.8. (Normal Form) A term that does not contain any $\beta$-redex, does not admit any $\beta$-reductions, and is therefore said to be in $\beta$-nf or just normal form.

For example, the term $(\lambda x. z)((\lambda xy.xy)(\lambda x.x))$ is not in $\beta$-nf form since it has a $\beta$-redex $(\lambda xy.xy)(\lambda x.x)$. The term admits a $\beta$-nf, which is $z$.

The $\beta$-reduction system is Church-Rosser [Church 36, Barendregt 84], which ensures that reduction is confluent.

Theorem 2.1.1. (Church-Rosser) Let $M$ be a $\lambda$-term, if $M \rightarrow^*_{\beta} N_1$ and $M \rightarrow^*_{\beta} N_2$ then, there is a $N$ such that $N_1 \rightarrow^*_{\beta} N$ and $N_2 \rightarrow^*_{\beta} N$.

A $\lambda$-term that has a normal form, which admits an infinite reductions sequence, reaches the normal form if all the subterms of $M$ that do not have normal forms, are erased.

2.1.4 Reduction Strategies

The goal of this section is to define and clarify the relation between programming language concepts such as call-by-name and call-by-value, and $\lambda$-calculus concepts such as Normal Order Reduction and Applicative Order Reduction.

We will start by defining a reduction strategy, which is a procedure that finds the next redex to be contracted. That redex can be:

- **Leftmost-Outermost**: the leftmost redex not contained in any other redex;
### 2.1. LAMBDA-CALCULUS

#### Table 2.1: Normal Forms, where $M_i \in \Lambda$ (Definition 2.1.1)

- **Leftmost-Innermost**: the leftmost redex that does not contain a redex.

The are other redexes, e.g. **Rightmost-Innermost**, **Rightmost-Outermost**, etc., but in the interest of this thesis, they will not be defined.

Note that a redex is to the left of any other redex if his $\lambda$-abstractor is further to the left in the syntactic form of the term. Also note that, in a reduction of a **Leftmost-Outermost** redex, if in an application $(M_1 M_2)$, if $M_1 \rightarrow^1_{\beta} (\lambda x.M)$ then the term $((\lambda x.M)M_2)$ has to be reduced before any redex in $M$ (otherwise it would not be **Outermost**).

Although the definition for Normal Form has already been introduced (Definition 2.1.8), in order to really comprehend the strategies we are going to present, we will have to explore other concepts of normal forms such as **Weak Normal Form**, **Head Normal Form** and **Weak Head Normal Form** (Table 2.1, where $E$ is the term in the relevant normal form). We will distinguish between strategies that choose the next redex to be contracted and strategies that prevent some redex from being contracted, since the set of normal forms will differ depending on the strategy used. Note that not all the strategies reduce under abstractions as it is usual in Functional Programming Languages. The main difference between the strategies relies on whether we reduce under abstractions and whether we reduce the arguments before substitution (in strict languages) or not (in non-strict languages).

We will consider four different reduction strategies:

1. **Call-by-Name Reduction**: consists on reducing the **Leftmost-Outermost** redex that is not under any $\lambda$-abstraction, because this strategy does not reduce under abstractions. It reduces to a term in **Weak Head Normal Form**, if the term has one;

2. **Normal Order Reduction**: also consists on reducing the **Leftmost-Outermost**
2.1. LAMBDA-CALCULUS

redex, with the difference that this strategy reduces under λ-abstractions. It reaches to a term in Normal Form, if the term has one;

3. **Call-by-Value Reduction**: consists on reducing the Leftmost-Innermost redex that is not under any λ-abstraction, since this strategy, as the first one, does not reduce under abstractions. It reduces to a term in Weak Normal Form, if the term has one;

4. **Applicative Reduction**: also consists on reducing the Leftmost-Innermost redex but as it was for the second strategy, it reduces under λ-abstraction, being this what differentiates it from the previous one. This strategy reaches a term in Head Normal Form, if the term has one.

Note that the **Normal Order Reduction**, has the distinct property of being normalizing, in the sense that it reaches the β-normal form if the term has one.

**Definition 2.1.9** (Strong Normalization). *If for any λ-term M, all the reduction strategies find its normal form, then M is strongly normalisable.*

A **call-by-name** evaluation is going to be presented for the PCF language (Chapter 3) and System $L_{rec}$ (Chapter 4), and a **call-by-value** evaluation is also going to be presented for System $L_{rec}$.

For further knowledge on this subject one can read [Sestoft 02].

2.1.5 **Linear Calculi**

Several subsystems of the λ-calculus can be obtained by restricting the set of terms. Some of those subsystems are the $λ_I$-calculus, where function parameters have to occur in the body of the function, the **affine** λ-calculus, where function parameters occur at most one time in the body of the function, and, the one we are going to present in detail, the **linear** λ-calculus.

In the linear λ-calculus, every variable in every term $M$ occurs free exactly once in any subterm of $M$.

**Definition 2.1.10.** Let $\mathbb{V}$ be an infinite set of variables. The set of linear λ-terms, $\Lambda_\mathcal{L}$ is inductively defined from $\mathbb{V}$ in the following way:

\[
\begin{align*}
    x & \in \mathbb{V} \quad \Rightarrow \quad x \in \Lambda_\mathcal{L} \\
    M, N & \in \Lambda_\mathcal{L}, FV(M) \cap FV(N) = \emptyset \quad \Rightarrow \quad (M \ N) \in \Lambda_\mathcal{L} \quad \text{(Application)} \\
    M & \in \Lambda_\mathcal{L}, x \in fv(M) \quad \Rightarrow \quad (\lambda x. M) \in \Lambda_\mathcal{L} \quad \text{(Abstraction)}
\end{align*}
\]
All the notions defined for Λ, are defined in an analogous way for Λ. In Chapter 4, we will present a system which extends the linear λ-calculus with natural numbers, pairs and a recursor.

2.2 The Simple Typed Lambda Calculus

In the previous section we discussed the type-free λ-calculus. In this section we will present a simple type system for λ-calculus, the Curry Type System [Curry 34], which initial motivation was to avoid paradoxical uses of the untyped calculus [Church 40]. This system was first studied for the theory of combinators, and was then modified for the λ-calculus in [Curry 58]. The definitions and proofs of results in this section can be found in [Barendregt 84].

In this thesis we will focus on typed calculi, such as PCF and System $L_{\text{rec}}$, which both extent the simply typed λ-calculus that we are going to present in this section. We will start by defining the set of types for the simply typed λ-calculus.

**Definition 2.2.1.** Let $\mathbb{V}_T$ be an infinite set of type variables. The set of simple types, $T_c$, is inductively defined from $\mathbb{V}_T$ in the following way:

\[
\begin{align*}
\alpha &\in \mathbb{V}_T \Rightarrow \alpha \in T_c \\
\tau, \tau' &\in T_c \Rightarrow (\tau \rightarrow \tau') \in T_c
\end{align*}
\]

**Notation** Since the type constructor $\rightarrow$ is right associative, if $\tau_1, \ldots, \tau_n \in T_c$, then

\[\tau_1 \rightarrow \tau_2 \rightarrow \ldots \rightarrow \tau_n\]

is an abbreviation for:

\[\tau_1 \rightarrow (\tau_2 \rightarrow \ldots \rightarrow (\tau_{n-1} \rightarrow \tau_n))\].

**Definition 2.2.2.** If $x$ is a term variable in $\mathbb{V}_T$ and $\tau$ is a type in $T_c$ then:

- A statement is of the form $M : \tau$, where the type $\tau$ is called the predicate, and the variable $x$ is called the subject of the statement.

- A declaration is a statement where the subject is a term variable.

- A basis $\Gamma$ is a set of declarations where all the subjects are distinct.
Definition 2.2.3. If $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_2\}$ is a basis, then:

- $\Gamma$ is a partial function, with domain, denoted $\text{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$.
- We define $\Gamma_x$ as $\Gamma \setminus \{x : \tau\}$

Definition 2.2.4. In the Curry Type System, we say that $M$ has type $\tau$ given the basis $\Gamma$, and write $\Gamma \vdash_C M : \tau$, if $\Gamma \vdash_C M : \tau$ can be obtained from the following derivation rules:

$$
\begin{align*}
\Gamma_x \cup \{x : \tau\} &\vdash_C x : \tau \quad \text{(Axiom)} \\
\Gamma \vdash_C x_1 : \tau_1 \quad \Gamma \vdash_C M : \tau_2 &\vdash_C \Gamma \vdash_C x_1 : \tau_1 \rightarrow \tau_2 \quad \text{($\rightarrow$ Intro)} \\
\Gamma \vdash_C M : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash_C N : \tau_1 &\vdash_C \Gamma \vdash_C MN : \tau_2 \quad \text{($\rightarrow$ Elim)}
\end{align*}
$$

Example 2.2.1. For the $\lambda$-term $(\lambda xy.x)(\lambda x.x)$ the following derivation is obtained in the Curry Simple Type System:

$$
\begin{align*}
\{x : \alpha \rightarrow \alpha, y : \beta\} &\vdash_C x : \alpha \rightarrow \alpha \\
\{x : \alpha \rightarrow \alpha\} &\vdash_C \lambda y.x : \beta \rightarrow \alpha \rightarrow \alpha \\
\{x : \alpha\} &\vdash_C x : \alpha \\
\vdash_C \lambda xy.x : (\alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \alpha \rightarrow \alpha \\
&\vdash_C \lambda x.x : \alpha \rightarrow \alpha \\
&\vdash_C (\lambda xy.x)(\lambda x.x) : \beta \rightarrow \alpha \rightarrow \alpha
\end{align*}
$$

Theorem 2.2.1. (Subject Reduction) Let $M$ be a $\lambda$-term, and $M \rightarrow^* M'$, then $\Gamma \vdash_C M : \tau \Rightarrow \Gamma \vdash_C M' : \tau$

The implication in the other direction, called subject expansion, does not hold for this system. Take, for example, $(\lambda xy.y)(\lambda z.zz) \rightarrow^*_\beta (\lambda y.y)$, then $(\lambda y.y)$ is typable, and $(\lambda xy.y)(\lambda z.zz)$ is not.

Theorem 2.2.2. Strong normalization Let $M$ be a $\lambda$-term.

$$
\Gamma \vdash_C M : \tau \Rightarrow M \text{ is strongly normalisable.}
$$

Notice that, the implication in the other direction does not hold. There are many strongly normalisable $\lambda$-terms that are not typable in this system. For example, the term $\lambda x.xx$ is not typable in the Curry Simple Type System (since to type the subterm $xx$, the variable $x$ as to be both of type $\alpha$ and $\alpha \rightarrow \beta$) although it is strongly normalisable since it is in normal form.
2.3 Operational Semantics and Abstract Machine

In this section we will present two operational semantics for \( \lambda \)-calculus following call-by-name and call-by-value evaluations, respectively. We will also present an abstract machine with a call-by-name evaluation.

### 2.3.1 Call-by-Name and Call-by-Value

Since different reduction strategies have already been introduced, we will now present evaluations for a call-by-name and call-by-value reduction strategy.

The operational semantics with call-by-name evaluation follows the rules presented bellow:

\[
\begin{align*}
\text{Value} & : V \downarrow V \\
\text{App} & : S \downarrow \lambda x. U \quad U[T/x] \downarrow V \\
& \quad ST \downarrow V
\end{align*}
\]

The operational semantics with call-by-value evaluation follow the rules presented bellow:

\[
\begin{align*}
\text{Value} & : V \downarrow V \\
\text{App} & : S \downarrow \lambda x. U \quad T \downarrow V' \quad U[V'/x] \downarrow V \\
& \quad ST \downarrow V
\end{align*}
\]

Once again, the only difference relies on the application rule, which, in this last case, evaluates the second argument before applying the substitution. Note that the set of values \( V \) is composed by variables and \( \lambda \)-abstractions.

### 2.3.2 Krivine Machine

Since later on we will present two stack machines for the \textbf{PCF} language and System \( \mathcal{L}_{\text{rec}} \), in this section will be presented an introduction to the Krivine stack machine.
2.3. OPERATIONAL SEMANTICS AND ABSTRACT MACHINE

[Krivine 07], a simple lazy machine for the $\lambda$-calculus.

The Krivine Machine is a simple stack-based call-by-name evaluator restricted to $\lambda$-terms. The evaluation of this machine follows a call-by-name strategy, since on the application rule ($M N$), the machine pushes the term $N$ on the stack unevaluated, and only when the associated variable is referenced does it evaluate the later.

In the following definition for the Krivine machine, a state is defined as a pair $(C, S)$, where $S$ is a stack of $\lambda$-terms and $C$ is a closure, that pairs free variables with environments in order to bind them, since subterms may contain free variables. The environment $\rho$, of those closures is a function that associates variables to closures.

**Definition 2.3.1 (Krivine Configuration).** The state of the Krivine machine is a pair $(C, S)$, such that:

\[
C ::= \langle M, \rho \rangle, \quad \text{where } M \text{ is a } \lambda\text{-term}
\]
\[
\rho ::= \emptyset | [x \mapsto C], \quad \text{where } \emptyset \text{ is the empty environment}
\]

We consider the following special cases for states:

- **Inicial State**: $(\langle M, \emptyset \rangle, \emptyset)$
- **Final State**: $(\langle V, \emptyset \rangle, \emptyset)$, where $V$ is a Value or a non-reducible term

The execution of the machine emulates the following rules:

- Given a **variable**: if this variable is in the environment $\rho$ then the machine returns the closure associated with it;

- Given an **abstraction ($\lambda x. M$) with a closure on top of the stack**: it pushes the closure in the environment associated with the variable of the abstraction ($x$) and then evaluates $M$ with that environment;

- Given an **application ($M N$)**: it pushes the argument $N$ associated with the environment of the application to the stack and evaluates $M$ with that same environment.

The execution consists in constantly updating the closure and the stack. The operational semantics of the Krivine machine can be seen in Table 2.2, where for every pair of states $s, t$, there is a rule such that $s \Rightarrow t$.

If an execution sequence does not end in a value $V$, then the final state will be the
last reduction possible. Note that the Abs rule corresponds to the substitution rule (Definition 2.1.4).

**Example 2.3.1.** To better understand the execution of the machine, we will present an example using the term \((\lambda x. (\lambda y. y) x) 2\):

\[
\begin{align*}
((\lambda x. (\lambda y. y) x) 2), &\emptyset, \emptyset) \downarrow \text{App} \\
((\lambda x. (\lambda y. y) x), &\emptyset, \langle 2, \emptyset \rangle : \emptyset) \downarrow \text{Abs} \\
((\lambda y. y) x, &\langle x \mapsto \langle 2, \emptyset \rangle \rangle), \emptyset) \downarrow \text{App} \\
((\lambda y. y), &\langle x \mapsto \langle 2, \emptyset \rangle \rangle), \langle x, \langle x \mapsto \langle 2, \emptyset \rangle \rangle \rangle : \emptyset) \downarrow \text{Abs} \\
\langle y, &\langle y \mapsto \langle x, \langle x \mapsto \langle 2, \emptyset \rangle \rangle \rangle \rangle \rangle), \emptyset) \downarrow \text{Var} \\
\langle x, &\langle x \mapsto \langle 2, \emptyset \rangle \rangle \rangle), \emptyset) \downarrow \text{Var} \\
\langle 2, &\emptyset \rangle), \emptyset)
\end{align*}
\]

The Krivine machine is considered to be inefficient due to the repeated evaluations of the same operand. However, it is a well-known standard machine for languages implementing call-by-name evaluation, which is the case of PCF. Based on the Krivine machine we will present two stack machines, for the PCF language and for System \(\mathcal{L}_{rec}\).

Another well-known stack machine is the SECD [Landin 64], designed for call-by-value evaluation. We will not implement a call-by-value evaluator since we are interested in comparing the semantics of two languages following a call-by-name strategy.
2.3. OPERATIONAL SEMANTICS AND ABSTRACT MACHINE

Operational Semantics for Linear Languages
Chapter 3

PCF

In this section we will describe the PCF language (Programming Language for Computable Functions), a variant of the typed λ-calculus with the addition of ground types \texttt{int} and \texttt{bool} and a restricted conversion relation. The implementation of this language, using Haskell [Has 03], will also be described.

The PCF language was originally introduced by Gordon Plotkin [Plotkin 77], inspired by the language LCF (Language for Computable Functions [Scott 93]).

3.1 Syntax

We will start by defining the set of types for the PCF language.

Definition 3.1.1. The set of PCF types, $\mathbb{T}_p$, is inductively defined in the following way:

- $\alpha \in \{\texttt{int}, \texttt{bool}\} \Rightarrow \alpha \in \mathbb{T}_p$
- $\tau, \tau' \in \mathbb{T}_p \Rightarrow (\tau \rightarrow \tau') \in \mathbb{T}_p$

Identically to the notation described in Chapter 2, we will assume that types associate to the right.

For each type $\tau \in \mathbb{T}_p$, we have an infinite set of typed variables and a collection of typed constants:

$$V_\tau = \{x^\tau, y^\tau, z^\tau, \ldots\} \quad \mathbb{C}_\tau = \{c^\tau\}$$

We will now define the set of constants for the PCF language.
Definition 3.1.2. The set of constants, $\mathbb{C}_\tau$, is defined as follows:

- $\bar{n}$: int for $n = 0, 1, 2, \ldots$
- $\text{tt, ff}$: bool
- $\text{succ, pred}$: int $\rightarrow$ int
- $\text{iszer}$: int $\rightarrow$ bool

And for each type $\tau \in \mathbb{T}_p$:

- $\text{cond}_\tau$: bool $\rightarrow$ $\tau$ $\rightarrow$ $\tau$ $\rightarrow$ $\tau$
- $\text{Y}_\tau$: ($\tau$ $\rightarrow$ $\tau$) $\rightarrow$ $\tau$

Definition 3.1.3. The set $\Lambda_p$ of typed PCF terms $M : \tau$, with $\tau \in \mathbb{T}_p$ is defined as:

- $x^\tau : \tau \in \Lambda_p$ (variables)
- $c^\tau : \tau \in \Lambda_p$ (constants)
- $M : \tau_1 \in \Lambda_p \Rightarrow \lambda x^\tau . M : \tau \rightarrow \tau_1 \in \Lambda_p$ (abstraction)
- $M : \tau_1$ $\rightarrow$ $\tau_2$ $\in$ $\Lambda_p$, $N : \tau_1$ $\in$ $\Lambda_p$ $\Rightarrow$ $MN : \tau_2$ $\in$ $\Lambda_p$ (application)

Again, we assume that application associates to the left. Some examples of PCF terms:

- $\text{pred} \bar{6}$ : int
- $\text{cond}_{\text{int}} (\text{iszer} (\text{succ} \bar{2})) \bar{6}$ : int $\rightarrow$ int

As another example, consider the following recursive definition of factorial:

$f(x) = \text{if } x=0 \text{ then } 1 \text{ else } x*f(x-1)$

One can easily encode it as a PCF term:

$\text{Y}_{\text{int}} (\lambda f^{\text{int}} \text{ int} . \lambda x^{\text{int}} . \text{cond}_{\text{int}} (\text{iszer} x) \bar{1} \text{ (mult } x (f (\text{pred } x))))$

Where,

$\text{mult} = \text{Y}_{\text{int}} (\lambda f^{\text{int}} \text{ int} \rightarrow \text{ int} . \lambda m^{\text{int}} . \lambda n^{\text{int}} . \text{cond}_{\text{int}} (\text{iszer } m) \bar{0} \text{ (add } f (\text{pred } m) n) n)$
### 3.2 Implementation

In Figure 3.1, one can see a diagram of the **PCF** implementation, starting by an analysis of the language, followed by a call-by-name and abstract machine interpreters, which evaluate the expression into a normal form, if one exists.

The implementation of this language started with a parser in Happy [Gill 01], that transformed the **PCF** input into a data structure in Haskell (Code 3.1). Then,
3.2. IMPLEMENTATION

![Diagram of PCF Implementation]

Figure 3.1: PCF Implementation

a call-by-name evaluation function was created, and finally the stack machine was implemented.

Code 3.1: PCF Language Data Structure

```
ExpPCF :: = Var String | Nat Int | TT | FF
    | IsZ | Succ | Pred | Cond | PtoF
| Lambd String ExpPCF
| App ExpPCF ExpPCF
    deriving (Show, Eq)
```

In the data structure created in Haskell (Code 3.1), the structures Var String and Nat Int, hold the names of the variables and the values of the integers, respectively; the TT and FF serve as the values tt and ff of the PCF language; the PtoF is the equivalent of PCF fixed point, Y. For the application and λ-terms, the structures Lambd String ExpPCF and App ExpPCF ExpPCF were created.

Finally, the structures IsZ, Succ, Pred and Cond will serve as the PCF constants iszer, succ, pred and cond.

3.2.1 Call-by-Name Interpreter

The call-by-name evaluation function, which complete code can be seen in Appendix (Code A.1), will be explained in several parts. This function receives a PCF expression, returning the evaluation of that same expression. We started by implementing the evaluation of values (Code 3.2) that cannot be reduced, which means that the function will return them as they are.
3.2. IMPLEMENTATION

We consider as values variables, abstractions and the PCF constants \texttt{iszer}, \texttt{pred}, \texttt{succ}, \texttt{cond}, \texttt{Y}. And since these PCF functions were implemented as values and do not receive arguments, we use the application to evaluate them with a term. For example, the term \texttt{cond BN} is translated to \texttt{App (App (App cond b)m)n}, as one can see in Code 3.3.

The eval function evaluates following the rules on Table 3.1.

In order to implement the call-by-name evaluation function, there was the necessity to create a substitution function (Definition 2.1.4, Code A.2) for the application of a \texttt{\lambda}-term.

| eval :: ExpPCF → ExpPCF |
| eval (Var a) = (Var a) |
| eval (Lambd x m) = (Lambd x m) |
| eval (Nat x) = (Nat x) |
| eval Succ = Succ |
| eval Pred = Pred |
| eval Cond = Cond |
| eval IsZ = IsZ |
| eval PtoF = PtoF |

| eval (App PtoF m) = eval (App m (App PtoF m)) |
| eval (App (Lambd x m) n) = eval (subst \( t \) m x n) |
| eval (App (App (App Cond TT) m) n) = eval m |
| eval (App (App (App Cond FF) m) n) = eval n |
| eval (App (App Succ m) = case (eval m) of |
| (Nat n) → (Nat (n+1)) |
| (t) → (App Succ t) |
| eval (App Pred m) = case (eval m) of |
| (Nat (n+1)) → (Nat n) |
| (t) → (App Pred t) |
| eval (App IsZ (Nat n)) = if (n=0) then TT else FF |
| eval (App IsZ m) = eval (App IsZ (eval m)) |
| eval (App s t) = eval_aux (App (eval s) t) |
| eval t = t |
(app) \((M \cdot N, S_p) \Rightarrow (M, N; S_p)\)
(abs) \(((\lambda x. M), N; S_p) \Rightarrow (M[N/x], S_p)\)
(fixed point) \((Y, M; S_p) \Rightarrow (M, (Y M); S_p)\)
(cond1) \((\text{cond}, M; N_1 : N_2; S_p) \Rightarrow (M, \text{COND}(N_1, N_2); S_p)\)
(cond2) \((\text{tt}, (\text{COND}(N_1, N_2)); S_p) \Rightarrow (N_1, S_p)\)
(cond3) \((\text{ff}, (\text{COND}(N_1, N_2)); S_p) \Rightarrow (N_2, S_p)\)
(succ1) \((\pi, \text{SUCC}; S_p) \Rightarrow (\text{succ}(M); S_p)\)
(succ2) \((\text{succ}, M; S_p) \Rightarrow (M, \text{SUCC}; S_p)\)
(pred1) \((\bar{0}, \text{PREV}; S_p) \Rightarrow (0, S_p)\)
(pred2) \((\bar{n} + 1, \text{PREV}; S_p) \Rightarrow (\bar{n}, S_p)\)
(pred3) \((\text{pred}, M; S_p) \Rightarrow (M, \text{PREV}; S_p)\)
(iszer1) \((\bar{0}, \text{ISZER}; S_p) \Rightarrow (\text{tt}, S_p)\)
(iszer2) \((\bar{n} + 1, \text{ISZER}; S_p) \Rightarrow (\text{ff}, S_p)\)
(iszer3) \((\text{iszer}, M; S_p) \Rightarrow (M, \text{ISZER}; S_p)\)

Table 3.2: PCF Stack Machine

Code 3.4: Auxiliary Evaluation Function

```plaintext
eval_aux :: ExpPCF -> ExpPCF
eval_aux (App (Lambda x u) t) = eval (subst u x t)
eval_aux (App s t) = (App s t)
```

We also created an auxiliary function (Code 3.4) that looks at the first term of the application: if it is a \(\lambda\)-term, it evaluates the substitution for that term, if it is not, and since it has already been evaluated and is therefore in normal form, our auxiliary function returns the application term as it is, because the call-by-name evaluation of the application does not reduce the second term. This way, we avoid the problem of trying to evaluate the first term when it is not evaluable.

### 3.2.2 Stack Machine

In this last section we will show how the PCF language can be implemented as a stack machine.

We will start by defining the elements of the stack followed by the definition of the machine configuration.

**Definition 3.2.1** (Stack Machine Elements). *Let \(S_p\) be the set of the elements of the stack machine, which is an extension of the PCF terms, then:*
3.2. IMPLEMENTATION

\[ \alpha \in \Lambda \Rightarrow \alpha \in S_p \]
\[ M, N \in \Lambda \Rightarrow \text{COND}(M, N) \in S_p \]

In order to make the call-by-name evaluation possible in the machine, the following terms are also added:

\[ \text{SUCC, PRED, ISZERO} \in S_p \]

**Definition 3.2.2 (PCF Stack Machine).** The state of the PCF stack machine is a pair \((M, S_p)\), such that:

- \(M\) is a PCF term
- \(S_p\) is a stack of PCF extended terms

We consider the following special cases for states:

- **Inicial State** : \((M, \varepsilon)\)
- **Final State** : \((V, S_p)\), where \(V\) is a Value

The basic principle of the machine is to find the next redex, using the stack \(S_p\) to store future computations. And for every pair of states \(s, t\), there is a rule, based on a call-by-name strategy, such that \(s \Rightarrow t\) (Table 3.2).

The execution of the stack terminates when a final state is reached, which means that no other reduction is possible.

**Example 3.2.1.** To better understand the execution of the machine, we will present an example using the term \(((\lambda m. (\lambda n. (((\text{cond} \text{ iszer} m)) n)(\text{pred} m)))) 2) 3)\:

\[
(((\lambda m. (\lambda n. (((\text{cond} \text{ iszer} m)) n)(\text{pred} m)))) 2) 3, \varepsilon) \\
\downarrow \text{app} \\
((\lambda m. (\lambda n. (((\text{cond} \text{ iszer} m)) n)(\text{pred} m)))) 2, 3 : \varepsilon \\
\downarrow \text{app} \\
(\lambda m. (\lambda n. (((\text{cond} \text{ iszer} m)) n)(\text{pred} m))), 2 : 3 : \varepsilon \\
\downarrow \text{abs} \\
(((\lambda n. (((\text{cond} \text{ iszer} 2)) n)(\text{pred} 2))), 3 : \varepsilon \\
\downarrow \text{abs} \\
(((\text{cond} \text{ iszer} 2)) 3)(\text{pred} 2), \varepsilon \\
\downarrow \text{app} \\
(((\text{cond} \text{ iszer} 2)) 3), (\text{pred} 2) : \varepsilon)
\]
Considering that later on, we will show a stack machine for System $L_{rec}$ (Section 4.3), there was also the necessity to implement it on PCF in order to make a better comparison between the two languages.

Code 3.5: Data Structure for the Stack

```haskell
type Stck = [ExpStck]
data ExpStck = E ExpPCF
  | COND (ExpPCF, ExpPCF)
  | deriving (Show)
```

In order to implement the stack machine function and to represent the new terms associated with it, a data structure was created (Code 3.5), in which the ExpStck is the structure that will hold the elements of the stack machine $S_p$.

One can see the complete implementation in the Appendix Code A.3.

The implementation of this stack machine will prove useful once we start to compare results between the PCF language and System $L_{rec}$, which we will describe in the next chapter.
Chapter 4

System $\mathcal{L}_{\text{rec}}$

We will now introduce System $\mathcal{L}_{\text{rec}}$, a syntactically linear $\lambda$-calculus extended with numbers, pairs and an unbounded recursor that preserves the syntactic linearity of the calculus, introduced in [Alves 11].

4.1 Syntax

As an extension of $\lambda$-calculus, the $\mathcal{L}_{\text{rec}}$ system also uses the notion of substitution to model the definition and application of functions. The representations of the application and the abstraction are also the same.

**Definition 4.1.1 ($\mathcal{L}_{\text{rec}}$).** Let $\mathcal{V}$ be an infinite set of variables. The set of $\mathcal{L}_{\text{rec}}$ terms $\Lambda_{\mathcal{R}}$, is built in the following way:

1. **(Zero)** $0 \Rightarrow 0 \in \Lambda_{\mathcal{R}}$
2. **(Variables)** $x \in \mathcal{V} \Rightarrow x \in \Lambda_{\mathcal{R}}$
3. **(Application)** $M, N \in \Lambda_{\mathcal{R}} \Rightarrow (MN) \in \Lambda_{\mathcal{R}}$, if $\text{FV}(M) \cap \text{FV}(N) = \emptyset$
4. **(Abstraction)** $x \in \mathcal{V}, M \in \Lambda_{\mathcal{R}} \Rightarrow (\lambda x. M) \in \Lambda_{\mathcal{R}}$, if $x \in \text{FV}(M)$
5. **(Successor)** $M \in \Lambda_{\mathcal{R}} \Rightarrow (S \ M) \in \Lambda_{\mathcal{R}}$
6. **(Pairs)** $M, N \in \Lambda_{\mathcal{R}} \Rightarrow ((M, N)) \in \Lambda_{\mathcal{R}}$, if $\text{FV}(M) \cap \text{FV}(N) = \emptyset$
7. **(Let)** $x, y \in \mathcal{V}, M, N \in \Lambda_{\mathcal{R}} \Rightarrow (\text{let } (x, y) = M \text{ in } N) \in \Lambda_{\mathcal{R}}$, if $x, y \in \text{FV}(N), x \neq y, \text{FV}(M) \cap \text{FV}(N) = \emptyset$
8. **(Rec)** $M_1, M_2, M_3, M_4 \in \Lambda_{\mathcal{R}} \Rightarrow (\text{rec } M_1 \ M_2 \ M_3 \ M_4) \in \Lambda_{\mathcal{R}}$, if $\text{FV}(M_i) \cap \text{FV}(M_j) = \emptyset$, for $i \neq j$

Once again, we consider the application left associative and the abstraction right associative.
4.1. Variables and Substitution

We will now formalize the concepts of free and bound variables of a term for System $\mathcal{L}_{\text{rec}}$.

Definition 4.1.2. (Free Variables) Let $M \in \Lambda_{\mathcal{R}}$, the set $FV(M)$ of free variables of $M$ is inductively defined as follows:

\[
\begin{align*}
FV(0) &= \emptyset \\
FV(x) &= \{x\} \\
FV(MN) &= FV(M) \cup FV(N) \\
FV(\lambda x. M) &= FV(M) \setminus \{x\} \\
FV(S M) &= FV(M) \\
FV((M,N)) &= FV(M) \cup FV(N) \\
FV(\langle x, y \rangle = M \text{ in } N) &= FV(M) \cup (FV(U) \setminus \{x, y\}) \\
FV(\text{let } \langle x, y \rangle = M \text{ in } N) &= FV(T) \cup FV(U) \cup FV(V) \cup FV(W)
\end{align*}
\]

Definition 4.1.3. (Bound Variables) Let $M \in \Lambda_{\mathcal{R}}$, the set $BV(M)$ of bound variables of $M$ is inductively defined as follows:

\[
\begin{align*}
BV(0) &= \emptyset \\
BV(x) &= \emptyset \\
BV(MN) &= BV(M) \cup BV(N) \\
BV(\lambda x. M) &= BV(M) \cup \{x\} \\
BV(S M) &= \emptyset \\
BV((M,N)) &= BV(M) \cup BV(N) \\
BV(\langle x, y \rangle = M \text{ in } N) &= BV(M) \cup (BV(U) \cup \{x, y\}) \\
BV(\text{let } \langle x, y \rangle = M \text{ in } N) &= BV(T) \cup BV(U) \cup BV(V) \cup BV(W)
\end{align*}
\]

Note that regarding the names of free and bound variables, this system, similar to the $\lambda$-calculus in Chapter 2, assumes the Barendregt’s convention.

Definition 4.1.4. ($\mathcal{L}_{\text{rec}}$ Substitution) The result of substituting the free occurrences of $x$ by $L$ in $M$ (denoted by $M[L/x]$) is defined as:

\[
FV(0) = \emptyset \\
FV(x) = \{x\} \\
FV(MN) = FV(M) \cup FV(N) \\
FV(\lambda x. M) = FV(M) \setminus \{x\} \\
FV(S M) = FV(M) \\
FV((M,N)) = FV(M) \cup FV(N) \\
FV(\langle x, y \rangle = M \text{ in } N) = FV(M) \cup (FV(U) \setminus \{x, y\}) \\
FV(\text{let } \langle x, y \rangle = M \text{ in } N) = FV(T) \cup FV(U) \cup FV(V) \cup FV(W)
\]
4.1. SYNTAX

\[ 0[L/y] \equiv 0 \]

\[ x[L/y] \equiv \begin{cases} L & \text{if } x \equiv y \\ x & \text{otherwise} \end{cases} \]

\[ (M N)[L/y] \equiv (M[L/y]) \ (N[L/y]) \]

\[ (\lambda x.M)[L/y] \equiv \begin{cases} (\lambda x.M) & \text{if } x \equiv y \\ \lambda x.(M[L/y]) & \text{otherwise} \end{cases} \]

\[ (\langle M, N \rangle)[L/y] \equiv \langle M[L/y], N[L/y] \rangle \]

\[ (S M)[L/y] \equiv S \ (M[L/y]) \]

\[ (\text{let } \langle x, y \rangle = M \text{ in } N)[L/z] \equiv \begin{cases} \text{let } \langle x, y \rangle = M \text{ in } N & \text{if } x \equiv z \lor y \equiv z \\ \text{let } \langle x, y \rangle = M[L/y] \text{ in } N[L/y] & \text{otherwise} \end{cases} \]

\[ (\text{rec } T U V W)[L/y] \equiv \text{rec } (T[L/y]) \ (U[L/y]) \ (V[L/y]) \ (W[L/y]) \]

Since it is a linear language, the substitution could be done taking into consideration the rules for free variables which could restrict the substitution to one branch of the term.

The reduction rules for this system consist of a restriction of the \( \beta \)-reduction from \( \lambda \)-calculus (Section 2.1.3) for closed terms, a rule for \text{Let} and two rules for the recursor as shown in the following definition.

**Definition 4.1.5 (\( L_{rec} \) Reduction).** The following contraction rules define the notion of reduction on System \( L_{rec} \):

\[ \beta : \quad (\lambda x.M)N \rightarrow M[N/x], \quad \text{if } FV(N) = \emptyset \]

\[ \text{Let} : \quad (\text{let } \langle x, y \rangle = (M, N) \text{ in } T) \rightarrow (T[M/x])[N/y], \quad \text{if } FV(M) = FV(N) = \emptyset \]

\[ \text{Rec} : \quad (\text{rec } \langle \emptyset, T \rangle U V W) \rightarrow U, \quad \text{if } FV(T) = FV(V) = FV(W) = \emptyset \]

\[ \text{Rec} : \quad (\text{rec } \langle S T_1, T_2 \rangle U V W) \rightarrow V(\text{rec } (W \langle T_1, T_2 \rangle) U V W), \quad \text{if } FV(V) = FV(W) = \emptyset \]

The linearity is preserved by reduction since the \text{Rec} rules are only triggered when the conditions for free variables hold. These rules pattern-match on a pair of numbers whereas the usual bounded recursor works on a single number, because we are representing both bounded and unbounded recursion with the same operator. The pair

Operational Semantics for Linear Languages
\langle M, N \rangle \) for this recursor represents the value being tested \( M = f(N) \) and the value \( N \) that produced \( M \). In this way, we can preserve the last iteration number more efficiently without interfering with the computation of the function.

The last parameter in the recursor is used to compute the next pair of numbers, so we can program unbounded and bounded recursion.

We will now give some examples, that later on will prove useful in the compilation from \textsc{pcf} to \textsc{lrec}.

**Example 4.1.1 (Projections and duplication of natural numbers).** Defining projections of pairs \( \langle a, b \rangle \) of natural numbers, is easy using recursion:

\[
pr_1 = \lambda x.\text{let} \langle a, b \rangle = x \text{ in } (\text{rec} \langle b, 0 \rangle a (\lambda x.x) (\lambda x.x))
\]

\[
pr_2 = \lambda x.\text{let} \langle a, b \rangle = x \text{ in } (\text{rec} \langle a, 0 \rangle b (\lambda x.x) (\lambda x.x))
\]

And the function for copying numbers:

\[
C = \lambda x.\text{rec} \langle x, 0 \rangle \langle 0, 0 \rangle (\lambda x.\text{let} x = \langle a, b \rangle \text{ in } (\text{S} a, \text{S} b)) (\lambda x.x)
\]

We also define some arithmetic functions, which will be used later on in the compilation.

**Example 4.1.2 (Arithmetic Functions).** The arithmetic functions can be encoded in \textsc{lrec} in the following way:

- \( \text{add} = \lambda mn.\text{rec} \langle m, 0 \rangle n (\lambda x.\text{S} x) (\lambda x.x) \);

- \( \text{mult} = \lambda mn.\text{rec} \langle m, 0 \rangle 0 (\text{add} n) \);

- \( \text{pred} = \lambda n.pr_1 (\text{rec} \langle n, 0 \rangle \langle 0, 0 \rangle F (\lambda x.x)) \)
  where \( F = \lambda x.\text{let} t, 0 = C(pr_2 x) \text{ in } \langle t, \text{S} u \rangle \);

- \( \text{iszero} = \lambda n.pr_1 (\text{rec} \langle n, 0 \rangle \langle 0, 0 \rangle (\lambda x.C(pr_2 x)) (\lambda x.x)) \)

These functions will be very useful later on in this thesis in order to compare results between \textsc{pcf} and \textsc{lrec}.

The correctness of these encodings can be easily proved by induction.
4.1.2 Types for System $\mathcal{L}_{rec}$

We will start by defining the linear types of System $\mathcal{L}_{rec}$.

**Definition 4.1.6.** Let $\mathcal{T}_R$ be the set of $\mathcal{L}_{rec}$ types defined inductively in the following way:

\[
\begin{align*}
\text{nat} & \in \mathcal{T}_R \\
\tau_0, \tau_1 & \in \mathcal{T}_R \Rightarrow (\tau_0 \rightarrow \tau_1) \in \mathcal{T}_R \\
\tau_0, \tau_1 & \in \mathcal{T}_R \Rightarrow (\tau_0 \otimes \tau_1) \in \mathcal{T}_R
\end{align*}
\]

Where $\text{nat}$ is a type for natural numbers.

**Definition 4.1.7.** Let $\Gamma$ be a type environment for System $\mathcal{L}_{rec}$. We say that $M$ has type $\tau$ given a basis $\Gamma$, and write:

$$
\Gamma \vdash_{\mathcal{L}} M : \tau,
$$

if $\Gamma \vdash_{\mathcal{L}} M : \tau$ can be obtained from the derivation rules shown in Table 4.1.

To denote the set of variables that occur in $\Gamma$, we write $\text{dom}(\Gamma)$, the same as for the $\lambda$-calculus.

**Theorem 4.1.1** (Properties of reductions in $\mathcal{L}_{rec}$). The properties of reduction in $\mathcal{L}_{rec}$ are:

1. If $\Gamma \vdash_{\mathcal{L}} T : \tau$ then $\text{dom}(\Gamma) = \text{FV}(\tau)$.

2. **Subject Reduction:** Reductions preserves types.

3. **Church-Rosser:** System $\mathcal{L}_{rec}$ is confluent.

4. **Adequacy:** If $\vdash_{\mathcal{L}} T : \tau$ in System $\mathcal{L}_{rec}$ and $T$ is a normal form, then there are $\mathcal{L}_{rec}$ terms $U, M$ such that:

\[
\begin{align*}
\tau = \text{nat} & \quad \Rightarrow \quad T = S(S \ldots (S 0)) \\
\tau = \tau_0 \otimes \tau_1 & \quad \Rightarrow \quad T = \langle U, M \rangle \\
\tau = \tau_0 \rightarrow \tau_1 & \quad \Rightarrow \quad T = \lambda x. M
\end{align*}
\]

5. **System $\mathcal{L}_{rec}$ is not strongly normalizing,** even for typable terms.
Axiom:

\[ \{x : \tau_0\} \vdash \ell x : \tau_0 \quad (Axiom) \]

**Logical Rules:**

\[ \Gamma, \{x : \tau_0\} \vdash T : \tau_1 \quad \Gamma \vdash \ell x : \tau_0 \rightarrow \tau_1 \quad \Delta \vdash \ell U : \tau_0 \quad \Gamma \cup \Delta \vdash \ell T U : \tau_1 \quad (\rightarrow \text{Intro}) \]

\[ \Gamma \vdash \ell x : \tau_0 \rightarrow \tau_1 \quad \Delta \vdash \ell U : \tau_0 \quad \Gamma \cup \Delta \vdash \ell T U : \tau_1 \quad (\rightarrow \text{Elim}) \]

\[ \Gamma \vdash \ell T : \tau_0 \quad \Delta \vdash \ell U : \tau_1 \quad \Gamma \cup \Delta \vdash \ell \langle T, U \rangle : \tau_0 \otimes \tau_1 \quad \Gamma \vdash \ell \langle T, U \rangle : \tau_0 \otimes \tau_1 \quad \Delta_{x,y} \cup \{x : \tau_0, y : \tau_1\} \vdash \ell U : \tau_2 \quad \Delta \vdash \ell \text{let } \langle x, y \rangle = T \text{ in } U : \tau_2 \quad (\otimes \text{Intro}) \]

\[ \Gamma \vdash \ell T : \tau_0 \quad \Delta \vdash \ell U : \tau_1 \quad \Gamma \cup \Delta \vdash \ell \langle T, U \rangle : \tau_0 \otimes \tau_1 \quad \Gamma \vdash \ell \langle T, U \rangle : \tau_0 \otimes \tau_1 \quad \Delta_{x,y} \cup \{x : \tau_0, y : \tau_1\} \vdash \ell U : \tau_2 \quad (\otimes \text{Elim}) \]

**Numbers:**

\[ \vdash \ell 0 : \text{nat} \quad \Gamma \vdash \ell N : \text{nat} \quad \Gamma \vdash \ell S N : \text{nat} \quad (\text{Zero}) \quad (\text{Succ}) \]

\[ \Gamma \vdash \ell T : \text{nat} \otimes \text{nat} \quad \Theta \vdash \ell U : \tau_0 \quad \Delta \vdash \ell V : \tau_0 \rightarrow \tau_0 \quad \Sigma \vdash \ell W : \text{nat} \otimes \text{nat} \rightarrow \text{nat} \otimes \text{nat} \quad \Gamma \cup \Theta \cup \Delta \cup \Sigma \vdash \ell \text{rec } T U V W : \tau_0 \quad (\text{Rec}) \]

Table 4.1: Type System for System \( \mathcal{L}_{rec} \)

The proofs for this and other properties on \( \mathcal{L}_{rec} \) can be found in [Alves 11].

In the linear \( \lambda \)-calculus terms are consumed by reduction, even though we are not able to discard arguments of functions. In order to erase, System \( \mathcal{L}_{rec} \) applies the technique of Solvability [Barendregt 84], generalizing the encoding of projections given in Example 4.1.1.

To erase a term \( T \) of type \( \tau \), \( \mathcal{L}_{rec} \) uses a function \( \text{Erase}(\mathcal{E}(T, \tau)) \) and a function \( \text{Make}(\mathcal{M}(\tau)) \) to build a term of a specific type (\( \mathcal{E} \) and \( \mathcal{M} \) are mutually recursive).

**Definition 4.1.8 (Erasing).** If \( \Gamma \vdash \ell T : \tau \), then \( \mathcal{E}(T, \tau) \) is defined as follows:

\[ \mathcal{E}(T, \text{nat}) = \text{rec } \langle T, 0 \rangle (\lambda x.x) (\lambda x.x) (\lambda x.x) \]

\[ \mathcal{E}(T, \tau_0 \otimes \tau_1) = \text{let } \langle x, y \rangle = T \text{ in } (\mathcal{E}(x, \tau_0) \mathcal{E}(y, \tau_1)) \]

\[ \mathcal{E}(T, \tau_0 \rightarrow \tau_1) = \mathcal{E}(T \mathcal{M}(\tau_0), \tau_1) \]

and

\[ \mathcal{M}(\text{nat}) = 0 \]

\[ \mathcal{M}(\tau_0 \otimes \tau_1) = \langle \mathcal{M}(\tau_0), \mathcal{M}(\tau_1) \rangle \]

\[ \mathcal{M}(\tau_0 \rightarrow \tau_1) = \lambda x.\mathcal{E}(x, \tau_0)\mathcal{M}(\tau_1) \]

Since we are in a non-normalizing calculus, not every term can be erased in this way. In order to copy closed terms, the System \( \mathcal{L}_{rec} \) as also defined a duplication function:

---

**Operational Semantics for Linear Languages**
4.1. SYNTAX

<table>
<thead>
<tr>
<th>V is a value</th>
<th>S ↓ λx.U</th>
<th>U[T/x] ↓ V</th>
<th>T ↓ ⟨T₁, T₂⟩ (λxy.U)T₁T₂ ↓ V</th>
<th>Let ⟨x, y⟩ = T in U ↓ V</th>
</tr>
</thead>
<tbody>
<tr>
<td>V ↓ V</td>
<td>ST ↓ V</td>
<td>App</td>
<td></td>
<td>Rec₁</td>
</tr>
<tr>
<td>T ↓ ⟨T₁, T₂⟩</td>
<td>T₁ ↓ 0</td>
<td>U ↓ V</td>
<td></td>
<td>Rec₂</td>
</tr>
<tr>
<td>rec T U V W W ↓ V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: CBN evaluation for System $L_{rec}$

**Definition 4.1.9** (Duplication). $D^τ : τ → τ ⊗ τ$ is defined as:

$$\lambda x.\text{rec} \langle S(S 0), 0 \rangle \langle M(τ), M(τ) \rangle F(\lambda x.x)$$

where $F = (\lambda y.\text{let} \langle z, w \rangle = y \text{ in } E(z, τ) \langle w, x \rangle)$.

These definitions will prove useful in order to define the compilation from PCF to System $L_{rec}$.

Results concerning the properties of these functions were proved in [Alves 11].

### 4.1.3 Evaluation Strategies

We will define two evaluation strategies for the $L_{rec}$ system, call-by-name and call-by-value, and derive a simple stack machine.

**Call-by-Name** The call-by-name evaluation function for the $L_{rec}$ system is presented in Table 4.2. When a closed term $T$ evaluates to a value, in System $L_{rec}$ we write $T \downarrow V$. In this system, a value, or a weak normal form, consists of the following terms: $0, ST, \lambda x.T$ and $\langle S, T \rangle$. In System $L_{rec}$ the symbol $S$ is used as a constructor of natural numbers, and therefore is not evaluated, unlike PCF that used the successor as a function. Also note that no closedness condition is needed in the evaluation rules, since we are only considering closed terms.

Since in System $L_{rec}$ we write the Let rule using application, the two strategies only differ in the application rule.

**Call-by-Value** Similar to the $\lambda$-calculus, the call-by-value evaluation changes only in the application rule:

$$S \downarrow \lambda x.U \quad T \downarrow V' \quad U[V'/x] \downarrow V \quad ST \downarrow V$$
Since the $\text{Rec}$ and $\text{Let}$ rules rely on the application rule, there is no need to change them for the call-by-value strategy.

Unlike CBN, the CBV strategy does not always reach a value, even if a close term has one.

### 4.2 Implementation

Similar to the PCF language, the diagram of System $\mathcal{L}_{\text{rec}}$ implementation in Figure 4.1, starts by an analysis of the language, followed by interpreters call-by-name, call-by-value and an abstract machine, that follows a call-by-name evaluation. The final goal of this implementation is for the expression to reach its normal form, if one exists.

![Figure 4.1: System $\mathcal{L}_{\text{rec}}$ Implementation](image)

Following the same implementation of the PCF language, the System $\mathcal{L}_{\text{rec}}$ implementation started with a parser, also written in Happy which converts the input, $\mathcal{L}_{\text{rec}}$ terms, into a data structure, Code 4.1.

**Code 4.1: Data Structure for the $\mathcal{L}_{\text{rec}}$ System**

```haskell
data Exp =
    Rec Exp Exp Exp Exp
  | Pair Exp Exp
  | Let (String, String) Exp Exp
  | Var String
  | Zero
  | Suc Exp
  | Lambda String Exp
  | App Exp Exp
  deriving (Show, Eq)
```

Operational Semantics for Linear Languages 52
4.2. IMPLEMENTATION

This structure was built according to the needs of the language:

- (Zero) represents the only numerical number in $\mathcal{L}_{rec}$, 0;
- (Suc Exp), (Var String), (Lambd String Exp), (Pair Exp Exp) represent the values of the system: successor, variables, abstraction and pairs, respectively;
- (Rec Exp Exp Exp Exp) represents the rec term;
- the (Let (String, String) Exp Exp) structure was created with a tuple of strings instead of a data structure pair, in order to make the implementation of the system in Haskell easier;
- (App Exp Exp) obviously represents the term application $MN$.

In the following subsections we will present the implementations of the call-by-name and call-by-value evaluations and a stack machine for the $\mathcal{L}_{rec}$ system, using this data structure.

4.2.1 Call-by-Name Interpreter

The call-by-name implementation, was done according to Table 4.2. We will discuss here some aspects of the implementation. One can see the complete code for that function in the Appendix (Code A.4).

```
Code 4.2: The call-by-name Evaluation for System $\mathcal{L}_{rec}$

```

```

eval_cbn (Suc a) | (value a) = (Suc a)
| otherwise = (Suc (eval_cbn a))
```

In Code 4.2 one can see that the evaluation of the successor term was done using a function called `value :: Exp -> Bool` which is used to determine if the expression within the successor is a value or not. Although in the call-by-name evaluation rules of System $\mathcal{L}_{rec}$ the evaluation within the successor was not described, when implementing we felt the need to evaluate under successor since we realized that the evaluation of the terms might not end in a normal form if we did not evaluate inside the successor. As for the other values, when `eval_cbn` receives them, it returns them as they are, since those expressions are already in their normal form.
There are two rules for the application (Code 4.3):

- **Application of a λ-abstractor**: where the substitution function is used (Definition A.7, Appendix Code A.7) to evaluate following the rules in Table 4.2;
- **Application of any other two terms**: where we use the auxiliary function (Code 4.4) in order to evaluate the application without causing the same problem described in Section 3.2.1.

In the let expression, if we do not have a pair (Code 4.5: line 3), we first evaluate that term in order to reach the pair, and then make the correct let evaluation (Code 4.5: line 1) according to Table 4.2.

Finally, the rec expression has several different cases:

- **The pair starts with the value**:
  - *Zero* (Code 4.6: line 1): then we return the term \( U \) (Table 4.2: \( \text{Rec}_1 \));
  - *Suc t* (Code 4.6: line 2): the successor rule applies, so we use the recursion rule (Table 4.2: \( \text{Rec}_2 \)).

- **The pair is not reduced to a value**: so we call the function again in order to evaluate the first term of the pair (Table 4.2: \( \text{Rec}_1 \));

- **There is no pair**: we call the function recursively in order to find the pair so we can apply the evaluation (Table 4.2).
If none of the above rules applies to the expression, then the expression is already in normal form, and the evaluation terminates.

### 4.2.2 Call-by-Value Interpreter

As it was mentioned in Section 5.2, the call-by-value function changes only in the application rule. Therefore, the function is identical to the one described above, except for the application rule (Code 4.7) where we evaluate the second term before calling the substitution function.

#### Code 4.7: call-by-value Evaluation Function

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>eval_cbv (App (Lambda x u) t) = eval_cbv (subst u x (eval_cbv t))</code></td>
</tr>
<tr>
<td>2</td>
<td><code>eval_cbv (App s t) = cbv_aux (App (eval_cbv s) t)</code></td>
</tr>
</tbody>
</table>

In this function we also used an auxiliary function (Code A.6: \texttt{cbv\_aux :: Exp -> Exp}), once again, to prevent continually trying to evaluate something that is already in its normal form.

### 4.2.3 Stack Machine

We will start by defining the elements of the stack machine.

**Definition 4.2.1 (System \( \mathcal{L}_{\text{rec}} \) Stack Machine Elements).** Let \( S_{\mathcal{R}} \) be the set of elements of the stack machine, which is an extension of the System \( \mathcal{L}_{\text{rec}} \) terms, \( \Lambda_{\mathcal{R}} \), then:
4.2. IMPLEMENTATION

(app) \((MN, S) \Rightarrow (M, N : S)\)
(abs) \(((\lambda x. M), N : S) \Rightarrow (M[N/x], S)\)
(let) \((\text{let}(x, y) = N \text{ in } M, S) \Rightarrow (N, \text{LET}(x, y, M) : S)\)
(pair1) \((\langle N_1, N_2 \rangle, \text{LET}(x, y, M) : S) \Rightarrow (M[N_1/x][N_2/y], S)\)
(rec) \((\text{rec } N U V W, S) \Rightarrow (N, \text{REC}(U, V, W) : S)\)
(pair2) \((\langle N_1, N_2 \rangle, \text{REC}(U, V, W) : S) \Rightarrow (M[N_1/x][N_2/y], S)\)
(rec1) \((\text{rec } (S N), U V W : S) \Rightarrow (V, (\text{rec } W (N, T)) U V W : S)\)
(succ1) \((\text{SUCC} : S) \Rightarrow (N, \text{SUCC} : \text{SUCC} : S)\)
(succ2) \((S N, S) \Rightarrow (N, \text{SUCC} : S)\)
(succ3) \((S N, S) \Rightarrow (S N, S)\)

Table 4.3: \(L_{\text{rec}}\) Stack Machine

<table>
<thead>
<tr>
<th>SUCC \in S_R</th>
</tr>
</thead>
<tbody>
<tr>
<td>M \in \Lambda_R \Rightarrow M \in S_R</td>
</tr>
<tr>
<td>x, y \in V \text{ and } T \in \Lambda_R \Rightarrow \text{LET}(x, y, T) \in S_R</td>
</tr>
<tr>
<td>U, V, W \in \Lambda_R \Rightarrow \text{REC}(U, V, W) \in S_R</td>
</tr>
<tr>
<td>N, U, V, W \in \Lambda_R \Rightarrow \text{REC}'(N, U, V, W) \in S_R</td>
</tr>
</tbody>
</table>

As it can be seen in Table 4.3, the stack machine for System \(L_{\text{rec}}\) is based on the call-by-name strategy.

**Definition 4.2.2 (\(L_{\text{rec}}\) Stack Configuration).** The \(L_{\text{rec}}\) Stack Machine is composed by a states of the form \((M, S)\), such that:

- \(M \in \Lambda_R\)
- \(S\) is a stack of extended terms, \(S_R\)

We consider the following special cases for states:

- Initial State: \((M, \epsilon)\)
- Final State: \((V, \epsilon)\), where \(V\) is a \(L_{\text{rec}}\) Value

The basic principle of the machine is the same as for \(\text{PCF}\): find the next redex, using the stack \(S\) to store future computations.

Although the Krivine machine (Table 2.3.2) included an environment, in this case none is needed since, when the \(L_{\text{rec}}\) Stack Machine finds a binding of a variable to a term, replaces the unique occurrence of that variable, i.e., it replaces the occurrence...
of the variable by the term.
The \( \mathcal{L}_{\text{rec}} \) Stack Machine ends when a normal form is reached, or, in case it can't be reached, with the last reduction possible.

**Example 4.2.1.** To better understand the execution of the machine, we will present an example using the term \(((\lambda m. (\lambda n. \text{rec} \langle m, 0 \rangle n (\lambda x. S x) (\lambda x. x))) \ (S \ 0)) \ (S \ (S \ 0))\):
4.2. IMPLEMENTATION

\[ \Downarrow \text{succ2} \]
\[ (0, \text{SUCC} : \text{SUCC} : \text{SUCC} : \epsilon) \]
\[ \Downarrow \text{succ3} \]
\[ (S (S (S 0)), \epsilon) \]

The implementation of the stack machine was done according to Table 4.3, using the data structure shown below in Code 4.8, where we created the new terms that were described (Description 4.2.1): \text{LET}(x, y, t) (line 3); \text{REC}(u, v, w) (line 4) and \text{REC}'(n, u, v, w) (line 5); and also a term (\text{SUCC}) that will help in the stack evaluation of the successor, as in the call-by-name and call-by-value evaluations, where we had to evaluate inside the term.

The structure (\text{E Exp}) was created in order to include the System \text{L}_{rec} terms in the new extended set of the stack, \text{ExpStck}.

Code 4.8: Data Structure for the Stack

\begin{verbatim}
type Stck = [ExpStck]
data ExpStck = E Exp |
| LET (String, String, Exp)
| REC (Exp, Exp, Exp)
| REC' (Exp, Exp, Exp, Exp)
| SUCC
deriving (Show)
\end{verbatim}

The complete code for this implementation can be found in the Appendix (Code A.8), but some of it will be better explained next.

Code 4.9: Successor Evaluation in the Stack Machine

\begin{verbatim}
stck ((Suc n), SUCC:s) = if (value n) then stck (Suc (Suc n), s) else stck (n, SUCC:SUCC:s)
stck ((Suc n), s) = if (value n) then ((Suc n), s) else stck (n, SUCC:s)
stck (n, SUCC:s) = if (value n) then stck ((Suc n), s) else ((Suc n), s)
\end{verbatim}

We will start by explaining the need for the term \text{SUCC}. As one can see in Code 4.9, first we verify if the term inside the successor is a value or not. If it is not a value, then we add the new data structure \text{SUCC} to the stack, so we can first evaluate inside the successor, and then rebuild the successor term. If it is a value, then there is nothing to evaluate and the machine just needs to rebuild the successor term.

The rest of the stack implementation is straightforward, if one follows Table 4.3. Note that we use the substitution function (Definition 2.1.4, Code A.7), in the application
of a λ-term.
In the next chapter, we will compare the results from this stack machine with the PCF stack machine.
4.2. IMPLEMENTATION

Operational Semantics for Linear Languages
Chapter 5

Compiling

In this chapter we will show how the PCF language can be encoded in System $\mathcal{L}_{rec}$. We will start by defining the compilation from PCF to $\mathcal{L}_{rec}$, following with the implementation of the compilation and finally, we will compare evaluation results between both. We will also present a variant of System $\mathcal{L}_{rec}$ with built-in natural numbers, its implementation and compare its evaluation results with the previous ones.

5.1 Compiling

We will start by defining how the PCF types and environments can be translated into System $\mathcal{L}_{rec}$ types.

Definition 5.1.1. PCF types and environments are translated into System $\mathcal{L}_{rec}$ types using $\langle \cdot \rangle$:

\[
\begin{align*}
\langle \text{int} \rangle &= \text{nat} \\
\langle \text{bool} \rangle &= \text{nat} \\
\langle \tau_1 \rightarrow \tau_2 \rangle &= \langle \tau_1 \rangle \rightarrow \langle \tau_2 \rangle \\
\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle &= x_1 : \langle \tau_1 \rangle, \ldots, x_n : \langle \tau_n \rangle
\end{align*}
\]

Using the encoding for types and environments, we will now show how a PCF program can be encoded into System $\mathcal{L}_{rec}$. In the following abbreviations, the variables $x_1$ and $x_2$ are assumed fresh, and $[x]T$ will be defined bellow.

\[
C_{x_1:x_2}^{\tau_1,\tau_2} T = \text{let } \langle x_1, x_2 \rangle = D^\tau x \text{ in } T
\]
\[
\begin{align*}
\langle tt \rangle &= 0 \\
\langle ff \rangle &= S 0 \\
\langle n \rangle &= S^n 0 \\
\langle \text{succ} \rangle &= \lambda n. \text{rec} (n, 0) (S 0) (\lambda x. S x) (\lambda x. x) \\
\langle \text{pred} \rangle &= \lambda n. \text{pr} (\text{rec} (n, 0) (0, 0) (\lambda x. \text{let} \langle t, u \rangle = D^{\text{nat}} (pr_2 x) \text{ in} \langle t, S u \rangle) (\lambda x. x)) \\
\langle \text{iszero} \rangle &= \lambda n. \text{pr} (\text{rec} (n, 0) (0, S 0) (\lambda x. D^{\text{nat}} (pr_2 x)) (\lambda x. x)) \\
\langle \text{Y} \rangle &= \lambda f. \text{rec} (S 0, 0) M (\langle \tau \rangle) f (\lambda x. \text{let} \langle y, z \rangle = x \text{ in} \langle S y, z \rangle) \\
\langle \text{cond} \rangle &= \lambda tuv. \text{rec} (t, 0) u (\lambda x. (\text{rec} (0, 0) (\lambda x. x) x (\lambda x. x)) v) (\lambda x. x) \\
\langle x \rangle &= x \\
\langle UV \rangle &= \langle U \rangle \langle V \rangle \\
\langle \lambda x^\tau. T \rangle &= \begin{cases} \\
\lambda x. [x^\tau] (T) & \text{if } x \in FV(T) \\
\lambda x. (\text{rec} (0, 0) (\lambda x. x) x (\lambda x. x)) (T) & \text{otherwise}
\end{cases}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Table 5.1: PCF compilation into $\mathcal{L}_{\text{rec}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_y^x T = ([x] T)[y/x]$</td>
</tr>
</tbody>
</table>

**Definition 5.1.2** (Linearization). Let $T$ be a PCF term, with $FV(T) = \{x_1, \ldots, x_n\}$ and $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash T : \tau$. The compilation into System $\mathcal{L}_{\text{rec}}$ is defined as: $[x_1^{\tau_1}] \ldots [x_n^{\tau_n}] (T)$, where $(\cdot)$ is defined in Table 5.1, and for a term $T$ and a variable $x$, such that $x \in FV(T)$, $[x] T$ is inductively defined in the following way:

\[
\begin{align*}
[x] (S \ U) &= S([x] U) \\
[x] x &= x \\
[x] (\lambda y. U) &= \lambda y. [x] U \\
[x^\tau] (M \ U) &= \begin{cases} \\
C_{x^\tau}^{x_1, x_2} (A_x^{x_1} M) (A_x^{x_2} U) & x \in FV(M) \cap FV(U) \\
([x] M) U & x \notin FV(U) \\
M ([x] U) & x \notin FV(M)
\end{cases}
\end{align*}
\]

In this definition, $[x] T$, is not defined for the entire syntax of System $\mathcal{L}_{\text{rec}}$, because although others syntactic constructors may appear in $T$, these are the result of $(\cdot)$ and are therefore closed terms were $x$ does not occur free. The complete encoding can be seen in Table 5.1.

Also note that $\langle \text{succ} \rangle$ is not encoded as $\lambda x. S x$, since System $\mathcal{L}_{\text{rec}}$ originally does not evaluate under abstractions or $S$. Although when the implementation was completed, there was the necessity to evaluate inside the successor, the encoding was preserved since $\text{cond} \_ (\text{succ} (Y, (\lambda x. x)))$ $P$ $Q$ is $\langle \text{cond} \_ \rangle (\langle \text{succ} \rangle (\langle Y \rangle (\lambda x. x)))$ $\langle P \rangle$ $\langle Q \rangle$, so if we encoded $\langle \text{succ} \rangle$ into $\lambda x. S x$ we would have $\langle Q \rangle$ which is not correct.

In the second case of the abstraction, we use a recursor over zero which returns
the identity function discarding the argument. The variable $x$ is used directly as a parameter of the function, since when we implemented the encoding we noted that, we only used integers types, \texttt{int} and \texttt{bool}, the use of the erasing function was unjustified. We will now give an example of a compilation from \textsf{PCF} to System $\mathcal{L}_{\text{rec}}$.

**Example 5.1.1.** Using Table 5.1 the \textsf{PCF} term, $(\text{cond}_{\text{int}} (\text{iszer} \ m) \ \overline{T} \ (\text{pred} \ m))$, can be encoded into System $\mathcal{L}_{\text{rec}}$ as:

$$
((\lambda t u v. \ \text{rec} \left< t, u \right> v) (\lambda x. x) (\lambda x. (\text{rec} \left< 0, 0 \right> (\lambda x. x) \ x (\lambda x. x))) v) (\lambda x. x)) \\
(\lambda n. \ \text{pr}_1 (\text{rec} \left< n, 0 \right> \left< 0, S \ 0 \right> (\lambda x. D^{\text{nat}}(\text{pr}_2 x)) (\lambda x. x) \ m) (S \ 0)) \\
(\lambda n. \ \text{pr}_1 (\text{rec} \left< n, 0 \right> \left< 0, 0 \right> (\lambda x. \text{let} \left< t, u \right> = D^{\text{nat}}(\text{pr}_2 x) \ \text{in} \ (t, S \ u) (\lambda x. x)) \ m)
$$

The implementation starts by changing the data structure of \textsf{PCF} terms so that its names in Haskell would not be the same as the $\mathcal{L}_{\text{rec}}$ terms. In that sense, the new data structure for \textsf{PCF} expressions can be seen in Code 5.1.

**Code 5.1: \textsf{PCF} Terms**

```haskell
data ExpPCF = VarPCF String | NatPCF Int | TT | FF |
  | IsZ | Succ | Pred | Cond |
  | PtoF Type |
  | LambdPCF (String , Type) ExpPCF |
  | AppPCF ExpPCF ExpPCF |
  deriving (Show , Eq)
```

The complete code for the encoding function can be seen in Appendix Code A.9, where the function \texttt{linearFV} (Code A.10) was done according to Definition 5.1.2, where Type is a new Haskell type created to encode \textsf{PCF} types into $\mathcal{L}_{\text{rec}}$ types. Note that the functions \texttt{erase} and \texttt{make} were also implemented in Appendix Code A.11 and A.12; and the \texttt{copy} function is the one used in Definition 5.1.2 for copying the variables.

## 5.2 Comparing Results

In order to compare the results between the two languages we decided to implement an abstract time measure from [Lago 06]: with the purpose of giving an cost evaluator for the $\lambda$-calculus where elementary reductions steps are counted proportionally to the number of corresponding steps in a Turing machine.

We will start by defining how the cost for the abstract time measure will be calculated, where we denote the size of a $\lambda$-term $M$ as $|M|$. 

---

\[\text{Operational Semantics for Linear Languages}\]
Definition 5.2.1 (Abstract Time Measure). An abstract time measure can be defined as:

\[
\begin{align*}
M & \rightarrow^e N & M \rightarrow N & n = \max\{1, |N| - |M|\} & M \rightarrow^\alpha N & N \rightarrow^\beta L \\
\end{align*}
\]

Were \(\alpha \cdot \beta\) denotes the concatenation of \(\alpha, \beta \in \mathbb{N}^*\), and given \(\alpha = n_1 \cdot \ldots \cdot n_m \in \mathbb{N}^*\), we define \(\|\alpha\| = \sum_{i=1}^{m} n_i\).

To implement this definition, we started by creating a function to calculate the size of a \(L_{\text{rec}}\) expression (Code 5.2), and similarly for the PCF language in Appendix Code A.13.

Code 5.2: \(L_{\text{rec}}\) expression size

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>esizeLRec :: Exp -&gt; Int</td>
</tr>
<tr>
<td>2</td>
<td>esizeLRec (Zero) = 1</td>
</tr>
<tr>
<td>3</td>
<td>esizeLRec (Var a) = 1</td>
</tr>
<tr>
<td>4</td>
<td>esizeLRec (Suc a) = 1+(esizeLRec a)</td>
</tr>
<tr>
<td>5</td>
<td>esizeLRec (Pair n m) = (esizeLRec m)+(esizeLRec n)</td>
</tr>
<tr>
<td>6</td>
<td>esizeLRec (Lambda x t) = 1+(esizeLRec t)</td>
</tr>
<tr>
<td>7</td>
<td>esizeLRec (App m n) = (esizeLRec m)+(esizeLRec n)</td>
</tr>
<tr>
<td>8</td>
<td>esizeLRec (Let (x, y) p u) = 2+(esizeLRec p)+(esizeLRec u)</td>
</tr>
<tr>
<td>9</td>
<td>esizeLRec (Rec t u v w) =</td>
</tr>
<tr>
<td>10</td>
<td>(esizeLRec t)+(esizeLRec u)+(esizeLRec v)+(esizeLRec w)</td>
</tr>
</tbody>
</table>

After the function for calculating the expression size, we changed the call-by-name evaluation function, so that instead of returning just an evaluated expression, it returns a tuple with the evaluated expression, the number of reductions that took to evaluate it and the cost defined in Definition 5.2.1. The complete code for the new evaluation function can be seen in the Appendix (Code A.14 for System \(L_{\text{rec}}\) and Code A.15 for the PCF language).

The initial arguments of the evaluation function are the expression to be evaluated and the counter, starting as zero. Every time a reduction is made, the function adds one number to the reduction counter, calculates the cost and adds it to the previous cost count, in order to calculate \(\alpha \cdot \beta\) (Definition 5.2.1).

Code 5.3: \(L_{\text{rec}}\) evaluation with Abstract Time Measure

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>eval_cbn (App s t) n =</td>
</tr>
<tr>
<td>2</td>
<td>let (e1, nr1, it) = (eval_cbn s n)</td>
</tr>
<tr>
<td>2</td>
<td>(e2, nr2, it2) = (cbn_aux(App e1 t) nr1)</td>
</tr>
<tr>
<td>4</td>
<td>in (e2, nr2, it2+it)</td>
</tr>
<tr>
<td>4</td>
<td>eval_cbn (Let (a, b) p u) n =</td>
</tr>
</tbody>
</table>
As one can see in Code 5.3, in some cases of the System $\mathcal{L}_{rec}$ evaluation function, the number of reductions is not updated and the cost is not calculated, since a reduction is not actually made. Although we still make the final sum of the costs and number of reductions from previous reductions.

Note that when the expression is a value, the function also does not add on the number of reductions or calculate the cost.

Code 5.4: PCF language evaluation with Abstract Time Measure

In the evaluation function for the PCF language, although the counter for reductions is increased, some cases do not calculate the cost even when a reduction is made (Code 5.4), that happens due to the fact that the difference between the expressions $|N| - |M|$ would always be negative, so the maximum between one and the difference would always be one, therefore, we avoid having to calculate the cost by replacing it with the number one.

Also note that this function has special cases, as it was for System $\mathcal{L}_{rec}$ where no new counter or cost is calculated (Code 5.5).
5.2. COMPARING RESULTS

<table>
<thead>
<tr>
<th>Function</th>
<th>PCF</th>
<th>$\mathcal{L}_{rec}$</th>
<th>PCF to $\mathcal{L}_{rec}$</th>
<th>Number of reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td>add 2 3</td>
<td>56</td>
<td>34</td>
<td>4975</td>
<td>23</td>
</tr>
<tr>
<td>mult 2 3</td>
<td>247</td>
<td>108</td>
<td>5091</td>
<td>87</td>
</tr>
<tr>
<td>fib 4</td>
<td>1255</td>
<td>402</td>
<td>289499</td>
<td>197</td>
</tr>
<tr>
<td>fact 4</td>
<td>22165</td>
<td>3530</td>
<td>5464324</td>
<td>4546</td>
</tr>
</tbody>
</table>

Table 5.2: Results Comparison

After this implementation some tests were made to the program in PCF, System $\mathcal{L}_{rec}$ and to the one with the encoding from PCF to $\mathcal{L}_{rec}$. The results can be seen in Table 5.2 were the functions called add, mult, fib and fact, are encodings of the addition, multiplication, fibonacci and factorial functions to PCF and System $\mathcal{L}_{rec}$, respectively. Although the results between PCF and System $\mathcal{L}_{rec}$ seem to be what was expected, the results with the encoding from PCF to $\mathcal{L}_{rec}$ could be better. Partially this could rely on the fact that $\mathcal{L}_{rec}$ does not operate on natural numbers but on successors over zero, therefore we decided to implement System $\mathcal{L}_{rec}$ with built-in natural numbers. We will call it $\mathcal{L}_{rec}^{\text{nat}}$.

5.2.1 System $\mathcal{L}_{rec}$ with built-in naturals

Adding naturals to System $\mathcal{L}_{rec}$ was done by adding in the parser a rule for the numbers, but in order to really try to simplify $\mathcal{L}_{rec}$ and make it more efficient we decided to implement the predecessor and the function iszero, as it exists in PCF. In that sense, the syntax of the $\mathcal{L}_{rec}^{\text{nat}}$ will have the variables, application, abstraction, pairs, rec and let from the System $\mathcal{L}_{rec}$, with the addition of naturals, such that, for the set of $\mathcal{L}_{rec}^{\text{nat}}$ terms, $\Lambda_{RN}$:

$$\bar{\pi} \in \mathbb{N} \Rightarrow \bar{\pi} \in \Lambda_{RN}$$
5.2. COMPARING RESULTS

\[
\begin{align*}
M \Rightarrow M' & \quad \text{suc } \pi \Rightarrow \overline{n + 1} \\
\text{suc } M \Rightarrow \text{suc } M' & \\
M \Rightarrow M' & \quad \text{pre } 0 \Rightarrow 0 \\
\text{pre } M \Rightarrow \text{pre } M' & \quad \text{pre } 0 + 1 \Rightarrow \pi
\end{align*}
\]

Table 5.3: $\mathcal{L}_{\text{rec}}^{\text{nat}}$ Call-by-Name Evaluation

- \((\text{app})\) \((MN, S) \Rightarrow (M, N:S)\)
- \((\text{abs})\) \((\lambda x.M), N:S \Rightarrow (M[N/x], S)\)
- \((\text{succ}1)\) \((\pi, \text{SUCC}:S) \Rightarrow (\overline{n + 1}, S)\)
- \((\text{succ}2)\) \((\text{suc }, M:S) \Rightarrow (M, \text{SUCC}:S)\)
- \((\text{pred}1)\) \((\overline{0}, \text{PRE}:S) \Rightarrow (\overline{0}, S)\)
- \((\text{pred}2)\) \((\overline{n + 1}, \text{PRE}:S) \Rightarrow (\overline{n}, S)\)
- \((\text{iszero}1)\) \((\overline{0}, \text{ISZERO}:S) \Rightarrow (\overline{0}, S)\)
- \((\text{iszero}2)\) \((\overline{n + 1}, \text{ISZERO}:S) \Rightarrow (\overline{0}, S)\)
- \((\text{iszero}3)\) \((\text{iszero}, M:S) \Rightarrow (M, \text{ISZERO}:S)\)
- \((\text{let})\) \((\text{let} < x, y >= N \text{ in } M, S) \Rightarrow (N, \text{LET}(x,y,M):S)\)
- \((\text{pair}1)\) \((< N_1, N_2 >, \text{LET}(x,y,M):S) \Rightarrow (M[N_1/x][N_2/y], S)\)
- \((\text{rec})\) \((\text{rec } N U V W, S) \Rightarrow (N, \text{REC}(U,V,W):S)\)
- \((\text{pair}2)\) \((< N_1, N_2 >, \text{REC}(U,V,W):S) \Rightarrow (N_1, \text{REC}^*(N_2,U,V,W):S)\)
- \((\text{zero})\) \((\overline{0}, \text{REC}^*(T,U,V,W):S) \Rightarrow (U, S)\)
- \((\text{rec}1)\) \((\overline{n + 1}, \text{REC}^*(T,U,V,W):S) \Rightarrow (V, (\text{rec}(W < \pi, T >) U V W):S)\)

Table 5.4: $\mathcal{L}_{\text{rec}}^{\text{nat}}$ Stack Machine

And also the addition of the functions \textbf{suc}, for the successor, \textbf{pre}, for the predecessor, and \textbf{iszero}. The evaluation of these terms will be done as for the \textbf{PCF} language, as one can see by Table 5.3. Note that the call-by-name evaluation for the rest of the terms, will be identical to the one in $\mathcal{L}_{\text{rec}}$.

The Stack Machine for $\mathcal{L}_{\text{rec}}^{\text{nat}}$ can be seen in Table 5.4, were the new rules for the successor, predecessor and \textbf{iszero} were added.

Although positive results were seen in the evaluation function, the main difference between $\mathcal{L}_{\text{rec}}$ and $\mathcal{L}_{\text{rec}}^{\text{nat}}$ is noticeable in the compilation function, were the most significant changes are in the successor and predecessor, were instead of a very long $\mathcal{L}_{\text{rec}}$ expression, we now have a one-sized $\mathcal{L}_{\text{rec}}^{\text{nat}}$ expression, Table 5.5.

The implementation of natural numbers on $\mathcal{L}_{\text{rec}}$, started with the addition of new data structure elements to the expression data (Code 5.6).
5.2. Comparing Results

Table 5.5: PCF compilation into $L^\text{nat}_{\text{rec}}$

<table>
<thead>
<tr>
<th>data  Exp = ...</th>
<th>Nat Int</th>
<th>Suc</th>
<th>Pre</th>
<th>Iszer</th>
</tr>
</thead>
<tbody>
<tr>
<td>deriving (Show, Eq)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With these new data structures, naturals were implemented in $L_{\text{rec}}$, along with the new functions $\text{Pre}$ for the predecessor and $\text{Iszer}$ (similar to PCF $\text{iszero}$). Note that the successor no longer has an argument and will now be used as a function, using the application to apply it to an argument. The rest of the terms were kept as they were in $L_{\text{rec}}$.

The implementation of the call-by-name evaluation and Stack Machine can be seen in Appendix Code A.16 and Code A.17, where the code is almost a merge between the $L_{\text{rec}}$ code and the PCF code, and therefore needs no explanation. The same goes for the compilation function (Appendix Code A.18).

In the next section we will show the results of this new implementation.

5.2.2 Comparing Results with $L^\text{nat}_{\text{rec}}$

For the results of the abstract time measure cost (Section 5.2) in the $L^\text{nat}_{\text{rec}}$ system, the evaluation function of this system was also changed (Appendix Code A.19). Note that in this code, there are also special cases for some reductions that do not calculate the cost, as it happened for the PCF language and System $L_{\text{rec}}$.

The results for this new function can be seen on Table 5.6, where the values of the

\[
\begin{align*}
\langle tt \rangle &= \text{Nat }0 \\
\langle ff \rangle &= \text{Nat }1 \\
\langle n \rangle &= \text{Nat }n \\
\langle \text{succ} \rangle &= \text{Suc} \\
\langle \text{pred} \rangle &= \text{Pre} \\
\langle \text{iszero} \rangle &= \text{Iszer} \\
\langle Y_\tau \rangle &= \lambda f.\text{rec} (\text{Nat }1, \text{Nat }0) \mathcal{M}(\langle \tau \rangle) f (\lambda x.\text{let } (y, z) = x \text{ in } \langle S y, z \rangle) \\
\langle \text{cond}_\tau \rangle &= \lambda t u v.\text{rec} (t, \text{Nat }0) u (\lambda x.\text{rec} (\text{Nat }0, \text{Nat }0) (\lambda x.x) x (\lambda x.x)) v (\lambda x.x) \\
\langle x \rangle &= x \\
\langle UV \rangle &= (U)(V) \\
\langle \lambda x^\tau.T \rangle &= \begin{cases} \\
\lambda x.[x^\tau](T) & \text{if } x \in \text{FV}(t) \\
\lambda x.\text{rec} (\text{Nat }0, \text{Nat }0) (\lambda x.x) x (\lambda x.x))(T) & \text{otherwise} \\
\end{cases}
\end{align*}
\]
5.2. COMPARING RESULTS

<table>
<thead>
<tr>
<th>Function</th>
<th>Abstract Time Measure</th>
<th>Number of reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCF</td>
<td>$L_{\text{rec}}^{\text{nat}}$</td>
</tr>
<tr>
<td>add 2 3</td>
<td>56</td>
<td>31</td>
</tr>
<tr>
<td>mult 2 3</td>
<td>247</td>
<td>105</td>
</tr>
<tr>
<td>fib 4</td>
<td>1255</td>
<td>434</td>
</tr>
<tr>
<td>fact 4</td>
<td>22165</td>
<td>3717</td>
</tr>
</tbody>
</table>

Table 5.6: Results Comparison with $L_{\text{rec}}^{\text{nat}}$

<table>
<thead>
<tr>
<th>Function</th>
<th>Abstract Time Measure</th>
<th>Number of reductions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PCF to $L_{\text{rec}}$</td>
<td>PCF to $L_{\text{rec}}^{\text{nat}}$</td>
</tr>
<tr>
<td>add 2 3</td>
<td>4975</td>
<td>871</td>
</tr>
<tr>
<td>mult 2 3</td>
<td>5091</td>
<td>10703</td>
</tr>
<tr>
<td>fib 4</td>
<td>289499</td>
<td>45202</td>
</tr>
<tr>
<td>fact 4</td>
<td>5464324</td>
<td>98520</td>
</tr>
</tbody>
</table>

Table 5.7: Results Comparison Between Compilations

cost and number of reductions for $L_{\text{rec}}^{\text{nat}}$ are better than $L_{\text{rec}}$ for small functions, as add and mult, but get higher with more complex functions. The results for the number of reductions show that $L_{\text{rec}}$ and $L_{\text{rec}}^{\text{nat}}$ have smaller number of reductions needed to evaluate an expression than the PCF language, even though the evaluation with the encoding takes a lot more reductions to achieve normal form. This is expected since the compilation from PCF to $L_{\text{rec}}$ turns the expression terms of PCF into term in $L_{\text{rec}}$ of a much larger length, and therefore will have more reductions before achieving normal form. Note that the number of reductions is higher in the encoding from PCF to $L_{\text{rec}}$ than that of the encoding from PCF to $L_{\text{rec}}^{\text{nat}}$ (Table 5.7), which is also normal since $L_{\text{rec}}^{\text{nat}}$ has smaller length terms than $L_{\text{rec}}$ after encoding them from PCF. However, note that the number of reductions is smaller in $L_{\text{rec}}$ than that of $L_{\text{rec}}^{\text{nat}}$, since terms originally created in $L_{\text{rec}}$ are faster to evaluate than that of terms originally created in $L_{\text{rec}}^{\text{nat}}$.

As for the abstract time measure, as one can see in Figure 5.1, PCF terms have a much higher cost than the terms of $L_{\text{rec}}$ and $L_{\text{rec}}^{\text{nat}}$, specially in more complex functions. Which comes to show that System $L_{\text{rec}}$ is truly more efficient evaluating expressions with a recursor, than the fixed point operator of the PCF.

In Figure 5.2, one can see the difference between encoding from PCF to $L_{\text{rec}}$ and
5.2. COMPARING RESULTS

Figure 5.1: Abstract time measure results

encoding from $\textbf{PCF}$ to $L_{\text{nat}}^{\text{rec}}$. Once again, this happens because when the terms are encoded to $L_{\text{rec}}$, their length increases almost exponentially which makes them more expensive to reduce to normal form. And when the terms are encoded to $L_{\text{rec}}^{\text{nat}}$, due to the built-in naturals, the $\text{succ}$, the $\text{pred}$ and the $\text{iszero}$ function, the terms length do not have such a large increase and therefore do not have to go through so many reductions in order to achieve normal form. The main reason relies on the problem that if the function has a large natural number, $L_{\text{rec}}$ will transform it into a long line of application of successors that complicates the evaluation.

In the next chapter we will take some conclusions about these results.
Figure 5.2: Abstract time measure results for the encodings
Chapter 6

Conclusion

Throughout this thesis we discussed the implementation of:

- **PCF**, a λ-calculus based language;
- System $L_{rec}$, a linear language, based on the linear λ-calculus;
- $L_{rec}^{nat}$, an extension of System $L_{rec}$ with built-in naturals.

The implementations were started with a parser in Happy that encoded the input in PCF, $L_{rec}$ or $L_{rec}^{nat}$, into a created data structure in Haskell, in order to evaluate the expressions of the languages. We also compare the abstract time measure of their evaluation in order to assess the efficiency of $L_{rec}$ when compared to PCF.

For some of the functions analyzed, the encoding in $L_{rec}$ proved to be more efficient than the respective encoding in PCF and even more than the encoding resulting from translating from PCF to $L_{rec}$. In that sense, future work can be done in the encoding function from PCF to $L_{rec}$, so that the length of terms do not increase so drastically and therefore improve the evaluation of such terms.

This improvement would have to take advantage of the recursor. Note that in functional programming languages a way to improve poor performance programs is to transform recursive functions into more efficient versions [Harrison 92], sometimes using an iterative solution. This can be a technique used when translating programs written using $Y$ into programs using the $L_{rec}$ recursor.

Note also that with the built-in naturals, System $L_{rec}$ showed improved results when encoding from PCF to $L_{rec}^{nat}$, which supports that $L_{rec}$ can also be improved with built-in arithmetic functions and by adding constructors so it can be used as a pro-
gramming language or an intermediate code, as the Spineless Tagless G-machine (STG [Jones 92]) is for Haskell.
Appendix A

Code

A.1 PCF Language

Code A.1: Evaluation Function

```
 eval :: ExpPCF -> ExpPCF
 eval (Var a) = (Var a)
 eval (Lambda x m) = (Lambda x m)
 eval (Nat x) = (Nat x)
 eval (Succ) = (Succ)
 eval (Pred) = (Pred)
 eval (Cond) = (Cond)
 eval (IsZ) = (IsZ)
 eval (PtoF) = (PtoF)
 eval (App PtoF m) = eval (App m (App PtoF m))
 eval (App (Lambda x m) n) = eval (subst m x n)
 eval (App (App (App Cond TT) m) n) = eval (m)
 eval (App (App (App Cond FF) m) n) = eval (n)
 eval (App (App (App Cond b) m) n) =
     eval (App (App (App Cond (eval b)) m) n)
 eval (App Succ m) = case (eval m) of
     (Nat n) -> (Nat (n+1))
     t     -> (App Succ t)
 eval (App Pred m) = case (eval m) of
     (Nat (n+1)) -> (Nat n)
     t     -> (App Pred t)
 eval (App IsZ (Nat n)) = if (n==0) then (TT) else (FF)
 eval (App IsZ m) = eval (App IsZ (eval m))
 eval (App s t) = eval_aux (App (eval s) t)
 eval t = t
```
Code A.2: Substitution Function

```haskell
substt :: ExpPCF -> String -> ExpPCF -> ExpPCF
substt (Nat a) y l = (Nat a)
substt (Succ) y l = (Succ)
substt (Pred) y l = (Pred)
substt (IsZ) y l = (IsZ)
substt (PtoF) y l = (PtoF)
substt (Cond) y l = (Cond)
substt (Var x) y l =
    | x == y = 1
    | otherwise = (Var x)
substt (Lambda x m) y l =
    | x==y = (Lambda x m)
    | otherwise = (Lambda x (substt m y l))
substt (App m n) y l = (App (substt m y l) (substt n y l))
```

Code A.3: Stack Function

```haskell
stack :: (ExpPCF, Stack) -> (ExpPCF, Stack)
stack ((App m n), s) = stack (m, (E n):s)
stack ((Lambda x m), (E n):s) = stack ((substt m x n), s)
stack ((PtoF), (E m):s) = stack (m, (E (App PtoF m)):s)
stack (Cond, (E m):(E n1):(E n2):s) = stack (m, COND(n1, n2):s)
stack (TT, (COND (n1, n2)):s) = stack (n1, s)
stack (FF, (COND (n1, n2)):s) = stack (n2, s)
stack ((Nat n), (E (Succ)):s) = stack ((Nat (n+1)), s)
stack ((Succ), (E m):s) = stack (m, (E (Succ)):s)
stack ((Nat 0), (E (Pred)):s) = stack ((Nat 0), s)
stack ((Nat (n+1)), (E (Pred)):s) = stack ((Nat n), s)
stack ((Pred), (E m):s) = stack (m, (E (Pred)):s)
stack ((Nat 0),(E (IsZ)):s) = stack (TT, s)
stack ((Nat (n+1)),(E (IsZ)):s) = stack (FF, s)
stack ((IsZ), (E m):s) = stack (m, (E (IsZ)):s)
stack (v, s) = (v, s)
```

A.2 \( \mathcal{L}_{rec} \) Language

Code A.4: call-by-name Evaluation Function

```haskell
eval_cbn :: Exp -> Exp
eval_cbn (Suc a)
```
eval_cbn (Lambda a b) = (Lambda a b)

| (value a)            = (Suc a)
| otherwise            = (Suc (eval_cbn_a))

Code A.5: call-by-value Evaluation Function

eval_cbn (App (Lambda x u) t) = eval_cbn(subst t u x t)

eval_cbn (App s t) = cbv_aux(App (eval_cbn s) t)

eval_cbn (Let (x, y) (Pair t1 t2) u) =
  | eval_cbn (App (App (Lambda x (Lambda y (u))) t1) t2)
  |
  | eval_cbn (App (App (Lambda x y) (Pair t1 t2)) u v w)
  
Code A.6: Auxiliary Evaluation Function for CBV

eval_cbn (Rec (Pair (Zero) t2) u v w) = eval_cbn u

eval_cbn (Rec (Pair (Suc t) t2) u v w) =
  | eval_cbn (App v (Rec (App w (Pair t t2)) u v w))
  |
  | eval_cbn (Rec (Pair (eval_cbn p) u v w)

eval_cbn (Rec p u v w) = eval_cbn (Rec (eval_cbn p) u v w)

eval_cbn t = t

Code A.7: Substitution Function

cbv_aux :: Exp -> Exp
cbv_aux (App (Lambda x u) t) = eval_cbn(subst u x (eval_cbn t))

cbv_aux (App s t) = (App s t)
Code A.8: Stack Function

```
stack :: (Exp, Stck) -> (Exp, Stck)
stack ((App m n), s)       = stack (m, (E n):s)
stack ((Lambda x m), (E n):s) = stack ((subst m x n),s)
stack ((Let (x,y) n m),s)  = stack (n, LET(x,y,m):s)
stack ((Pair n1 n2), (LET (x,y,m))):s) =
    stack ((subst m x n1) y n2), s)
stack ((Rec n u v w), s) = stack (n, REC(u,v,w):s)
stack ((Pair n1 n2), (REC (u,v,w))):s) = stack (n1, RECC(n2, u, v, w):s)
stack ((Zero), (RECC (t,u,v,w))):s) = stack (u,s)
stack ((Suc n), (RECC (t,u,v,w))):s) =
    stack (v,(E (Rec (App w (Pair n t)) u v w))):s)
stack ((Suc n), SUCC):s) =
    if (value n) then stack (Suc (Suc n), s) else stack (n, SUCC:SUCC):s)
stack ((Suc n), s) =
    if (value n) then (Suc n),s) else stack (n, SUCC):s)
stack (n, SUCC):s) =
    if (value n) then ((Suc n),s) else ((Suc n),s)
stack (v,s) = (v,s)
```

A.3 Compilation

Code A.9: Compilation Function

```
comp :: ExpPCF -> Exp
```

Operational Semantics for Linear Languages 78
comp (TT) = (Zero)
comp (FF) = (Suc (Zero))
comp (NatPCF 0) = (Zero)
comp (NatPCF (n+1)) = (Suc (comp_nat n))
comp (Succ) =
  (Lambda "n" (Rec (Pair (Var "n") (Zero)) (Suc (Zero)))
   (Lambda "x" (Suc (Var "x"))) idntt))
comp (Pred) =
  (Lambda "n" (App pr1 (Rec (Pair (Var "n") (Zero))) (Pair (Zero) (Zero))
   (Lambda "x" (Let ("t", "u") (App (dup (TInt)) (App pr2 (Var "x"))))
   (Pair (Var "t") (Suc (Var "u")))))))
comp (IsZ) =
  (Lambda "n" (App pr1 (Rec (Pair (Var "n") (Zero))) (Suc (Zero))
   (Lambda "x" (App (dup (TInt))
    (Pair (Var "x"))))) idntt))
comp (PtoF a) =
  (Lambda "f" (Rec (Pair (Suc (Zero)) (Zero)) (make a) (Var "f"))
   (Lambda "x" (Let ("y", "z") (Var "x"))
    (Pair (Suc (Var "y")) (Var "z"))))
comp (Cond) =
  (Lambda "t" (Lambda "u" (Lambda "v" (Rec (Pair (Var "t") (Zero))
    (Var "u") (Lambda "x" (App (Rec (Pair (Zero) (Zero)) idntt
     (Var "x"))))) idntt)))
comp (VarPCF x) = (Var x)
comp (AppPCF u v) = (App (comp u) (comp v))
comp (LambdaPCF (x, tp) t)
  | (elem (VarPCF x) (fvcPCF t)) = (Lambda x (linearFV (Var x, tp) (comp t)))
  | otherwise =
    (Lambda x (App (Rec (Pair (Zero) (Zero)) idntt (Var x) idntt) (comp t)))

Code A.10: Linearization Function

linearFV :: (Exp, Type) -> Exp -> Exp
linearFV (x, tp) (Suc u) = (Suc (linearFV (x, tp) u))
linearFV (x, tp) (Lambda y u) = (Lambda y (linearFV (x, tp) u))
linearFV (x, tp) (App s u)
  | (elem x (fvc s)) && (elem x (fvc u)) =
    (copy ("x1", "x2") (x, tp) (App (substt (linearFV (x, tp) s) "x1" x)
     (substt (linearFV (x, tp) u) "x2" x)))
  | not (elem x (fvc u)) = (App (linearFV (x, tp) s) u)
  | not (elem x (fvc s)) = (App s (linearFV (x, tp) u))
linearFV (x, tp) (Let (a, b) u v) = (Let (a, b) u (linearFV (x, tp) v))
linearFV (x, tp) (Var y) = x

code A.11: Erase Function

Operational Semantics for Linear Languages
Code A.12: Make Function

\[
\text{make} :: \text{Type} \rightarrow \text{Exp} \\
\text{make} (\text{TInt}) = (\text{Zero}) \\
\text{make} (\text{TBool}) = (\text{Zero}) \\
\text{make} (\text{TApp } t a t b) = (\text{Lambd } "x" (\text{App } (\text{erase } ((\text{Var } "x"), t a)) (\text{make } t b)))
\]

Code A.13: PCF expression size

\[
\text{esizePCF} :: \text{ExpPCF} \rightarrow \text{Int} \\
\text{esizePCF} (\text{TT}) = 1 \\
\text{esizePCF} (\text{FF}) = 1 \\
\text{esizePCF} (\text{NatPCF } n) = 1 \\
\text{esizePCF} (\text{Succ}) = 1 \\
\text{esizePCF} (\text{Pred}) = 1 \\
\text{esizePCF} (\text{IsZ}) = 1 \\
\text{esizePCF} (\text{PtoF } t p) = 1 \\
\text{esizePCF} (\text{Cond}) = 1 \\
\text{esizePCF} (\text{VarPCF } x) = 1 \\
\text{esizePCF} (\text{AppPCF } n m) = (\text{esizePCF } n) + (\text{esizePCF } m) \\
\text{esizePCF} (\text{LambdPCF } (x, t p) t) = 1 + (\text{esizePCF } t)
\]

Code A.14: $L_{rec}$ evaluation with Abstract Time Measure

\[
\text{type} \quad \text{Counter} = \text{Int} \\
\text{type} \quad \text{Cost} = \text{Int} \\
\text{eval_cbn} :: \text{Exp} \rightarrow \text{Counter} \rightarrow (\text{Exp, Counter, Cost}) \\
\text{eval_cbn} (\text{Lambd } x u) n = ((\text{Lambd } x u), n, 0) \\
\text{eval_cbn} (\text{Pair } a b) n = ((\text{Pair } a b), n, 0) \\
\text{eval_cbn} (\text{Suc } a) n \\
\hspace{1cm} | (\text{value } a) = ((\text{Suc } a), n, 0) \\
\hspace{1cm} | \text{otherwise} = \\
\hspace{2cm} \text{let } (e, nr, it) = (\text{eval_cbn } a (n+1)) \\
\hspace{2cm} \hspace{1cm} \text{fi} = (\text{max } 1 ((\text{esizeLRec } a) - (\text{esizeLRec } (\text{Suc } a)))) \\
\hspace{2cm} \hspace{1cm} \text{in } ((\text{Suc } e), nr, fi+it) \\
\text{eval_cbn} (\text{App } (\text{Lambd } x u) t) n = \\
\hspace{1cm} \text{let } (e, nr, it) = (\text{eval_cbn } (\text{subst } u x t) (n+1)) \\
\hspace{2cm} \hspace{1cm} \text{fi} = (\text{max } 1 ((\text{esizeLRec } (\text{subst } u x t)) - \\
\hspace{3cm} (\text{esizeLRec } (\text{App } (\text{Lambd } x u) t)))) \\
\hspace{2cm} \hspace{1cm} \text{in } (e, nr, fi+it)
\]


```haskell
18  eval_cbn (App s t) n =  
    let (e1, nr1, it) = (eval_cbn s n)  
        (e2, nr2, it2) = (cbn_aux (App e1 t) nr1)  
    in (e2, nr2, it2+it)

20  eval_cbn (Let (x, y) (Pair t1 t2) u) n =  
    let (e, nr, it) = (eval_cbn (subst (substt u x t1) y t2) (n+1))  
        fi = (max (1 ((esizeLRec (subst (subst t x t1) y t2))−  
                     (esizeLRec (Let (x, y) (Pair t1 t2) u))))))  
    in (e, nr, fi+it)

22  eval_cbn (Lett (x, y) (Pair t1 t2) u v w) n =  
    let (e, nr, it) = (eval_cbn (Lett (x, y) (Pair t1 t2) u v w) (n+1))  
        fi = (max (1 ((esizeLRec u)−  
                     (esizeLRec (Lett (x, y) (Pair t1 t2) u v w)))))  
    in (e, nr, fi+it)

24  eval_cbn (Rec (Pair (Zero) t2) u v w) n =  
    let (e, nr, it) = (eval_cbn (Rec (Pair (Zero) t2) u v w) (n+1))  
        fi = (max (1 ((esizeLRec (Rec (Pair (Zero) t2) u v w))−  
                     (esizeLRec (Rec (Pair (Zero) t2) u v w))))))  
    in (e, nr, fi+it)

26  eval_cbn (Rec (Pair (Suc t) t2) u v w) n =  
    let (e, nr, it) = (eval_cbn (Rec (Pair (Suc t) t2) u v w) (n+1))  
        fi = (max (1 ((esizeLRec (Rec (Pair (Suc t) t2) u v w))−  
                     (esizeLRec (Rec (Pair (Suc t) t2) u v w))))))  
    in (e, nr, fi+it)

28  eval_cbn (Rec (Pair p1 p2) u v w) n =  
    let (e, nr, it) = (eval_cbn (Rec (Pair p1 p2) u v w) (n+1))  
        fi = (max (1 ((esizeLRec (Rec (Pair p1 p2) u v w))−  
                     (esizeLRec (Rec (Pair p1 p2) u v w))))))  
    in (e, nr, fi+it)

30  eval_cbn t n = (t, n, 0)
```

---

**Code A.15: PCF evaluation with Abstract Time Measure**

```haskell
1  type Counter = Int
2  type Cost = Int
3
4  eval :: ExpPCF -> Counter -> (ExpPCF, Counter, Cost)
5  eval (Var a) n = ((Var a), n, 0)
6  eval (Lambda x m) n = ((Lambda x m), n, 0)
7  eval (Nat x) n = ((Nat x), n, 0)
8  eval (Succ) n = ((Succ), n, 0)
9  eval (Pred) n = ((Pred), n, 0)
10  eval (Cond) n = ((Cond), n, 0)
```

---

*Operational Semantics for Linear Languages*
A.3.1 \( \mathcal{L}_{\text{rec}}^{\text{nat}} \) Code

### Operational Semantics for Linear Languages

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>eval (IsZ) n = ((IsZ), n, 0)</td>
</tr>
<tr>
<td>eval (PtoF) n = ((PtoF), n, 0)</td>
</tr>
<tr>
<td>eval (App PtoF m) n = let (e, nr, c) = (eval (App m (App PtoF m)) (n+1)) fc = (esizePCF m)</td>
</tr>
<tr>
<td>in (e, nr, fc+c)</td>
</tr>
<tr>
<td>eval (App (Lambda x m) n) nr =</td>
</tr>
</tbody>
</table>
| let (e, nr1, c) = (eval (subst m x n) (nr+1)) fc = (max 1 ((esizePCF (subst m x n))−)
| (esizePCF (App (Lambda x m) n))) |
| in (e, nr1, fc+c) |
| eval (App (App (App Cond TT) m) n) nr = |
| let (e, nr1, c) = (eval m (nr+1)) fc = (max 1 ((esizePCF m)−)
| (esizePCF (App (App (App Cond TT) m) n))) |
| in (e, nr1, c+fc) |
| eval (App (App (App Cond FF) m) n) nr = |
| let (e, nr2, c) = (eval n (nr+1)) fc = (max 1 ((esizePCF n)−)
| (esizePCF (App (App (App Cond FF) m) n))) |
| in (e, nr2, c+fc) |
| eval (App (App (App Cond b) m) n) nr = |
| let (e1, nr1, c1) = (eval b nr) |
| (e2, nr2, c2) = (eval (App (App (App Cond e1) m) n) nr1) |
| in (e2, nr2, c2+c1) |
| eval (App Succ m) n = case (eval m n) of |
| ((Nat n), nr, c) −> ((Nat (n+1)), (nr+1), c+1) |
| (t, nr, c) −> ((App Succ t), nr, c) |
| eval (App Pred m) n = case (eval m n) of |
| ((Nat (n+1)), nr, c) −> ((Nat n), (nr+1), c+1) |
| (t, nr, c) −> ((App Pred t), nr, c) |
| eval (App IsZ (Nat n)) nr = if (n==0) |
| then (TT, (nr+1), 1) |
| else (FF, (nr+1), 1) |
| eval (App IsZ m) n = let (e1, nr1, c) = (eval m n) |
| (e2, nr2, c2) = (eval (App IsZ e1) nr1) |
| in (e2, nr2, c+c2) |
| eval (App s t) n = let (e1, nr1, c) = (eval s n) |
| (e2, nr2, c2) = (eval_aux (App e1 t) nr1) |
| in (e2, nr2, c+c2) |
| eval t n = (t, n, 0) |
Code A.16: $L_{\text{nat}}^{\text{rec}}$ call-by-name Evaluation Function

```plaintext
eval_cbn :: Exp \rightarrow Exp
eval_cbn (Lambd a b) = (Lambd a b)
eval_cbn (Pair a b) = (Pair a b)
eval_cbn (App (Lambd x u) t) = eval_cbn(substt u x t)
eval_cbn (App (Iszer) n) = if ((eval_cbn n) == (Nat 0)) then (Nat 0) else (Nat 1)
eval_cbn (App (Suc) n) = if (isnumber num) then (Nat (numS num)) else (App (Suc) n)
where
num = (eval_cbn n)
eval_cbn (App (Pre) n) = if (isnumber num) then (Nat (numP num)) else (App (Pre) n)
where
num = (eval_cbn n)
eval_cbn (App s t) = cbn_aux(App (eval_cbn s) t)
eval_cbn (Let (x, y) (Pair t1 t2) u) = eval_cbn (App (Lambd x (Lambd y (u))) t1) t2)
eval_cbn (Let (a, b) p u) = eval_cbn (Let (a, b) (eval_cbn p) u)
eval_cbn (Rec (Pair (Nat 0) t2) u v w) = eval_cbn u
10 eval_cbn (Rec (Pair (Nat (n+1)) t2) u v w) = eval_cbn (App v (Rec (App w (Pair (Nat n) t2)) u v w))
eval_cbn (Rec (Pair p1 p2) u v w) = eval_cbn (Rec (Pair (eval_cbn p1) p2) u v w)
eval_cbn (Rec p u v w) = eval_cbn (Rec (eval_cbn p) u v w)
eval_cbn t = t
```

Code A.17: $L_{\text{nat}}^{\text{rec}}$ Stack Machine

```plaintext
stck :: (Exp, Stck) \rightarrow (Exp, Stck)
stck ((App m n), s) = stck (m, (E n):s)
stck ((Lambda x m), (E n):s) = stck ((substt m x n), s)
stck ((Nat n), (E (Suc)):s) = stck ((Nat (n+1)), s)
stck ((Suc), (E m):s) = stck (m, (E (Suc)):s)
stck ((Nat 0), (E (Pre)):s) = stck ((Nat 0), s)
stck ((Nat (n+1)), (E (Pre)):s) = stck ((Nat n), s)
stck ((Pre), (E m):s) = stck (m, (E (Pre)):s)
stck ((Nat 0), (E Iszer):s) = stck ((Nat 0), s) -- True
stck ((Nat (n+1)), (E Iszer):s) = stck ((Nat 1), s) -- False
stck ((Iszer), (E n):s) = stck (n, (E Iszer):s)
stck ((Let (x, y) n m), s) = stck (n, LET(x, y, m):s)
stck ((Pair n1 n2), (LET (x, y, m):s) =
  stck ((substt (substt m x n1) y n2), s)
```

Operational Semantics for Linear Languages 83
Code A.18: PCF to $\mathcal{L}_{\text{rec}}^{\text{nat}}$ compilation

```plaintext
comp :: ExpPCF -> Exp
comp (TT) = (Nat 0 )
comp (FF) = (Nat 1 )
comp (NatPCF n) = (Nat n )
comp (Succ) = (Suc)
comp (Pred) = (Pre)
comp (ISZ) = (ISzer)
comp (PtoF a) = (Lambd "f" (Rec (Pair (Nat 1 ) (Nat 0 )))
                (make a) (Var "f") (Lambd "x"
                (Let ("y","z") (Var "x")
                (Pair (App (Suc) (Var "y")) (Var "z"))))))
comp (Cond) = (Lambd "t" (Lambd "u" (Lambd "v"
                (Rec (Pair (Var "t") (Nat 0 )) (Var "u"))
                (Lambd "x" (App (Rec
                (Pair (Nat 0 ) (Nat 0 )) idntt (Var "x") idnttt)
                (Var "v"))) idnttt))))
comp (VarPCF x) = (Var x)
comp (AppPCF u v) = (App (comp u) (comp v))
comp (LambdPCF (x,tp) t)
  | (elem (VarPCF x) (fvcPCF t)) =
      (Lambd x (linearFV (Var x, tp) (comp t)))
  | otherwise =
      (Lambd x (App (Rec (Pair (Nat 0 ) (Nat 0 )) idntt (Var x)
      idntt ) (comp t)))
```

Code A.19: $\mathcal{L}_{\text{rec}}^{\text{nat}}$ with Abstract Time Measure

```plaintext
eval_cbn :: Exp -> Counter -> (Exp, Counter, Cost)
eval_cbn (Lambd x u) n = ((Lambd x u),n,0)
eval_cbn (Pair a b) n = ((Pair a b),n,0)
eval_cbn (App (Lambd x u) t) n =
  let (e,nr,it) = (eval_cbn (substt u x t) (n+1))
    fi = (max 1 (esizeLRecN (substt u x t))-
      (esizeLRecN (App (Lambd x u) t))))
  in (e,nr,fi+it)
eval_cbn (App (ISzer) (Nat n)) nr = if (n==0)
  then ((Nat 0),(nr+1),1)
```

Operational Semantics for Linear Languages
Code

```haskell
11     eval_cbn (App (Iszer) m) n =
12         let (e1,nr1,c) = (eval_cbn m n)
13             (e2,nr2,c2) = (eval_cbn (App Iszer e1) nr1)
14         in (e2,nr2,c+c2)
15     eval_cbn (App (Suc) m) n =
16         let (e,nr,c) = (eval_cbn m n)
17             c1 = (max 1 ((esizeLRecN (App Suc e)) - (esizeLRecN (App Suc m))))
18         in if (isnumber e)
19             then ((Nat (numS e)),(nr+1),c+1) else ((App (Suc) e),nr,c+c1)
20     eval_cbn (App Pre m) n =
21         let (e,nr,c) = (eval_cbn m n)
22             c1 = (max 1 ((esizeLRecN m)-(esizeLRecN (App Pre m))))
23         in if (isnumber e)
24             then ((Nat (numP e)),(nr+1),c+1) else ((App (Pre) m),nr,c+c1)
25     eval_cbn (App s t) n =
26         let (e1,nr1,c) = (eval_cbn s n)
27             (e2,nr2,c2) = (cbn_aux (App e1 t) nr1)
28         in (e2,nr2,c+c2)
29     eval_cbn (Let (x, y) (Pair t1 t2) u) n =
30         let (e,nr,c) = (eval_cbn (subtt (subtt u x t1) y t2) (n+1))
31             fc = (max 1 ((esizeLRecN (subtt (subtt u x t1) y t2))-
32                              (esizeLRecN (Let (x, y) (Pair t1 t2) u))))
33         in (e,nr,fc+c)
34     eval_cbn (Let (a, b) p u) n =
35         let (e,nr,c) = (eval_cbn p n)
36             (e1,nr1,c1) = (eval_cbn (Let (a, b) e u) nr)
37         in (e1,nr1,c+c1)
38     eval_cbn (Rec (Pair (Nat 0) t2) u v w) n =
39         let (e,nr,it) = (eval_cbn u (n+1))
40             fc = it+(max 1 ((esizeLRecN u)-
41                              (esizeLRecN (Rec (Pair (Nat 0) t2) u v w))))
42         in (e,nr,fc)
43     eval_cbn (Rec (Pair (Nat (n+1)) t2) u v w) nr =
44         let (e,nr1,it) =
45             (eval_cbn (App v (Rec (App w (Pair (Nat n) t2)) u v w)) (nr+1))
46             fc = it + (max 1 ((esizeLRecN (App v (Rec (App w (Pair (Nat n) t2)) u v w)))-
47                              (esizeLRecN (Rec (Pair (Nat (n+1)) t2) u v w))))
48         in (e,nr1,fc)
49     eval_cbn (Rec (Pair p1 p2) u v w) n =
50         let (e2,nr2,it) = (eval_cbn p1 n)
51             (e1,nr1,it2) = (eval_cbn (Rec (Pair e2 p2) u v w) nr2)
52         in (e1,nr1,it2+it)
53     eval_cbn (Rec p u v w) n =
```

Operational Semantics for Linear Languages
let \((e_2, n_{r2}, i_t) = (\text{eval}_c b_n \ p \ n)\) 
\((e_1, n_{r1}, i_{t2}) = (\text{eval}_c b_n \ (\text{Rec} \ e_2 \ u \ v \ w) \ n_{r2})\) 
in \((e_1, n_{r1}, i_{t2+it})\)

\text{eval}_c b_n t \ n \quad = (t, n, 0)
References


