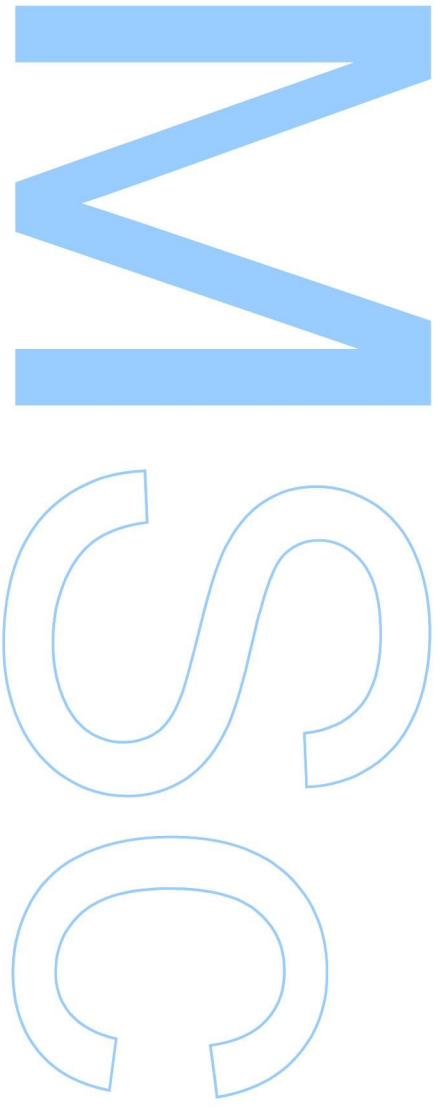
Resource aware calculi for programming languages

Inês Duarte Lima
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2019

Orientador
Sandra Alves, Professor Auxiliar, Faculdade de Ciências da Universidade do Porto

Coorientador
Mário Florido, Professor Associado, Faculdade de Ciências da Universidade do Porto



U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, / /

Abstract

In this document we will present work related to resource calculi, where an argument can be copied as many times as needed. Resource calculi could have a major importance when talking about memory space, time, or number of steps in operational semantics. Our work focus on two calculi: λ-calculus with multiplicities and resource calculi. In this work, we prove the existence of a relation between the two calculi and present a translation between them.

Resumo

Nesta tese, apresentaremos o trabalho relacionado ao cálculo de recursos, onde um argumento pode ser copiado as vezes que for necessário. O cálculo de recursos pode ter uma grande importância quando falamos sobre espaço na memória, tempo ou número de passos na semântica operacional. O nosso trabalho concentra-se em dois cálculos: λ-calculus com multiplicidades e cálculos de recursos. Neste trabalho, provamos a existência de uma relação entre os dois cálculos e apresentamos uma tradução entre eles.

Acknowledgements

First of all, I would like to thank to my advisor and co-advisor Sandra Alves and Mário Florido for all the patience with me and for being comprehensive and honest. They always tried to put me in the right path and aware of reality.

Then, I would like to thank my family, my parents and my sister, for always believing that I was capable of this. They kept motivating me and tried to relieve some of the stress.

I also want to give a big thank you to my boyfriend, Alberto Barbosa. He saw me in my worst phase but was always there to motivate me and helped me. Thank you for not letting me fall.

Finally, thank you to my friends. To my group of lunch that in that hour always could make me smile. To Vanda, who made the thesis in the same year as me, for crying and laughing with me when things weren't going to well. To Claudia and Marina, that are my friends since the 1st year. They were also there for my ups and downs. And, finally to my great friend Inês, that stressed with me, but then we calmed each other down each.

Thank you to you all.

I dedicate this to everyone

Contents

Chapter 1

Introduction

Formal systems for proving properties of programs are, nowadays, crucial for computer science. One important property to be checked is the amount of resources that the execution of a given program will consume. Resources can be of different kinds such as memory space, time or number of steps in operational semantic. To address this problem Boudol [8], Ronchi Della Rocca, *et al* [24], and others have done work related to the λ-calculus with resource consumption.

Boudol [7] focus his work on the λ-calculus with multiplicities. In his calculus he presents the notion of bag, that consists of the argument of a function, which is a multiset. It is the bag that indicates how many copies of the argument are available. Based on the work of Boudol [7], Ronchi Della Rocca, *et al* [24] beyond the notion of bags, came up with the notion of resources. According to her, it is the resource who control the number of uses of an argument.

Our main goal with this work is to study the relation between these two calculi. In order to achieve this, we will implement theoretical models for resource calculi and the λ-calculus with multiplicities, to fully understand the details of both approaches. We first started by implementing the λ-calculus with multiplicities, in Haskell. After that, we implemented the resource calculus setting according to Ronchi Della Rocca, *et al* [24]. The next step was to prove the relation between the two calculi. We decided to start the relation from the resource calculus to the λ-calculus with multiplicities. We made this decision based on the grammar. Since resource calculus presents everything in more detail, it seamed more relevant to follow the approach. After the implementation and the tests, we tried to prove the other way around, that the λ-calculus with multiplicities can be transformed into the resource calculus. Our main problem implementing these translations was that the two calculi, although similar, have important differences that needed to be attended.

- The resource calculus works with lists, while the λ-calculus with multiplicities does not. To resolve this problem Boudol [7] uses the idea of par $(P|Q)$.
- The λ-calculus with multiplicities has explicit substitutions, while the other calculus does not have explicit substitutions in its grammar.

1.1 Outline

This thesis will be organised as follows.

- **Chapter 2:** we present some notions of topics that we believed will help the reader fully understanding our work.
- **Chapter 3:** we present the λ-calculus with multiplicities. We explain the calculus and then we present an implementation with examples for this calculus.
- **Chapter 4:** we present the resource calculus where we explain the calculus and then, we also present our implementation and some examples.
- **Chapter 5:** we present our translation from one calculus to the other. We start by defining some theorems that will help us on this subject and then we present our implementation and also some examples.
- **Chapter 6:** we present the conclusion of our work, some future work and some final remarks.

Chapter 2

Background

In this chapter we present some relevant notions on the λ-calculus.

First, we will introduce some basic notions of the λ-calculus [5] for the sake of self-completeness and to present to the user the theoretical basis behind both our work and some related work.

We will talk about some concepts about the resource calculus, which is an extension of the usual λ-calculus.

2.1 λ-calculus

The λ-calculus was introduced in the 1930s by Alonzo Church [9] as a way of formalizing the concept of effective computability. The λ-calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism, so it is computationally equivalent to Turing machines.

The central concept in the λ-calculus is the *term*, represented in this context as $M, N, \ldots$. A *variable* is an identifier that can be any letter, represented in the context of this document as $x, y, z, \ldots$. We start by defining the set of the λ-terms admissible.

Definition 1. *Let $\mathcal{V}$ be an infinite set of variables. The set Λ of λ-terms, is inductively defined from $\mathcal{V}$ as follows:*

$$\begin{array}{ll} x \in \mathcal{V} \Rightarrow x \in \Lambda & \\ M, N \in \Lambda \Rightarrow (MN) \in \Lambda & (\textit{Application}) \\ M \in \Lambda, x \in \mathcal{V} \Rightarrow (\lambda x.M) \in \Lambda & (\textit{Abstraction}) \end{array}$$

$\Diamond$

We adopt the convention that function application associates from the left, that is

$$(M_1 M_2 M_3 \dots M_n) \equiv (\dots((M_1 M_2) M_3) \dots M_n).$$

On the other hand, abstractions are right associative, that is

$$(\lambda x_1 x_2 \dots x_n.M) \equiv (\lambda x_1 (\lambda x_2 (\dots (\lambda x_n.M) \dots))).$$

In the λ-calculus variables can occur free or bound. We say that a variable x occurs bound in a term M if it is in the scope of an abstraction λx in M. Otherwise the variable x occurs free in M. We call *fv(L)* to the set of free variables of term L.

Example 1. In the λ-term $(\lambda x.x)(\lambda y.yx)$ we say that x occurs bound in $(\lambda x.x)$ and free in $(\lambda y.yx)$. ◄

The term presented in Example 1 describes an application. In order to compute the result of the application, one must be familiar with the concept of substitution in λ-calculus.

Definition 2. *We define the substitution of the free occurrences of x by a term L in M, and denote it as $M[L/x]$, as:*

$$y[L/x] \equiv \begin{cases} L \ \textit{if } x \equiv y, \\ y \ \textit{otherwise} \end{cases}$$

$$(MN)[L/x] \equiv (M[L/x])(N[L/x])$$

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

◊

Example 2. If we consider the term presented in Example 1 and take each of the parenthesized terms individually, we could say that:

$$(\lambda x.x)[y/x] \equiv (\lambda x.x)$$

since x is bounded in $(\lambda x.x)$. However, if we consider the the right side of the application, then:

$$(\lambda y.yx)[z/x] \equiv (\lambda y.yz)$$

◄

The substitution process exemplified in Example 2 does not come without some concerns. One needs to be careful when performing substitution, because a phenomenon known as *name*

capture can occur. This means that a variable that occurred free in L before the substitution, due to possessing the same name as a bound variable in M, might be *captured* by the abstraction in M, when performing $M[L/x]$, if there is a free occurrence of x in M under an abstraction $\lambda y.M$ and y belongs to $fv(L)$.

Definition 3. *A change of a bound variable x in a term M is the substitution of sub-terms of M of the form $(\lambda x.N)$ by $(\lambda y.N[y/x])$, where y is a variable that does not occurs in N.*

Changing bound variables preserves the meaning of the term, in the sense that it represents the same function. We call this notion α-congruence:

Definition 4. *M is α-congruent with N, (notation $M \equiv_\alpha N$), if N can be obtained from M by a series of changes of bound variables, and vice-versa.*

Example 3. Two terms that are reducible to each other by α-conversions are α-equivalent.

1. $\lambda x.x \equiv_\alpha \lambda y.y$
2. $\lambda xy.yx \equiv_\alpha \lambda zw.wz$
3. $\lambda x.M \equiv_\alpha \lambda M.[y/x]$ if $y \notin fv(M)$

A reducible term, or *redex*, is any term to which the main computation rule of the λ-calculus called *β-reduction* can be immediately applied.

We will now introduce a formal definition of β-reduction.

Definition 5. *The notion of β-reduction on Λ is defined as follows:*

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

$\Diamond$

Example 4. We say that $(\lambda y.y)z$ is a redex, more precisely a β-redex, and $\lambda x.(\lambda y.y)z$ is not. ◄

We call $(\lambda x.M)N$ a β-redex and $M[N/x]$ its *β-contractrum*.

Example 5. According to the Definition 5 we can now apply β-reduction to the λ-term presented in Example 1.

$$(\lambda x.x)(\lambda y.yx) \to_\beta (\lambda y.yx)$$

If we think about the result obtained through applying β-reduction to the λ-term above, we conclude that the first term corresponds to the λ-calculus codification of the identity function (I), so, applying the identity function to any other function, would yield the second function, by the known properties of the identity function. ◄

Reduction is lifted to general terms, denoted by $M \to N$, by applying β-reduction to some subterm of M to obtain N. More formally:

$$\frac{M \to^{\beta} M'}{M \to M'}$$

$$\frac{M \to M'}{(\lambda x.M) \to (\lambda x.M')}$$

$$\frac{M \to M'}{(MN) \to (M'N)}$$

$$\frac{M \to M'}{(NM) \to (NM')}$$

We denote by $\to^*$ the reflexive and transitive clousure of $\to$.

Some λ-terms have more than one redex, and a question arises when it comes to choosing the first redex to be reduced. To address that issue one uses the notion of reduction strategy. The most common strategies used in this situation are:

- Normal-order reduction: Choose the leftmost outermost redex first.
- Applicative-order reduction: Choose the rightmost innermost redex first.
- Call-by-need: Variation of the normal order reduction where we choose the leftmost redex first and never evaluate an argument more than once. Note that the evaluation only occurs when arguments are required.

Associated to the notion of reduction, we define the notion of reduction graph [29].

Definition 6. *The reduction graph of a term M, denoted by $G(M)$ is the set $\{N | M \to^* N\}$ directed by $\to$.* $\Diamond$

Here we present a brief example to give the reader a better understanding of reduction graphs.

Example 6. In Figure 2.1 we present the reduction graph of the λ-term $MIMI$ where $M \equiv \lambda x.xI(xI)$ and $I = \lambda x.x$. ◄

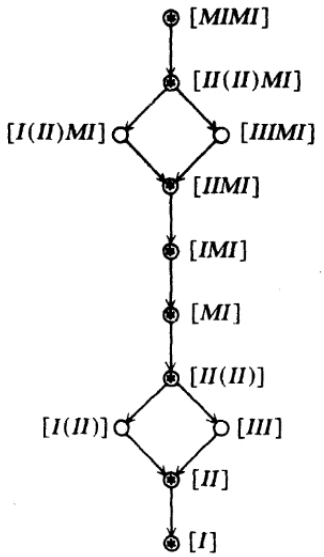


Figure 2.1: Reduction Graph of $MIMI$, taken from [21]

When an expression does not have a redex it is said to be in *normal form.*

One important characteristic of different reduction strategies is related to the termination of the evaluation of a given term.

Example 7. Some terms do not terminate using any of the above reduction strategies. For example,

$$(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \ldots$$

Some terminate under normal-order reduction, but not under applicative-order. For example, this term has two redexes.:

$$(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$$

If we choose the rightmost redex (applicative-order), it will not terminate. But if we choose the leftmost one, it will reduce in one step to the expression y.

Most importantly, if a term has a normal form, i.e., can be reduced to the point of having no redexes, that is using the normal-order reduction will always result in finding the normal form of the term [5]. The same does not apply to the applicative-order reduction and other strategies. ◀

2.2 Resource Aware Calculi

We briefly discuss resource aware calculi. We start by linearity given its relevance in resource aware computations.

2.2.1 Linearity

We will now address the problem of linearity in the λ-calculus and divide this section in two main parts: Linear Calculi and Annotated Calculi.

Regarding Linear Calculi, we will start by introducing the notion of linear and the affine λ-calculus, then we will talk about some linear extensions to some λ-calculus systems. After that, we will introduce Kfoury's [20] notion of linearization and some definitions that arise from that notion. We will finish this section by presenting the notion of weak linear λ-calculus.

2.2.1.1 Linear and Affine λ-calculus

In the linear λ-calculus the arguments are used exactly once. On the other hand, in the affine λ-calculus, arguments are used at most one time.

Linearity in logic is modelled by the widely used linear logic [11].

Linear logic is a substructural logic proposed by Jean-Yves Girard [11]. Linear logic is resource aware, in the sense that assumptions may not be arbitrarily copied nor erased.

There are three notions of linearity, based on linear logic, for functional calculi: *syntactical*, *operational* and *denotational*.

Syntactical linearity requires a linear use of variables in terms. Operational linearity means that arguments of functions cannot be duplicated or erased during evaluation. Finally, denotational linearity is achieved when all functions that can be defined in the language correspond to linear functions in a particular model.

In this thesis we are going to focus mainly on syntactical and operational linearity. We are going to describe in more detail those two types of linearity.

Syntactical linearity can be statically checked. On the other hand, operational linearity requires dynamic evaluation of terms. For the linear λ-calculus, syntactical linearity implies operational linearity.

Along with linear λ-calculus comes the definition of the affine λ-calculus, where the main difference is that a resource can be used at most once, so one can see affine linear logic, and therefore the affine λ-calculus, as linear logic with a weakening rule [27].

The weakening rule allows adding extra assumptions to a list of assumptions Γ that already proves a conclusion C without changing the actual validity of the original conclusion.

2.2.1.2 Linear Extensions

The λ-calculus is usually referred to as a type-free theory, since every expression can be applied to every other expression.

Type theories were first studied in the early 1900's with the purpose of helping surpass the mathematical paradoxes of the time, however they became a vary handy tool in proof-theory. Around the 1970's, the need for stronger programming languages forced computer scientists to study them as well, so they could build type theory based programming languages, such as ML [22].

Several techniques that emerged from this study were very relevant in making programming languages and the systems they represent much more robust. Such techniques include, as an example, type-checking algorithms [15] and type inference algorithms [14], that check if a given type can be assigned to a λ-term and automatically infer the type of a given term.

There are also typed versions of the λ-calculus [16]. Types are usually objects of a syntactic nature and may be assigned to expressions in the λ-calculus.

If M is a term, and A is a type assigned to M, then we can say "M has type A" or "M" in A. The notion used is $M : A$.

Example of typed extension of the λ-calculus include *Gödel's system* $\mathcal{T}$ [12] for bounded recursions and the *Plotkin's Programming Language for Computable Functions* (PCF) [25] for unbounded recursions.

System $\mathcal{T}$ was built from the simply typed λ-calculus, with the addition of number, booleans and a bounded recursor. Many linear versions of System $\mathcal{T}$ were studied in order to characterize classes of functions [2, 17].

PCF [25] is a Turing complete extension of the simply typed λ-calculus with an addiction of numbers, constants *pred*, *succ*, *iszero* a conditional operator and a fixpoint operator and forms the basis of programming languages such as ML [13] or Haskell [18]. In PCF typed are used to ensure the correct behavior of the program. Linear version of PCF were also defined [3].

2.2.1.3 Expanded λ-calculus

Kfoury [20] introduced for the first time the notion of linearization as a process of transforming non-linear functions into equivalent linear ones. Kfoury defined an expanded version of the λ-calculus with a new reduction $\rightarrow \beta$ in which the evaluation of these new linear λ-terms satisfies a particular case of what we here call the linearity condition:

Definition 7. *If the formal parameter x of a abstraction $(\lambda x.M)$ is not dummy, then the free occurrences of x in the body M of the abstraction are in a one-one correspondence with the arguments to which the function is applied.* $\Diamond$

Kfoury [20] also defined two very important notions in the linear λ-calculus: *contraction*, denoted as $|M|$, being M a λ-term and *lifting*. Contracting an expanded term in the new calculus yields a λ-term. Lifting a β-reduction generates a $\beta\hat{}$-reduction, which consumes arguments at most once. Well-formed terms of the new calculus are those for which there is a contracted term in the λ-calculus.

Theorem 1. *Let M be a standard λ-term. If there is a well-formed expanded λ-term N such that $|N| \equiv M$ and every β-reduction from M can be lifted to a $\beta\hat{}$-reduction from N, then M is β-strongly normalizable.* ■

Kfoury then concluded that for any strongly normalized λ-term M, there exists a term in $\Lambda\hat{}$, which contraction is M.

This work shown that there is a semantic between the standard λ-calculus and the linear λ-calculus. This was further studied in [10].

2.2.1.4 Weak-Linear λ-calculus

Using a linear subset of the λ-calculus, the question of whether one can or can not simulate the λ-calculus arises. Such subset is called the *weak linear λ-calculus* and the following definition arises, according to [4]:

Definition 8. *A λ-term M is called weak linear if and only if every redex $(\lambda x.N)S$, in the reduction graph of M, is such that x occurs at most once in N.* ◊

The weak linear λ-calculus is:

1. Strongly normalizing.
2. Typable in polynomial time.
3. Decidable in polynomial time.

The weak linear λ-calculus is *strongly normalizing* meaning that no terms are duplicated by reduction, then each reduction effectively reduces the size of the term, therefore, there cannot be infinite reductions. It is *typable in polynomial type* because it uses a restricted type system that is based on interaction types. Also, it is *decidable in polynomial time* because it uses a maximal reduction strategy and knowing that every weak linear term is normalized in a number of steps less or equal to its size.

2.2.2 Linear Types

In the next sections, we will talk briefly about Linear Type Systems.

Regarding Linear Type Systems, we will briefly explain their core characteristics. We briefly mention Linear Haskell, has as a reference for a practical use of Linear Types.

2.2.2.1 Linear Type Systems

Linear type systems [28] are based on Linear Logic introduced by Girard [11] and explicitly control the use of resources through the use of types.

A term of linear type $A \multimap B$ is a function from A to B that neither duplicates or ignores its argument.

The impossibility of duplication helps to guarantee an efficient implementation, in the sense that it is safe to update data structures destructively, like, for instance, overwriting a given index of an array with the given value.

Equally important is the impossibility of resource destruction, which means a resource must be used eventually. It avoids the need for space recovering through the use of, for example, a garbage collector, meaning that both allocation and deallocation of linear values are implicit in the program text [28].

Linear type systems start to appear now into most known programming languages. In [6], the functional language Haskell is extended for linear types, enabling safe update-in-place for mutable structures and access protocols for external API's.

2.2.3 Cost Annotated Operational Semantics

We will now address annotated operational semantics and lastly talk about two different systems based on resource calculi.

Annotated operational semantics are used by type systems to enhance the cost estimation associated to a given computation. Costs are measured in various forms, such as memory costs, time costs and number of steps.

Steffen Jost *et. al* [19] stated that predicting operational properties, such as time and space consumption can pose some difficulties. These difficulties are even more significant in non-strict programming languages such as Haskell [26]. Annotated operational semantics, as defined in [19] is presented as $M \rightarrow^n N$, where n is a non-negative integer that represents the cost associated with the reduction of M to N [19].

In [19] a type-based approach is described for obtaining static cost bounds for lazily evaluated functional programs. The analysis that they use, combines two independent analysis that considered the allocation of the costs of recursive and co-recursive programs.

This work also provides an operational semantics using annotated typing in order to generalize the cost model to a parametric one.

2.2.4 Resource Calculi

The explicit use of resources into the calculi was made by the definition of resource aware calculi such as [24] and [8].

We will now address two different approaches of resource calculus.

In one hand, we have an approach conceived by Gerard Boudol [7], whose formulation consists of a refinement of the λ-calculus, where one has possibly infinite resources.

In order to achieve that, the notion of *bag of resources* was introduced as the consumable argument of a function. Those bags consist of multisets of terms with associated multiplicities that indicate the number of available copies of such resource. It converges to the regular λ-calculus when all bags have a multiplicity of 1.

Bags are represented as a parallel composition $P = (M_1^{m_1} | \ldots | M_k^{m_k})$ where m_i represents the multiplicity associated to the term M_i.

A more detailed explanation on such formalism and its details is available in Chapter 3.

On the other hand, we have a different approach. Ronchi della Roca *et. all* [24] came up with a calculus that is an extension of the λ-calculus allowing to model resource computation. In this type of calculus, the argument of a function comes as a finite multiset of resources, which in turn can be either linear or reusable. Because of that, this calculus is non-deterministic.

In Chapters 3 and 4 we will cover in more depth both of these approaches, due to their high influence in the outcome of our work.

In this section we introduced some basic concepts on classic λ-calculus, then we mentioned resource aware calculi and their linear expansions. Then, we proceeded to talk about Linear type systems. Next, we talked about annotated calculi and finally we briefly addressed resource calculi.

In the next section we will focus on λ-calculus with multiplicities and resource calculi. We will expose both approaches in more detail and also present our implementation in Haskell of them.

Chapter 3

λ-calculus with multiplicities

The λ-calculus with multiplicities is a refinement of the usual λ-calculus, inspired by the encoding of the lazy λ-calculus into the π-calculus given by Milner [23]. The basic observation is that in a reduction step $(\lambda x.M)N \to M[N/x]$, the argument N is copied as many times as we need, which means, as much as there are free occurences of x in M.

According to Boudol [8], the argument of a function is a bag of resources, that is, a multiset of terms. Then each term in the bag comes with an explicit, possibly infinite multiplicity, indicating how many copies of it are available. One recovers the usual λ-calculus when the bags consist of just one term. So we can say that a bag can be written as a parallel composition $P = (M_1^{m_1} \mid \cdots \mid M_k^{m_k})$ of terms with multiplicities, where m_i is an integer greater or equal than zero, or ∞. The parallel composition is intended to be commutative and associative, with $\mathbf{1}$ as a neutral element.

Then, besides the variables $x, y, z, \ldots$ and the abstraction $\lambda x.M$, the syntax of this calculus includes applications of the form (MP), where M is any term, and P a bag of terms. The management of the resources is done by means of explicit substitutions [1]. The syntax of the λ-calculus with multiplicities is as follows:

$$M ::= x \mid \lambda x.M \mid (MP) \mid (M[P/x])$$
$$P ::= 1 \mid M \mid (P|P) \mid M^{\infty}$$

The set of terms will be denoted by Λ^M and the set of bags by Λ^P. Note that $\Lambda^M \equiv \Lambda^P$.

With the syntax of terms presented above, we can define the commutativity and associativity properties in bags of terms as follows, where $\equiv$ stands for structural equivalence and M^{∞} stands for a term with infinite multiplicity:

$$(P|\mathbf{1}) \equiv P$$
$$(P|Q) \equiv (Q|P)$$
$$(P|(Q|R)) \equiv ((P|Q)|R)$$
$$M^{\infty} \equiv (M|M^{\infty})$$

We can now define terms with explicit finite multiplicity:

$$M^0 = 1$$
$$M^{m+1} = (M|M^m)$$

We will now define evaluation rules that rely on a lazy evaluation mechanism. It means that neither the body M of an abstraction $\lambda x.M$ or the arguments of an application MP or in a substitution $M[P/x]$, namely P, are evaluated.

The one-step evaluation relation is denoted $M \to M'$. We use implicitly the rule that any two α-equivalent (one can transform one term into another through α-conversion) terms have the same reductions. The notion of α-conversion is the same that was introduced for λ-calculus. The rule for one step evaluation is presented in Rule 3.1:

$$\frac{M \to M'}{N \to M'} \; M =_{\alpha} N \tag{3.1}$$

We will now start by defining Π as the set of "bags of resources", Σ as the set of substitution items and Λ^m as the set of terms defined by the first clause presented in the syntax of λ-calculus, that is the terms of the form $[P/x]$ for $P \in \Pi$.

We are going to show an example of rule 3.1 to give to the reader a better understanding.

Example 8. Let M be $(\lambda x \lambda y.xy)(\lambda z.z)(\lambda w.w)$ we obtain by several $\to^{\beta}$ the following expression:

$M = (\lambda x \lambda y.xy)(\lambda z.z)(\lambda w.w) \to^{\beta} (\lambda y.(\lambda z.z)y)(\lambda w.w) \to^{\beta} (\lambda z.z)(\lambda w.w) \to^{\beta} \lambda w.w.$

And let N be $(\lambda w \lambda z.wz)(\lambda x.x)(\lambda y.y)$, by the same reductions, we obtain the following expression

$N = (\lambda w \lambda z.wz)(\lambda x.x)(\lambda y.y) \to^{\beta} (\lambda z.(\lambda x.x)z)(\lambda y.y) \to^{\beta} (\lambda x.x)(\lambda y.y) \to^{\beta} \lambda y.y.$

We observe that M and N, are α-equivalent where $\lambda x.x =_{\alpha} \lambda w.w$, $\lambda y.y =_{\alpha} \lambda z.z$, $\lambda z.z =_{\alpha} \lambda x.x$ and $\lambda w.w =_{\alpha} \lambda y.y$.

Boudol [8] observes that any term M may be written in a unique way as $AQ_1 \dots Q_k$ where each of the Q_i belongs to $\Pi \cup \Sigma$ and A is either a variable, or an abstraction, $\lambda x.N$. The grammar for the terms is written as follows:

$$M ::= A \mid (MQ)$$
$$A ::= x \mid \lambda x.M$$
$$Q ::= P \mid [P/x]$$

Where P is any term of Π. To evaluate any encoded function, resource calculi uses also the notion of value, which is any functional closure. This means that the values are the terms of Λ^m given by the grammar:

$$V ::= \lambda x.M \mid (V[P/x])$$

Now we are introducing the computation rules. The two rules presented next state that a reduction may be performed in the context of list of arguments or substitutions:

$$\frac{M \to M'}{MP \to M'P} \quad (3.2)$$

$$\frac{M \to M'}{M[P/x] \to M'[P/x]} \quad (3.3)$$

Then the actual computation depends on the form of the head subterm A of $M = AQ_1 \dots Q_k$. So there are two possible types for A: A is an abstraction or a variable. When A is an abstraction, we look for the first Q_i in the list, if any, which is an argument, that is a term P of Π to which the closure $AQ_1 \dots Q_{i-1}$ is applied. Using the context rules (rules 3.2 and 3.3), we perform a β-reduction of the form:

$$(\lambda x.M)[P_1/x_1] \cdots [P_{i-1}/x_{i-1}]PQ_{i+1} \cdots Q_k \to M[P/x][P_1/x_1] \cdots [P_{i-1}/x_{i-1}]Q_{i+1} \cdots Q_k \quad (3.4)$$

provided the $x's$ are not free in P. There are two rules that formalized what was mention before.

$$(\lambda x.M)P \to M[P/x] \quad (3.5)$$

$$\frac{(VP) \to M}{(V[R/x])P \to M[R/x]} \; x \notin \mathrm{fv}(P) \quad (3.6)$$

The function $fv(P)$ is defined similarly as for terms. A variable x occurs free in a bag P, if in one of the terms that compose the bag, x occurs free in M.

The head subterm can also be a variable x, and when it is the case, one looks for the first substitution $[P/x]$ for it, if any, in the list $Q_1 \dots Q_k$. Then one fetches a resource out of P, that is any term N of Λ^m such that $P \equiv (N|R)$, and leaves the rest R for future use. To state the *fetch* rule 3.10, we introduce an auxiliary relation $M[N/x] \to M'$, intended to formalize the

replacement of the head variable x of M by N, that is, $M = xQ_1 \dots Q_k$ and $M' = NQ_1 \dots Q_k$, where $Q_i \in \Pi \cup \Sigma$. The rules are:

$$\mathrm{x[M/x]} \to M \quad (3.7)$$

$$\frac{M[N/x] \to M'}{(MP)[N/x] \to M'P} \quad (3.8)$$

$$\frac{M[N/x] \to M'}{(M[P/z])[N/x] \to M'[P/z])} \; z \neq x \text{ and } z \notin \mathrm{fv}(N) \quad (3.9)$$

And the *fetch* rule is given by:

$$\frac{M[N/x] \to M'}{M[P/x] \to M'[R/x]} \; P \equiv (N|R) \; x \notin \mathrm{fv}(N) \quad (3.10)$$

But notice that in this rule the resource that is fetched is a term of Λ^m, and not a term of Π. It is important to emphasize two points. First, if the bag P in $M[P/x]$ is empty, nothing can be fetched out of it. For example, a term like $x[\mathbf{1}/x]$ is *deadlocked*. It is a closed normal form but not a value. Second since parallel composition is commutative and associative, any resource from the bag can be selected in the fetch operation. Then we can define a non-deterministic choice $(M \oplus N)$ as follows, provided x is not free in M or N:

$$(\mathrm{M} \oplus N) =_{def} x[(M|N)/x] \quad (3.11)$$

We have now introduced related work done by Boudol [8] that provides important concepts on λ-calculus with multiplicities, which will play a great role in our work, in the sense that part of it consists in studying this calculus.

3.1 Implementation

We are now going to present the relevant parts of our implementation of the calculus with multiplicities.

Following the paper of Gérald Boudol [8], we start by implementing the syntax of λ-calculus with multiplicities, in Haskell, and some auxiliary functions.

The two next coding boards will give the reader our Haskell implementation of this grammar, starting with the Bag implementation and then the Term implementation.

```
data Bag =  Empty
         | TSingle Term
         | TInfinite Term
```

```
        | Par Bag Bag
        deriving (Show, Eq)
```

The above code board represents the data in Haskell that represents the Bag data structure, according to the λ-calculus with multiplicities [7]. Note that we a Term can be represented as a Bag, either as a *TSingle* or a *TFinite*. Those different representations allow us to represent terms with infinite, and finite multiplicities. Also, the *Empty* token represents the empty Bag and the *Par* constructor represents the $(P|Q)$ operation.

Below, we have the data for a term. It can be a variable, a λ-term, an application of a term to a bag, or a substitution of a variable in a term by a bag.

```
data Term = V String
          | Lambda String Term
          | App Term Bag
          | Subs Term Bag String
          deriving (Show, Eq)
```

The representation for a Term is really straightforward. The V constructor represents a variable, the *Lambda* constructor represents a λ-term, the *App* constructor represents an application of a *Term* to a *Bag* and the *Subs* constructor represents a *Term* where we substitute a variable represented as a *String* by a *Bag*.

The next step was to create a function to check values

$$V ::= \lambda x.M \mid (V[P/x])$$

that are terms of Λ^m, and the function for parallel composition,

$$\begin{aligned}
&(P|\mathbf{1}) \equiv P \\
&(P|Q) \equiv (Q|P) \\
&(P|(Q|R)) \equiv ((P|Q)|R) \\
&M^\infty \equiv (M|M^\infty)
\end{aligned}$$

both previously defined in the beginning of Chapter 3 and are presented below, respectively:

```
value :: Term -> Bool

value (Lambda m x) = True
value (Subs v b x) = value v
value _ = False
```

The *value* function determines if a term is either a λ-term or a substitution on a value.

```
equals :: Bag -> Bag

equals (Par p Empty) = p
equals (Par p q) = Par q p
equals (Par p (Par q r)) = Par (Par p q) r
equals (TInfinite m) = Par (TSingle m) (TInfinite m)
```

The *equals* function represents the four cases of the parallel composition. The first case represents the parallel composition $P|1$, that evaluates to P. The second case represents the equivalence between the parallel composition $(P|Q)$ and $(Q|P)$. Next, we have the equivalence between $(P|(Q|R))$ and $((P|Q)|R)$. Finally, we have the unfolding of a term with infinite multiplicity, where $M^{\infty} \equiv (M|M^{\infty})$.

We also implemented functions for substitution and reduction. They follow the structure of Boudol [8], mentioned in Chapter 2. For reduction we decided to create a single function with the rules 3.2, 3.3, 3.5 and 3.6 following Boudol's ideas.

```
red :: Term -> Term

red (App (Lambda x  m) p) = Subs m p x
red (App (Subs v r x) p) = Subs (red (App v p)) r x
red (App m p) = App (red m) p
red (Subs m p x) = Subs (red m) p x
```

We will now present some reduction examples:

test11: $(\lambda x.x)(\lambda y.y)$

test12: $(\lambda z.z)(\lambda x.x(\lambda y.y))$

The translation of those examples to Haskell is presented below:

```
test11 :: Term
test11 = App (Lambda "x" (V "x")) (TSingle (Lambda "y" (V "y")))

test12 :: Term
test12 = App (Lambda "z" (V "z")) (TSingle test11)
```

The output of running **test11** in our system yields the following output:

```
[Subs (V "x") (TSingle (Lambda "y" (V "y"))) "x"]
```

This translates to λ-calculus with multiplicities as $x[\lambda y.y/x]$, which is, in fact, the expected result for the chosen example, in a one-step perspective.

Test12 results in:

```
[Subs (V "z") (TSingle (App (Lambda "x" (V "x")) (TSingle (Lambda "y" (V
    "y"))))) "z"]
```

This translates to λ-calculus as $z[(\lambda x.x)(\lambda y.y)/z]$, which is the expected result for a one-step reduction of the presented example.

We used the same method for substitution, but the rules that we developed were 3.7, 3.8 and 3.9.

```
sub :: Term -> Term

sub (Subs (V x) (TSingle m) y) |
  x == y = m
sub (Subs (V x) (TInfinite m) y) |
  x == y  = m
sub (Subs (App m p) n x) = App (sub (Subs m n x)) p
sub (Subs (Subs m p z) n x) = Subs (sub (Subs m n z)) p z
```

We will now present some substitution examples:

The λ-calculus representation of the examples is: **test21** $= x[\lambda y.y/x]$, **test24** $= \lambda x.x[y/x]$ and **test22** $= x[(\lambda x.x)(\lambda y.y)/x]$

```
test21 :: Term
test21 = Subs (V "x") (TSingle (Lambda "y" (V "y"))) "x"

test24 :: Term
test24 = Subs (Lambda "x" (V "x")) (TSingle(V "y")) "x"

test22 :: Term
test22 = Subs (V "x") (TSingle (App (Lambda "x" (V "x")) (TSingle (Lambda "y"
    (V "y"))))) "x"
```

The results of applying the substitution rules to those λ-terms is: **test21** $= (\lambda y.y)$, **test24** $= (\lambda x.x)$ and **test22** $= (\lambda x.x)(\lambda y.y)$.

Such terms are, in fact, the output of our system, when applying the substitution rules to those examples, as presented below:

```
test21 = [Lambda "y" (V "y")]

test24 = [Lambda "x" (V "x")]

test22 = [App (Lambda "x" (V "x")) (TSingle (Lambda "y" (V "y")))]
```

Finally, we created the *fetch* rule 3.10:

```
fetch :: Term -> Term
```

```
fetch (Subs m (Par n r) x) = Subs (sub (Subs m n x)) r x
```

We will present some examples of applying the fetch rule.

We tested the fetch rule on the following examples: **test31** $= (\lambda x.x)[(1|(\lambda y.y))/x]$ and **test32** $= (\lambda x.x)[(1|1|\lambda y.y)/x]$. We present their representation in our system below:

```
test31 :: Term
test31 = Subs (Lambda "x" (V "x")) (Par Empty (TSingle (Lambda "y" (V "y"))))
    "x"

test32 :: Term
test32 = Subs (Lambda "x" (V "x")) (Par Empty (Par Empty (TSingle $ (Lambda "y"
    (V "y"))))) "x"
```

The expected result of applying the fetch rule to those examples is: test31 $= \lambda x.x[(\lambda y.y)/x]$ and test32 $= \lambda x.x[(1|(\lambda y.y))/x]$.

This corresponds to the result of applying the fetch rule to the same examples in our system, as stated below:

```
test31 = [Subs (Lambda "x" (V "x")) (TSingle (Lambda "y" (V "y"))) "x"]

test32 = [Subs (Lambda "x" (V "x")) (Par Empty (TSingle (Lambda "y" (V "y"))))
    "x"]
```

Note that the choice of which resource to choose in the fetch rule is deterministic in the smaller step of fetch, in the sense that the algorithm always chooses the first resource available. We also implemented a *fetch_all* function that outputs a list of all possible choices the algorithm can, in fact, make. The code in Haskell for the *fetch_all* routine is presented below.

```
fetch_list :: [Bag] -> Term -> String -> [Term]
fetch_list [] m s = []
fetch_list (x:xs) m s = fetch (Subs m x s) ++ fetch_list xs m s

fetch_all :: Term -> [Term]
fetch_all (Subs m n x) = fetch_list (groupTerms $ all_terms n) m x
```

An example using the *fetch_all* routine is presented below.

```
*Main> fetch_all test32
[Subs (Lambda "x" (V "x")) (Par Empty (TSingle (Lambda "y" (V "y")))) "x",Subs
    (Lambda "x" (V "x")) (Par Empty (TSingle (Lambda "y" (V "y")))) "x",Subs
    (Lambda "x" (V "x")) (Par Empty Empty) "x",Subs (Lambda "x" (V "x")) (Par
    (TSingle (Lambda "y" (V "y"))) Empty) "x",Subs (Lambda "x" (V "x")) (Par
    Empty Empty) "x",Subs (Lambda "x" (V "x")) (Par (TSingle (Lambda "y" (V
    "y"))) Empty) "x"]
```

One can easily see that all possibilities are accounted in the output of our algorithm.

Implementing these λ-calculus rules in a functional language gave us a lot more insight about our work.

Chapter 4

Resource λ-calculus

The resource calculus [24] extends λ-calculus to model resource consumption. In this extension, arguments of a function are defined as finite multisets of resources. Those resources can either be linear, in the sense they must be used exactly once, or reusable, meaning that can be used *ad libitum*.

Evaluating a function applied to an argument yields different possible choices, due to the amount of different possible distributions of resources among the occurrences of the formal parameter, because the argument of a function is a multiset of resources. This specificity of the resource calculus generates a non-deterministic result, that is represented by a formal sum of all the possible cases.

This non-determinism introduces a notion of failure of computation different from non-termination: the empty sum. Whenever the number of available resources does not fit the number of occurrences of the variable abstracted in a redex, it evaluates to the empty sum or *crash*. The resource calculus is specially important in studying the relation between the notions of linearity and non-determinism [24].

This calculus can be seen as an evolution of Boudol's calculus of multiplicities, where the main differences between the two calculi lie on the fact that Boudol's calculus uses explicit substitutions and lazy operational semantics, while the resource calculus extends the classical λ-calculus. If one translates the application MN into $M[N^!]$ where $[N^!]$ represents the multiset containing one reusable copy of the resource N, then classic λ-calculus can, in fact, be embedded into Λ^r.

The resource calculus has three syntactical sorts:

- Terms, that are in functional position, represented by Λ^r;
- Bags, that represent the multiset of resources, and are represented by Λ^b;
- Finite formal sums, that represent the possible results of a computation, that are represented by $\Lambda^{(!)}$.

Recall that a resource can be linear or used *ad libitum* and bags are multisets presented in multiplicative notion. So $P \cdot Q$ is the multiset union, and $1 = []$ is the empty bag, meaning that $P \cdot 1 = P$ and $P \cdot Q = Q \cdot P$.

Sums are multisets with additive notation with the empty multiset represented by a 0. So $M + 0 = M$ and $M + N = N + M$.

An expression, denoted by $\Lambda^{(b)}$ is either a term or a bag.

$$\begin{array}{ll}
\Lambda^r : & M, N, L ::= x \mid \lambda x.M \mid MP \qquad \text{Terms} \\
\Lambda^{(!)} : & M^{(!)}, N^{(!)} ::= M \mid M^! \qquad \text{Resources} \\
\Lambda^b : & P, Q, R ::= 1 \mid [M^{(!)}] \mid P.Q \qquad \text{Bags} \\
\Lambda^{(b)} : & A, B ::= M \mid P \qquad \text{Expressions} \\
Nat\langle\Lambda^r\rangle : & \mathbb{M}, \mathbb{N} ::= 0 \mid M \mid \mathbb{M} + \mathbb{N} \qquad \text{Sums of Terms} \\
Nat\langle\Lambda^b\rangle : & \mathbb{P}, \mathbb{Q} ::= 0 \mid P \mid \mathbb{P} + \mathbb{Q} \qquad \text{Sums of Bags}
\end{array}$$

When we mention $[x, \lambda x.x, x^!]$ we will say $[x].[\lambda x.x].[x^!]$.

In this calculus, we have present both linear and reusable resources. Hence, the need for two different notions of substitution arises:

Definition 9. *We define both substitution operations as follows:*

1. *$A\{N/x\}$ is the classic λ-calculus substitution of N for x. It is extended to sums as in $\mathbb{A}\{\mathbb{N}/x\}$ by linearity in $\mathbb{A}$ and using the formal equalities below. The form $A\{x + N/x\}$ is called **partial substitution**.*

$$\begin{gathered}
\lambda x.(\textstyle\sum_i M_i) = \sum_i \lambda x.M_i \\
(\textstyle\sum_i M_i)P = \sum_i M_i P \\
M(\textstyle\sum_i P_i) = \sum_i M P_i \\
[(\textstyle\sum_i M_i)] \cdot P = \sum_i [M_i] \cdot P \\
[(\textstyle\sum_i M_i)^!] \cdot P = [M_1^!, \ldots, M_k^!] \cdot P
\end{gathered}$$

2. *$A\langle N/x\rangle$ is the **linear substitution** defined by:*

$$\begin{gathered}
y\langle N/x\rangle = \begin{cases} N \ \textit{if } x = y, \\ 0 \ \textit{otherwise} \end{cases} \\
[M]\langle N/x\rangle = [M\langle N/x\rangle] \\
[M^!]\langle N/x\rangle = [M\langle N/x\rangle, M^!] \\
(\lambda y.M)\langle N/x\rangle = \lambda y.(M\langle N/x\rangle) \\
(MP)\langle N/x\rangle = M\langle N/x\rangle P + M(P\langle N/x\rangle) \\
1\langle N/x\rangle = 0 \\
(P \cdot R)\langle N/x\rangle = P\langle N/x\rangle \cdot R + P \cdot R\langle N/x\rangle.
\end{gathered}$$

It is extended to $\mathbb{A}\{\mathbb{N}/x\}$ *by linearity in both* $\mathbb{A}$ *and* $\mathbb{N}$.

$\Diamond$

Linear substitution corresponds to the placement of the resource in exactly one linear occurrence of the variable. When one has multiple possibilities, all the choices are performed and the end result is the sum of all of them. We will now elaborate with an example: $(y[x][x])\langle N/x\rangle = y[N][x] + y[x][N]$. If there are no free linear occurrences, then linear substitution returns 0. For example $(\lambda y.y)\langle N/x\rangle = \lambda y.(y\langle N/x\rangle) = \lambda y.0 = 0$.

We will now present two different types of reduction steps [24]: the baby-step reduction and the giant-step reduction.

The two differ in the way they perform the substitutions when reducing redexes. While the baby step performs one substitution at a time, the giant-step consumes all the redex at once.

Definition 10. *The* ***baby-step*** *reduction is denoted by* $\rightarrow^b$ *and is defined by the following relation, assuming that* $x \notin fv(N)$*:*

1. $(\lambda x.M)1 \rightarrow^b M\{0/x\}$
2. $(\lambda x.M)[N].P \rightarrow^b (\lambda x.M\langle N/x\rangle)P$
3. $(\lambda x.M)[N^!].P \rightarrow^b (\lambda x.M\{N + x/x\})P$

The ***giant-step*** *reduction is denoted by* $\rightarrow^g$ *and is defined by the following relation for* $l, n \geq 0$*, assuming that* $x \notin fv(L_i)$ *and* $fv(N_i)$*:*

1. $(\lambda x.M)[L_i, \ldots, L_l, N_1^!, \ldots, N_n^!] \rightarrow^g M\langle L_1/x\rangle \ldots \langle L_l/x\rangle\{N_1 + \cdots + N_n/x\}$.

If $n = 0$*, then the reduct is* $M\langle L_1/x\rangle \ldots \langle L_l/x\rangle\{0/x\}$*. For every reduction, we will denote by* $\rightarrow^{x+}$ *its transitive closure and by* $\rightarrow^{x*}$ *its reflexive-transitive closure.*

Baby-step and giant-step reductions are clearly related. But notice that the giant-step reduction is defined independently of the ordering of the resource substitutions.

Let $\Delta = \lambda x.x[x^!]$ and $I = \lambda x.x$:

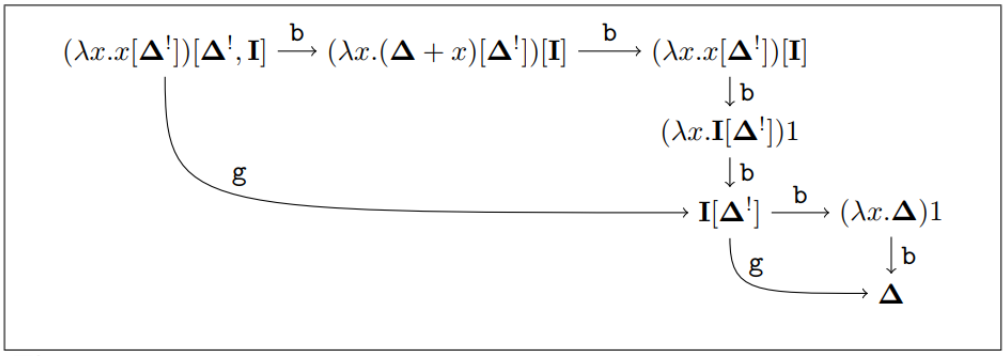


Figure 4.1: Example of baby and giant step reductions, taken from [24]

Figure 4.1 shows sequences of both baby-step and giant-step reductions. In [24] it is shown that both reduction strategies are related, in the sense that $\rightarrow^g \subset \rightarrow^{b*} \subset \rightarrow^{g*} \leftarrow^{g*}$, where the last denotes the relational composition between $\rightarrow^{g*}$ and its inverse $\leftarrow^{g*}$.

One should note that, even though both reduction strategies yield the same normal forms, they have different properties: for example, in Figure 4.1, the starting term is strongly normalizing for giant-step reduction but weak normalizing for baby-step reduction.

Definition 11. *Weak normalization means that any term has a terminating rewriting sequence, i.e. admits a finite amount of reduction steps which lead to a normal form.*

Definition 12. *Strong normalization means that any term can never be reduced infinitely many times. Any reduction sequence eventually reaches a normal form (hence it is finite).*

In the next section we will present some parts of our implementation for this strategy.

4.1 Implementation

We will now present the relevant parts of our implementation.

We start by implementing the sintax of the resource calculus, in Haskell, with the help of some auxiliary functions.

Recall the grammar presented in the beginning of this chapter.

The next coding board will describe our implementation of the Terms in this grammar.

```
data Rterms = EmptyT
            | Var String
            | Lambda String Rterms
            | App Rterms Rbags
            deriving (Show)
```

The above code board shows the different forms a Term can assume in the resource calculus. It can either be the empty term, represented as *EmptyT*, a variable, represented by the *Var* constructor, a λ-term, represented by the *Lambda* constructor, or an application of a term to a bag, represented by the *App* constructor.

Now we present the implementation of the Bag and Resource data structures.

```
data Resource = Term Rterms
              | BangTerm Rterms
              deriving Show
```

```
data Rbags = Empty
           | TSingle Rterms
           | Dot Rterms Rbags
           | Multi [Resource]
           deriving Show
```

A Resource can either be a term, as represented by the *RTerms* data, or a term with infinite multiplicity, represented by the *BangTerm* constructor.

A Bag, on the other hand, can be an empty bag (*Empty* constructor), a term (*TSingle* constructor), a $P \cdot Q$ (*Dot* constructor) or a list with resources (*Multi* constructor).

Next, we present our implementation of an Expression in resource calculus.

```
data Exp = ExpTerm Rterms
         | ExpBag Rbags
         deriving Show
```

An Expression can either be a term, as represented by the *ExpTerm* constructor, or a bag, as represented by the *ExpBag* constructor.

Finally, we present the data type for sums. We decided to treat sums of terms and sums of bags individually, as suggested in the grammar presented above.

```
data SumTerms = ZeroT
              | SumTerm Rterms
              | LambdaST String SumTerms
              | AppST SumTerms Rbags
              | AddT SumTerms SumTerms
              deriving Show
```

```
data SumBags = ZeroB
             | SumBag Rbags
             | AppSB SumTerms SumBags
             | DotSB SumBags SumBags
```

```
              | AddB SumBags SumBags
              | BangTermsSB SumTerms
              | MultiSB [SumBags]
              deriving Show
```

In these data structures, *SumTerms* and *SumBags*, their structure is pretty straightforward when compared to the grammar. They both have a zero, *ZeroT* and *ZeroB* respectively, or they are a term, in the case of *SumTerms*, or a bag, in the case of *SumBags*, or a sum of those.

```
data SumExp = Union SumTerms SumBags
              deriving Show
```

Having finished presenting the data structures implemented by us, we will now introduce some functions that represent some important operations in the resource calculus.

First, we will present the substitution implementation.

```
subR :: Rterms -> Rterms -> String -> Maybe Rterms

subR (Var y) n x
  | y == x = Just n
  | otherwise = Nothing

subR (Lambda y m) n x = Just $ Lambda y unpacked
                  where packed = subR m n x
                        unpacked = unpackTerms packed

subB :: Rbags  -> Rterms -> String -> Maybe Rbags

subB (Multi [Term m]) n x = Just $ Multi [Term (unpackTerms $ subR m n x)]

subB Empty _ _ = Nothing

subB (Multi [BangTerm m]) n x = Just $ Multi [Term (unpackTerms $ subR m n x),
    BangTerm m]

subR1 :: Rterms -> Rterms -> String -> SumTerms
subR1 (App m p) n x = AddT ( SumTerm ( App (unpackTerms $ subR m n x) p )) (
    SumTerm ( App  m (unpack $ subB p n x)))

subR2 :: Rbags -> Rterms -> String -> SumBags
subR2 (Dot p r) n x = AddB (SumBag (Dot (unpackTerms $ subR p n x) r)) (SumBag
    (Dot p (unpack $ subB r n x)))

subN :: Rterms -> Rterms -> String -> Rterms

subN (Var x) m y |
  x == y = m
```

```
subN (App m p) n x = App (subN m n x) p

subN (Lambda x t) n y = if elem y (freev (Lambda x t) [] ) then Lambda x (subN
    t n y) else Lambda x t
```

The *subR* function represents the substitution of a variable or a λ-term in Definition 9. The *subB* function represents the substitution rules applied to lists of resources in Λ^b, where one can either have a single term, or a term with infinite multiplicity. Both of those cases are considered, and our implementation follows the formulation previously presented in Definition 9. The *subR1* rule follows the formulation in Definition 9 where the substitution of an application will result in a sum of new terms. The *subR2* function represents the substitution of a $P.Q$ bag by a term t. The *subN* function represents the classical substitution in λ-calculus.

We will now show some examples to demonstrate how our implementation of the substitution function works:

We test the substitution function in some examples like **test22:** $(\lambda y.y)$ where we need to add x because the function needs another argument, and **test23:** $(\lambda x.x)(\lambda y.y)$. We also have a basic example that is going to be call to the other tests, **text21:** x.

```
test21 :: Rterms
test21 = Var "x"

test22 :: Rterms
test22 = Lambda "y" (Var "y")

test23 :: Rterms
test23 = App (Lambda "x" (Var "x")) (TSingle (Lambda "y" (Var "y")))
```

We should expect that the outcome of **test22** with **test21** and x to be $(\lambda y.y)$, and we obtian,

```
*Main> subN test21 test22 "x"
Lambda "y" (Var "y")
```

And for the **test23** with **test23** and x we expect $(\lambda x.x)(\lambda y.y)$, and the outcome was,

```
*Main> subN test21 test23 "x"
App (Lambda "x" (Var "x")) (TSingle (Lambda "y" (Var "y")))
```

As expected we obtained the same result.

Finally, we introduce the giant-step reduction. We only present the implementation of the giant-step reduction. The baby-step reduction was also implemented as an experiment to assure the consistency of the results generated by the giant-step reduction, but it will not be used in our experimental results.

```
redg :: Rterms -> Rbags -> Rterms

redg (Lambda x t) (Multi n) = apply_subR (apply_subN t x l') x l''
                where (l',l'') = split n [] []

apply_subN :: Rterms -> String -> [Resource] -> Rterms
apply_subN t y [] = t
apply_subN t y (Term x:xs) = apply_subN (subN t x y) y xs

apply_subR :: Rterms -> String -> [Resource] -> Rterms
apply_subR t y [] = t
apply_subR t y (BangTerm x:xs) = apply_subR (unpackTerms $ subR t x y) y xs

split :: [Resource] -> [Resource] -> [Resource] -> ([Resource], [Resource])
split [] ls xs = (ls,xs)
split ((Term t):ys) ls xs = split ys (ls ++ [Term t]) xs
split (BangTerm t:ys) ls xs = split ys ls (xs ++ [BangTerm t])
```

The implementation of the giant step reduction step follows directly from Definition 10.

Some examples of reduction follow:

To explain better the reduction we present and example. Consider **test13** and **test14** $(\lambda x.x)$ and $[xy]$ respectively.

```
test13 :: Rterms
test13 = Lambda "x" (Var "x")

test14 :: Rbags
test14 =  Multi [ Term (App (Var "x") (TSingle(Var "y"))) ]
```

If we test them both together we obtain,

```
test13 :: Rterms
test13 = Lambda "x" (Var "x")
```

And the expected result was $\lambda x.x$. So we can say that the results are the same.

These implementations gave us a strong insight of the basic notions of the resource calculus and the calculus with multiplicities. A natural question is what is the relation between the two calculi. We further explore this question in the next chapter.

Chapter 5

Compilation

We previously presented two calculi: the λ-calculus with multiplicities [7] and the resource calculus [24] and in this chapter we are going to explore a relation between these two calculi. We will start by defining a compilation function of the resource calculus into the λ-calculus with multiplicities. This will give us a better insight on the relation between these calculi.

We will introduce an example of an application in both settings, to highlight the similarities between the two calculi.

In both settings it is possible to create the term $(\lambda x.x)(1)$. This can be proven using the grammar for each setting. In the calculus with multiplicities, we can form the term $(\lambda x.x)(1)$, following $M \to MP \to (\lambda x.x)P \to (\lambda x.x)(1)$. In a similar way, one can generate, through the grammar of the resource calculus, the term $(\lambda x.x)(1)$, following $M \to MP \to (\lambda x.x)P \to (\lambda x.x)(1)$.

With this example we can see a trace of similarity between the two calculi, which we will further discuss.

In the resource λ-calculus [24], there were defined two substitutions: partial substitution and linear substitution. Recall the linear substitution mentioned in Chapter 4.

Having defined the linear substitution for $\mathbb{A}\{\mathbb{N}/x\}$, we are now in condition to translate the resource calculi into calculus with multiplicities.

The reduction rules that reduce a term in the λ-calculus with multiplicities to a term in the resource calculus are now presented.

$\mathcal{T}_{\mathcal{R}}\ (M)$ represents the translation of a term M from the resource calculus to the λ-calculus with multiplicities. $\mathcal{T}_{\mathcal{M}}\ (M)$ represents the translation of a term M from the λ-calculus with multiplicities to the resource calculus.

Definition 13. *Given $M = N_1 + \cdots + N_n$, $N_1, \ldots, N_n$ are called summands of M.*

We will now define the compilation rule from resource calculus to the λ-calculus with

multiplicities.

Definition 14. *Let M, N be terms, P and Q be bags. We define functions $\mathcal{T}_{\mathcal{R}} : \Lambda^R \to \Lambda^M$, $\mathcal{B} : \Lambda^b \to \Lambda^P$ in the following way:*

Compilation of Terms:
$\mathcal{T}_{\mathcal{R}}\ (x) = x$

$\mathcal{T}_{\mathcal{R}}\ (\lambda x.M) = \lambda x.\ \mathcal{T}_{\mathcal{R}}\ (M)$

$\mathcal{T}_{\mathcal{R}}\ (MP) = \mathcal{T}_{\mathcal{R}}\ (M)\ \mathcal{B}_{\mathcal{R}}\ (P)$

Compilation of Bags:
$\mathcal{B}_{\mathcal{R}}\ (1) = 1$

$\mathcal{B}_{\mathcal{R}}\ (P.Q) = \mathcal{B}_{\mathcal{R}}\ (P)|\ \mathcal{B}_{\mathcal{R}}\ (Q)$

$$\mathcal{B}_{\mathcal{R}}\ ([M^{(!)}]) = \begin{cases} T_R\ (M)\ \textit{if}\ M^{(!)} = M, \\ T_R\ (M)^{\infty}\ \textit{if}\ M^{(!)} = M^! \end{cases}$$

We will now exemplify the compilation of a term. Such term will be $L = (\lambda x.x)([x]\ .\ [\lambda y.y])$.

The compilation routine will execute as follows:

$\mathcal{T}_{\mathcal{R}}\ (L) = \mathcal{T}_{\mathcal{R}}\ (\lambda x.x)\ \mathcal{B}_{\mathcal{R}}\ ((x)\ .\ (\lambda y.y)) = (\lambda x.\ \mathcal{T}_{\mathcal{R}}\ (x))(\ \mathcal{B}_{\mathcal{R}}\ (x)\ |\ \mathcal{B}_{\mathcal{R}}\ (\lambda y.y)) = (\lambda x.x)((\ \mathcal{T}_{\mathcal{R}}\ (x)|\ \mathcal{T}_{\mathcal{R}}\ (\lambda y.y)) = (\lambda x.x)(x|\lambda y.\ \mathcal{T}_{\mathcal{R}}\ (y) = (\lambda x.x)(x\ |\ \lambda y.y)$

We now show that $\mathcal{T}_{\mathcal{R}}\ (M)$ is in fact a term in the calculus with multiplicities.

Theorem 2. *If $M \in \Lambda^R$, then $\mathcal{T}_{\mathcal{R}}\ (M) \in \Lambda^M$.*

Proof. For this proof we will use structural induction on the structure of terms. To prove this theorem we prove a more general result:

1. If $M \in \Lambda^R$, then $\mathcal{T}_{\mathcal{R}}\ (M) \in \Lambda^M$.
2. If $P \in \Lambda^b$ then $\mathcal{B}_{\mathcal{R}}\ (P) \in \Lambda^P$

- When $M = x$ we have: $x \in \Lambda^R$. By the compilation we obtain that $\mathcal{T}_{\mathcal{R}}\ (x) = x$, and $x \in \Lambda^M$.
- If $M = \lambda x.N$ we have that $\lambda x.N \in \Lambda^R$. By the compilation we have that $\mathcal{T}_{\mathcal{R}}\ (\lambda x.N) = \lambda x.\ \mathcal{T}_{\mathcal{R}}\ (N)$ so by induction hypothesis, $\mathcal{T}_{\mathcal{R}}\ (N) \in \Lambda^M$, therefore if $N \in \Lambda^R$ and $P \in \Lambda^b$, $\lambda x.\ \mathcal{T}_{\mathcal{R}}\ (N) \in \Lambda^M$.
- Finally if we are facing an application, NP, we have that $NP \in \Lambda^R$. By the compilation we have that $\mathcal{T}_{\mathcal{R}}\ (NP) = \mathcal{T}_{\mathcal{R}}\ (N)\ \mathcal{B}_{\mathcal{R}}\ (P)$. By induction hypothesis we obtain that $\mathcal{T}_{\mathcal{R}}\ (N) \in \Lambda^M$ and that $\mathcal{B}_{\mathcal{R}}\ (P) \in \Lambda^P$. So, we can state that $\mathcal{T}_{\mathcal{R}}\ (NP) \in \Lambda^M$.

- If $P = 1$, then, we have that $\mathcal{B}_{\mathcal{R}}\ (1) = 1$ and $1 \in \Lambda^P$.
- If $P = Q.R \in \Lambda^b \text{if} P.Q \in \Lambda^b$, then, we have that $\mathcal{B}_{\mathcal{R}}\ (Q.R) = \mathcal{B}_{\mathcal{R}}\ (Q) \mid \mathcal{B}_{\mathcal{R}}\ (R)$. By our induction hypothesis, we have that $\mathcal{B}_{\mathcal{R}}\ (Q)$ and $\mathcal{B}_{\mathcal{R}}\ (R)$ both belong to Λ^P. Therefore, $\mathcal{B}_{\mathcal{R}}\ (Q) \mid \mathcal{B}_{\mathcal{R}}\ (R) \in \Lambda^P$.
- If $P = [M^{(!)}]$, we have two possibilities. If $M^{(!)} = M \in \Lambda^R$, then, as proved before, $\mathcal{T}_{\mathcal{R}}\ (M) \in \Lambda^M$, belonging, therefore, to Λ^P (since $\Lambda^M \subseteq \Lambda^P$). On the other hand, if $M^{(!)} = M^!$, $\mathcal{B}_{\mathcal{R}}\ [M^!] = \mathcal{T}_{\mathcal{R}}\ (M)^\infty$ and by induction hypothesis $\mathcal{T}_{\mathcal{R}}\ (M) \in \Lambda^M$, so $\mathcal{T}_{\mathcal{R}}\ (M)^\infty \in \Lambda^P$.

□

We now present the translation on the other side to prove the relation between the two calculi.

Definition 15. *Let M, N be terms, P and Q be bags. We define functions $\mathcal{T}_{\mathcal{M}} : \Lambda^M \to \Lambda^R$, $\mathcal{P}_{\mathcal{M}} : \Lambda^B \to \Lambda^b$ in the following way:*

Compilation of Terms:
$\mathcal{T}_{\mathcal{M}}\ (x) = x$

$\mathcal{T}_{\mathcal{M}}\ (\lambda x.M) = \lambda x.\ \mathcal{T}_{\mathcal{M}}\ (M)$

$\mathcal{T}_{\mathcal{M}}\ (MP) = \mathcal{T}_{\mathcal{M}}\ (M)\ \mathcal{P}_{\mathcal{M}}\ (P)$

$\mathcal{T}_{\mathcal{M}}\ (M\langle P/x\rangle) = \mathcal{T}_{\mathcal{M}}\ S(M\langle P/x\rangle)$ *where $S(M)$ consists on applying the substitution rules represented in 3.7, 3.8 and 3.9 to M as many times as needed until it no longer represents a substitution.*

Compilation of Bags:
$\mathcal{P}_{\mathcal{M}}\ (1) = 1$

$\mathcal{P}_{\mathcal{M}}\ (M) = \mathcal{T}_{\mathcal{M}}\ (M)$

$\mathcal{P}_{\mathcal{M}}\ (P|Q) = \mathcal{P}_{\mathcal{M}}\ (P).\ \mathcal{P}_{\mathcal{M}}\ (Q)$

$\mathcal{P}_{\mathcal{M}}\ (M^\infty) = \mathcal{T}_{\mathcal{M}}\ (M^!)$

Theorem 3. *If $M \in \Lambda^M$, then $\mathcal{T}_{\mathcal{M}}\ (M) \in \Lambda^R$.*

Proof. For this proof we will use structural induction on the structure of terms. To prove this theorem we prove a more general result:

1. If $M \in \Lambda^M$ then $\mathcal{T}_{\mathcal{M}} \in \Lambda^R$.
2. If $P \in \Lambda^P$ then $\mathcal{P}_{\mathcal{M}} \in \Lambda^b$.

- When $M = x$ we have: $x \in \Lambda^M$. By the compilation we obtain that $\mathcal{T}_\mathcal{M}\ (x) = x$, and $x \in \Lambda^R$.
- If $M = \lambda x.N$ we have that $\lambda x.N \in \Lambda^M$. By the compilation we have that $\mathcal{T}_\mathcal{M}\ (\lambda x.N) = \lambda x.\ \mathcal{T}_\mathcal{M}\ (N)$ so by induction hypothesis, $\mathcal{T}_\mathcal{M}\ (N) \in \Lambda^R$, therefore if $N \in \Lambda^M$ and $P \in \Lambda^b$, $\lambda x.\ \mathcal{T}_\mathcal{M}\ (N) \in \Lambda^M$.
- Finally if we are facing an application, NP, we have that $NP \in \Lambda^M$. By the compilation we have that $\mathcal{T}_\mathcal{M}\ (NP) = \mathcal{T}_\mathcal{M}\ (N)\ \mathcal{P}_\mathcal{M}\ (P)$. By induction hypothesis we obtain that $\mathcal{T}_\mathcal{M}\ (N) \in \Lambda^M$ and that $\mathcal{P}_\mathcal{M}\ (P) \in \Lambda^P_M$. So, we can state that $\mathcal{T}_\mathcal{M}\ (NP) \in \Lambda^M$.
- If $M = N\langle P/x\rangle$, then $M \in \Lambda^M$. Since we apply successively rules 3.7, 3.8 and 3.9 to $N\langle P/x\rangle$, we get that the result, call it N', will also belong to Λ^M. Since $N' \in \Lambda^M$, we apply $\mathcal{T}_\mathcal{M}$ to it, which will yield a term in Λ^R.
- If $P = 1$, then, we have that $\mathcal{P}_\mathcal{M}\ (1) = 1$ and $1 \in \Lambda^P$.
- If $P = M$, with $M \in \Lambda^M$, then, by our induction hypothesis, we have that $\mathcal{T}_\mathcal{M}\ (M) \in \Lambda^R$.
- If $P = Q.R \in \Lambda^b$ if $Q.R \in \Lambda^b$, then, we have that $\mathcal{P}_\mathcal{M}\ (Q.R) = \mathcal{P}_\mathcal{M}\ (Q) \mid \mathcal{P}_\mathcal{M}\ (R)$. By our induction hypothesis, we have that $\mathcal{P}_\mathcal{M}\ (Q)$ and $\mathcal{P}_\mathcal{M}\ (R)$ both belong to Λ^P. Therefore, $\mathcal{P}_\mathcal{M}\ (Q) \mid \mathcal{P}_\mathcal{M}\ (R) \in \Lambda^P$.
- If $P = [M^{(!)}]$, we have two possibilities. If $M^{(!)} = M \in \Lambda^R$, then, as proved before, $\mathcal{T}_\mathcal{M}\ (M) \in \Lambda^M$, belonging, therefore, to Λ^P (since $\Lambda^M \subseteq \Lambda^P$). On the other hand, if $M^{(!)} = M^!$, $\mathcal{P}_\mathcal{M}\ [M^!] = \mathcal{T}\ (M)^\infty$ and by induction hypothesis $\mathcal{T}_\mathcal{M}\ (M) \in \Lambda^M$, so $\mathcal{T}_\mathcal{M}\ (M)^\infty \in \Lambda^P$.

□

Conjecture 1. *If $M \in \Lambda^R$ and $M \to^b_R N$ and $\mathcal{T}_\mathcal{R}\ (M) = M^*$ and $M^* \to^*_M N^*$, then $N^* = \mathcal{T}_\mathcal{M}\ (N_i)$ when N_i is a summand of N.*

Conjecture 2. *If $M \in \Lambda^M$ and $M \to^*_M N$ and $\mathcal{T}_\mathcal{M}\ (M) = M^*$ and $M^* \to^b_R N^*$, then $N^* = \mathcal{T}_\mathcal{R}\ (N_i)$ is a summand of N.*

In the next section we will present our implementation, in Haskell, of the translation methods presented in this Chapter.

5.1 Implementation

Let us now present the implementation of our translation.

The following board shows the compilation of a term $M \in \Lambda^R$ to a term $M' \in \Lambda^M$. One can see that all the implementation is pretty straightforward, in the sense that it results directly from the intuitive mapping between the two grammars of both calculi.

```
compileT :: Rterms -> Term
compileT EmptyT = TEmpty
compileT (Var s) = V s
compileT (Simona.Lambda s rt) = Lib.Lambda s (compileT rt)
compileT (Simona.App rt rb) = Lib.App (compileT rt) (compileB rb)
```

Next, we present the translation for bags. When it comes to bags, the translation may not be as direct as the one presented before. Nonetheless, the following board shows the compiling function of a term $P \in \Lambda^b$ to a term $P' \in \Lambda^P$.

If $P = Empty$ or $P = TSingle\ t$, the translation is again, pretty straightforward. However, if $R = (P.Q)$, one must be aware of some details. We do not have a $P.Q$ constructor *per se* in the λ-calculus with multiplicites. Nonetheless, it does contain the $P|Q$ that contains similar properties, being, therefore, possible to map one into the other, as proven in our Theorem above.

If $P = [M^{(!)}]$, then some details must be considered in the translation algorithm as well, since $[M^{(!)}]$ can either be M or $M^!$. Our *transformResources* function treats each $R \in \Lambda^!$ different according to their nature. If $[M^{(!)}] = M$, where $M \in \Lambda^R$, then, we have the case where M is translated as a term of Λ^R. However, M needs to be encapsulated in a Bag, since the constructor $(P|Q)$ deals with Bags on both arguments. That is no problem, since all terms can be bags, according to the grammar of both calculi. Therefore, a "bag" is created, which consists of a term $M \in \Lambda^R$ encapsulated in a bag constructor. When $[M^{(!)}] = M^!$, a possible way to translate $M^!$ to a term $N \in \Lambda^P$ is to transform the "!" into a ∞. It means that $M^!$ translates to $M^\infty \in \Lambda^P$. The only aspect in this translation algorithm left to explain is the use of the $(P|Q)$ constructor as a way of simulating the behavior of a list of resources in resource calculus. Since the λ-calculus with multiplicities does not contain any other structure that can agreggate multiple terms, the only way to "construct a list" in the λ-calculus with multiplicites is to use the $(P|Q)$ constructor. In such way, one can map $[M, N, P]$ into $(M|N|P)$, and maintain the order intact.

```
compileB :: Rbags -> Bag
compileB Simona.Empty = Lib.Empty
compileB (Simona.TSingle rt) = Lib.TSingle (compileT rt)
compileB (Dot rb1 rb2) = Par (compileB rb1)  (compileB rb2)
compileB (Multi l) = transformResources l
```

```
transformResources :: [Resource] -> Bag
transformResources [] = Lib.Empty
transformResources (BangTerm bt : xs) = Par (Lib.TInfinite (compileT bt))
    (transformResources xs)
transformResources (Term t : xs) = Par (Lib.TSingle (compileT t))
    (transformResources xs)
```

We also have created another translation function that transform the lambda-caluclus with multiplicities into resource calculi. Again, some of the aspects of the implementation can seem rather intuitive, while others may seem not so straightforward.

We begin by presenting our function that translates a term $M \in \Lambda^M$ to a term $N \in \Lambda^R$. The *Empty* term, the variable, the abstraction and the application cases are very straightforward, similar to the translation from the resource calculus to the λ-calculus with multiplicities. All the particularities of this function are related to translating the *Subs M P s* constructor. Since resource calculus does not contain an explicit constructor to represent substitutions, we had to create a function. The way we did this was to apply all the substitutions to the term $M = \text{Subs } N\ Ps$ until it no longer represented a substitution and then we translate it as we would translate a regular term $M \in \Lambda^M$. The needed substitutions are all performed by the auxiliary function *finalSub*.

```
compileT :: Term -> Rterms
compileT  TEmpty = EmptyT
compileT (V s) = Var s
compileT (Lib.Lambda s mt) = Simona.Lambda s (compileT mt)
compileT (Lib.App mt mb) = Simona.App (compileT mt) (compileB mb)
compileT (Subs mt mb s) = compileT (finalSub (Subs mt mb s))
```

```
finalSub :: Term -> Term
finalSub (Subs mt mb s) = finalSub (sub $ Subs mt mb s)
finalSub t = t
```

Below we present the translation of a bag P. If $P = 1$, then it translates directly to 1 in the resource calculus. If P is, in fact, a term $M \in \Lambda^M$, then the translation is also pretty straightforward, since both calculi have this situation explicitly represented in their grammars.

If $P = M^\infty$, then, according to the Theorem proved before, we can see that it translates to a $M^!$. However, we cannot represent, as a Bag, a loose $M^!$, so we represent it as a singleton list $[M^!]$, that is, in fact, a bag, according to the grammar of the resource calculus.

If $P = (P'|Q)$, then, analogously to what we did in translating the resource calculi into the λ-calculus with multiplicites, we can use the $(P.Q)$ constructor to represent $(P'|Q)$ as $(P'' = P'.Q)$. In that way, we can easily see that $P'' \in \Lambda^P$.

```
compileB :: Bag -> Rbags
compileB Lib.Empty = Simona.Empty
compileB (Lib.TSingle mt) = Simona.TSingle (compileT mt)
compileB (Lib.TInfinite mt) = Multi [BangTerm (compileT mt)]
compileB (Par (Lib.TSingle mt) mb) = Dot (compileT mt) (compileB mb)
```

5.1.1 Example

In this section we present some examples to prove the similarity between these two calculi.

```
*Translation> compileT (Simona.App (Var "x") (Dot (Simona.TSingle (Var
    "y"))(Simona.TSingle (Var "z"))))
App (V "x") (Par (TSingle (V "y")) (TSingle (V "z")))
```

In the box above we can observe that the translation was made. In the resource calculi we have $x(y.z)$, and in the λ-calculus with multiplicities we have $x(y|z)$. If we check the compilation we can conclude that the translation is correct, for this case. That means that,

$$\mathcal{T}_{\mathcal{R}}\ (x(y.z)) = \mathcal{T}_{\mathcal{R}}\ (x)\ \mathcal{B}_{\mathcal{R}}\ (y.z) = x\ \mathcal{B}_{\mathcal{R}}\ (y)\ \mathcal{B}_{\mathcal{R}}\ (z) = x(y|z).$$

Next we present an example of the translation from λ-calculus with multiplicities to resource calculi. For this example we test the output of the previous example and we expect the same result.

```
*Translation2> Translation2.compileT (Lib.App (V "x") (Par (Lib.TSingle (V
    "y")) (Lib.TSingle (V "z"))))
App (Var "x") (Dot (TSingle (Var "y")) (TSingle (Var "z")))
```

The box above presents the translation between the λ-calculus with multiplicities to the resource calculus. We will present a step-by-step translation of $(x)(y|z)$:

$$\mathcal{T}_{\mathcal{M}}\ (x)(y|z) = \mathcal{T}_{\mathcal{M}}\ (x)\ \mathcal{B}_{\mathcal{M}}\ (y|z) = x\ \mathcal{B}_{\mathcal{M}}\ (y).\ \mathcal{B}_{\mathcal{M}}\ (z) = x(y.z).$$

We can denote that the output is similar, and the meaning is the same, for this example. In λ-calculus with multiplicities our examples became $x(y|z)$ and the result $x(y.z)$, that is the expected result, based on the grammar and in the previous test.

Chapter 6

Conclusion and Future Research

In this chapter we will present some final thoughts on the matter related to our document.

Our main goal for this work, was to prove a relation between λ-calculus with multiplicities and resource calculus. To achieve this goal we implemented both calculi in Haskell, and then we created a translation. We programmed it Haskell because of previous experiences with this language and coding the grammar into a functional language seemed a natural choice.

Observing the implementation, one can see that the calculi are related, since one can translate the a term in one setting into a different term in the other setting.

6.1 Future Work

In the future, we believe that it is important to try to establish a relation between both of these calculi and linear calculus.

6.2 Final Remarks

Logic has always been an area of interest to me, in the vast field of Computer Science. In particular, in a MSc course, I had my first interaction with λ-calculus and this particular topic got me blown away by it's elegance and by the fact that it constitutes the basis of an important part of Computer Science. With this thesis, I have been having a great chance to dig deeper into this subject.

In terms of results, not only I learned a lot of theoretical models and different approaches in the most various ways in λ-calculus, but also have improved my skills in implementing theoretical models, in particular, using Haskell.

Finally, I sincerely hope that the reader also shares a little bit of my curiosity and enthusiasm and can appreciate what I present here as much as I have appreciated learning all of this.

Bibliography

[1] Abadi, M., Cardelli, L., Curien, P.-L., and Lévy, J.-J. (1991). Explicit substitutions. *J. Funct. Program.*, 1:375–416.

[2] Alves, S., Fernández, M., Florido, M., and Mackie, I. (2010). Gödel's system tau revisited. *Theor. Comput. Sci.*, 411(11-13):1484–1500.

[3] Alves, S., Fernández, M., Florido, M., and Mackie, I. (2011). Linearity and recursion in a typed lambda-calculus. In *Proceedings of the 13th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 20-22, 2011, Odense, Denmark*, pages 173–182.

[4] Alves, S., Fernández, M., Florido, M., and Mackie, I. (2014). Linearity: A roadmap. *J. Log. Comput.*, 24(3):513–529.

[5] Barendregt, H. P. et al. (1984). *The lambda calculus*, volume 3. North-Holland Amsterdam.

[6] Bernardy, J.-P. and Newton, R. (2017). Linear haskell.

[7] Boudol, G. (1993a). The lambda-calculus with multiplicities. In *International Conference on Concurrency Theory*, pages 1–6. Springer.

[8] Boudol, G. (1993b). The lambda-calculus with multiplicities (abstract). In *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, pages 1–6.

[9] Church, A. (1932). A set of postulates for the foundation of logic. *Annals of mathematics*, pages 346–366.

[10] Florido, M. and Damas, L. (2004). Linearization of the lambda-calculus and its relation with intersection type systems. *Journal of Functional Programming*, 14(5):519–546.

[11] Girard, J. (1987). Linear logic. *Theor. Comput. Sci.*, 50:1–102.

[12] Gödel, V. K. (1958). Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *dialectica*, 12(3-4):280–287.

[13] Gordon, M. J. C., Milner, R., Morris, L., Newey, M. C., and Wadsworth, C. P. (1978). A metalanguage for interactive proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 119–130.

[14] Henglein, F. (1993). Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289.

[15] Hindley, J. R. (1997). *Basic simple type theory*. Number 42. Cambridge University Press.

[16] Hindley, J. R. and Seldin, J. P. (2008). *Lambda-calculus and Combinators, an Introduction*, volume 13. Cambridge University Press Cambridge.

[17] Hofmann, M. (1999). Linear types and non-size-increasing polynomial time computation. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 464–473.

[18] Jones, S. P. and Wadler, P. (1992). A static semantics for haskell. *Draft paper, Glasgow*, 91.

[19] Jost, S., Vasconcelos, P., Florido, M., and Hammond, K. (2017). Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning*, 59(1):87–120.

[20] Kfoury, A. J. (2000). A linearization of the lambda-calculus and consequences. *J. Log. Comput.*, 10(3):411–436.

[21] Lévy, J.-J. (1978). *Réductions correctes et optimales dans le lambda-calcul*. PhD thesis.

[22] Milner, R. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375.

[23] Milner, R. (1992). Functions as processes. *Mathematical structures in computer science*, 2(2):119–141.

[24] Pagani, M. and Rocca, S. R. D. (2010). Solvability in resource lambda-calculus. In *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, pages 358–373.

[25] Plotkin, G. D. (1977). Lcf considered as a programming language. *Theoretical computer science*, 5(3):223–255.

[26] Thompson, S. (2011). *Haskell: the craft of functional programming*, volume 2. Addison-Wesley.

[27] Troelstra, A. S. (1992). Lectures on linear logic.

[28] Wadler, P. (1990). Linear types can change the world! In *Programming Concepts and Methods*.

[29] Zilli, M. V. (1984). Reduction graphs in the lambda calculus. *Theoretical Computer Science*, 29(3):251–275.

Appendix A

Implementation in Haskell

```
-----------------------------------------------------
-- Implementation of the calculus with multiplicities --
-----------------------------------------------------

module Lib
    ( Term(V, Lambda, App, Subs,TEmpty),
      Bag(Empty, TSingle, TInfinite, TFinite, Par),
      red, sub, redL, subL, fetch, rename, fv, all_terms, nth, construct_terms,
          groupTerms, createBags, fetch_all, garbage, clean, normalize, included
    ) where


import Data.List

---------------------
-- Data Structures --
---------------------

data Bag =  Empty
         | TSingle Term
         | TInfinite Term
         | Par Bag Bag
         deriving (Show, Eq)

data Term = TEmpty
          | V String
          | Lambda String Term
          | App Term Bag
          | Subs Term Bag String
          deriving (Show, Eq)

---------------------
-- Free Variables --
---------------------
```

```
fv :: Term -> [String] -> [String]

fv (V x) lb = if elem x lb then [] else [x]
fv (Lambda x t) lb = fv t (lb ++ [x])
fv (App t (TSingle b)) lb = fv t lb ++ fv b lb
fv (Subs t (TSingle b) s) lb = fv t lb ++ fv b lb

------------
-- Value --
------------

value :: Term -> Bool

value (Lambda m x) = True
value (Subs v b x) = value v
value _ = False

------------
-- Equals --
------------

equals :: Bag -> Bag

equals (Par p Empty) = p
equals (Par p q) = Par q p
equals (Par p (Par q r)) = Par (Par p q) r
equals (TInfinite m) = Par (TSingle m) (TInfinite m)

--------------------------
-- Reduction List -------
--------------------------

redL :: Term -> [Term]
redL t = [red t]

---------------
-- Reduction --
---------------

red :: Term -> Term

red (App (Lambda x  m) p) = Subs m p x
red (App (Subs v r x) p) = Subs (red (App v p)) r x
red (App m p) = App (red m) p
red (Subs m p x) = Subs (red m) p x

----------------------
-- Substitution List --
----------------------
```

```
subL :: Term -> [Term]
subL t = [sub t]

-- ------------------
-- Substitution --
-- ------------------

sub :: Term -> Term

sub (Subs (V x) (TSingle m) y) |
  x == y = m
sub (Subs (V x) (TInfinite m) y) |
  x == y  = m
sub (Subs (V x) (Empty) y) = if x == y then TEmpty else (V x)
sub (Subs (App m p) n x) = App (sub (Subs m n x)) p
sub (Subs (Subs m p z) n x) = Subs (sub (Subs m n z)) p z
sub (Subs (Lambda x t) n y) = if elem y (fv (Lambda x t) [] ) then Lambda x
    (sub $ Subs t n y) else Lambda x t

-- ------------------------------------------
-- Fetch Rule and Auxiliary Functions --
-- ------------------------------------------

all_terms :: Bag -> [Bag]

all_terms (Par p q) = [p] ++ all_terms q
all_terms p = [p]

nth :: [a] -> Int -> a

nth [] _ = error "Index too high"
nth (x:xs) 0 = x
nth (x:xs) n = nth xs (n-1)

construct_terms :: [Bag] -> Bag

construct_terms [x] = x
construct_terms (x:xs) = Par x (construct_terms xs)

groupTerms :: [Bag] -> [Bag]

groupTerms (x:xs) = createBags (permutations (x:xs))

createBags :: [[Bag]] -> [Bag]

createBags [] = []
createBags (x:xs) = construct_terms x : createBags xs


fetch_list :: [Bag] -> Term -> String -> [Term]
fetch_list [] m s = []
```

```
fetch_list (x:xs) m s = fetch (Subs m x s) ++ fetch_list xs m s

fetch_all :: Term -> [Term]
fetch_all (Subs m n x) = fetch_list (groupTerms $ all_terms n) m x

fetch :: Term -> [Term]

fetch (Subs m (Par n r) x) = [Subs (sub (Subs m n x)) r x]

-- ------------------------
-- Garbage Collection --
-- ------------------------

garbage :: Term -> Term
garbage (Subs m p x) = if not (elem x (fv m [])) then m else (Subs m p x)

-- ------------------------
-- Remove Duplicants --
-- ------------------------

clean :: Term -> [Term]
clean (Subs m n x) = map garbage (fetch_all (Subs m n x))
clean x = [x]

removeDuplicates :: [Term] -> [Term]
removeDuplicates t = rdHelper [] t
                where rdHelper seen [] = seen
                      rdHelper seen (x:xs) = if elem x seen then rdHelper seen
                          xs else rdHelper (seen ++ [x]) xs

-- ------------------------------------------------------------
-- Top Level from function Clean and auxiliary functions --
-- ------------------------------------------------------------

normalize :: Term -> [[Term]]
normalize t  = normalize_ [t] [[t]]

normalize_ :: [Term] -> [[Term]] -> [[Term]]
normalize_ ts ls = if t' == [] || length ( concat $ included t' ls) == length
    t' then ls else normalize_ t'  (ls ++ [t'])
                  where
                    t' = concat $ map clean ts

included :: Eq a => [a] -> [[a]] -> [[a]]
included ls nest = filter func nest
                where
                    func x = any (ls==) $ concat $ map subsequences $
                        permutations x
```

```
-- ------------------------------------------------
```

```
-- Implementation of the Resource Calculus --
-----------------------------------------------

module Simona
    (Rterms(EmptyT, Var, Lambda, App),
    Resource(Term, BangTerm),
    Rbags(Empty, Dot, Multi,TSingle),
    Exp(ExpTerm, ExpBag),
    SumTerms(ZeroT, SumTerm, AddT),
    SumBags(ZeroB, SumBag, AddB),
    SumExp(Union),
    subR, subB, subR1, subR2, subN, redg
    ) where

import Data.List

-----------------------
-- Data Structures --
-----------------------

data Rterms = EmptyT
            | Var String
            | Lambda String Rterms
            | App Rterms Rbags
            deriving (Show)

data Resource = Term Rterms
              | BangTerm Rterms
              deriving Show

data Rbags = Empty
           | TSingle Rterms
           | Dot Rbags Rbags
           | Multi [Resource]
           deriving Show

data Exp = ExpTerm Rterms
         | ExpBag Rbags
         deriving Show

data SumTerms = ZeroT
                | SumTerm Rterms
                | LambdaST String SumTerms
                | AppST SumTerms Rbags
                | AddT SumTerms SumTerms
                deriving Show

data SumBags = ZeroB
             | SumBag Rbags
             | AppSB SumTerms SumBags
             | DotSB SumBags SumBags
```

```
              | AddB SumBags SumBags
              | BangTermsSB SumTerms
              | MultiSB [SumBags]
              deriving Show

data SumExp = Union SumTerms SumBags
              deriving Show

-----------------------
-- Substitutions --
-----------------------

unpack :: Maybe Rbags -> Rbags
unpack (Just a) = a
unpack Nothing = Empty

unpackTerms :: Maybe Rterms -> Rterms
unpackTerms (Just a) = a
unpackTerms Nothing = EmptyT

subR :: Rterms -> Rterms -> String -> Maybe Rterms

subR (Var y) n x
  | y == x = Just n
  | otherwise = Nothing

subR (Lambda y m) n x = Just $ Lambda y unpacked
                  where packed = subR m n x
                        unpacked = unpackTerms packed

subB :: Rbags  -> Rterms -> String -> Maybe Rbags

subB (Multi [Term m]) n x = Just $ Multi [Term (unpackTerms $ subR m n x)]

subB Empty _ _ = Nothing

subB (Multi [BangTerm m]) n x = Just $ Multi [Term (unpackTerms $ subR m n x),
    BangTerm m]


subR1 :: Rterms -> Rterms -> String -> SumTerms
subR1 (App m p) n x = AddT ( SumTerm ( App (unpackTerms $ subR m n x) p )) (
    SumTerm ( App  m (unpack $ subB p n x)))

subR2 :: Rbags -> Rterms -> String -> SumBags
subR2 (Dot p r) n x = AddB (SumBag (Dot (unpack $ subB p n x) r)) (SumBag (Dot
    p (unpack $ subB r n x)))

-------------------------
-- Nat(Sum) Expressions --
-------------------------
```

```
nat1 :: SumTerms -> SumTerms
nat1 ZeroT = ZeroT
nat1 (LambdaST x (AddT st1 st2)) = AddT (LambdaST x st1) (LambdaST x (nat1 st2))
nat1 m = m

nat2 :: SumTerms -> SumTerms
nat2 ( AppST (SumTerm m) p) = AppST (SumTerm m) p
nat2 ( AppST (AddT st1 st2) p ) = AddT (AppST st1 p) (nat2 (AppST st2 p))

nat3 :: SumBags -> SumBags
nat3 ( AppSB m (SumBag p)) = AppSB m (SumBag p)
nat3 ( AppSB m (AddB sb1 sb2)) = AddB (AppSB m sb1) (nat3 (AppSB m sb2))

nat4 :: SumBags -> SumBags
nat4 (DotSB (MultiSB l) p) = DotSB (aux l) p

aux :: [SumBags] -> SumBags
aux = foldr (\x -> AddB (MultiSB [x])) ZeroB

nat5 :: SumBags -> SumBags
nat5 (DotSB (MultiSB l) p) = DotSB (MultiSB (aux2 l)) p

aux2 :: [SumBags] -> [SumBags]
aux2 [BangTermsSB (AddT st1 st2)] = [(BangTermsSB st1)] ++ (aux2 [BangTermsSB
    st2])

-------------------------
--Normal Substitution --
-------------------------

freev :: Rterms -> [String] -> [String]

freev (Var x) lb = if elem x lb then [] else [x]
freev (Lambda x t) lb = freev t (lb ++ [x])
freev (App t (TSingle b)) lb = freev t lb ++ freev b lb


subN :: Rterms -> Rterms -> String -> Rterms

subN (Var x) m y |
  x == y = m
subN (App m p) n x = App (subN m n x) p
subN (Lambda x t) n y = if elem y (freev (Lambda x t) [] ) then Lambda x (subN
    t n y) else Lambda x t

-------------------------
-- Reduction: Giant Step --
-------------------------

redg :: Rterms -> Rbags -> Rterms
```

```
redg (Lambda x t) (Multi n) = apply_subR (apply_subN t x l') x l''
                where (l',l'') = split n [] []

apply_subN :: Rterms -> String -> [Resource] -> Rterms
apply_subN t y [] = t
apply_subN t y (Term x:xs) = apply_subN (subN t x y) y xs

apply_subR :: Rterms -> String -> [Resource] -> Rterms
apply_subR t y [] = t
apply_subR t y (BangTerm x:xs) = apply_subR (unpackTerms $ subR t x y) y xs

split :: [Resource] -> [Resource] -> [Resource] -> ([Resource], [Resource])
split [] ls xs = (ls,xs)
split ((Term t):ys) ls xs = split ys (ls ++ [Term t]) xs
split (BangTerm t:ys) ls xs = split ys ls (xs ++ [BangTerm t])

-----------------
-- Normalize --
-----------------

normalizeS :: Rterms -> Rbags -> Rterms
normalizeS rterm rbag = redg rterm rbag
```

```
-------------------------------------------------------------------------
-- Translation from Resource Calculus to Lambda-Calculus with multiplicities --
-------------------------------------------------------------------------

module Translation where

import Lib
import Simona
import Data.List

-----------------------------
-- Compilation of Terms --
-----------------------------

compileT :: Rterms -> Term
compileT EmptyT = TEmpty
compileT (Var s) = V s
compileT (Simona.Lambda s rt) = Lib.Lambda s (compileT rt)
compileT (Simona.App rt rb) = Lib.App (compileT rt) x
                            where x = compileB rb

-------------------------
-- Compilation Bags --
-------------------------

compileB :: Rbags -> Bag
```

```
compileB Simona.Empty = Lib.Empty
compileB (Simona.TSingle rt) = Lib.TSingle (compileT rt)
compileB (Dot rb1 rb2) = Par (compileB rb1) x
                      where x = compileB rb2
compileB (Multi l) = transformResources l

-------------------------------------------------
-- Compiling of every term in the multi-set --
-------------------------------------------------

transformResources :: [Resource] -> Bag
transformResources [] = Lib.Empty
transformResources (BangTerm bt : xs) = Par (Lib.TInfinite (compileT bt))
    (transformResources xs)
transformResources (Term t : xs) = Par (Lib.TSingle (compileT t))
    (transformResources xs)
```

```
-------------------------------------------------------------------------
-- Translation from Lambda-Calculus with multiplicities to Resource Calculus --
-------------------------------------------------------------------------

module Translation2 where

import Lib
import Simona
import Data.List

-----------------------------
-- Compilation of Terms --
-----------------------------

compileT :: Term -> Rterms
compileT  TEmpty = EmptyT
compileT (V s) = Var s
compileT (Lib.Lambda s mt) = Simona.Lambda s (compileT mt)
compileT (Lib.App mt mb) = Simona.App (compileT mt) (compileB mb)
compileT (Subs mt mb s) = compileT (finalSub (Subs mt mb s))

-----------------------------
-- Compilation of Bags --
-----------------------------

compileB :: Bag -> Rbags
compileB Lib.Empty = Simona.Empty
compileB (Lib.TSingle mt) = Simona.TSingle (compileT mt)
compileB (Lib.TInfinite mt) = Multi [Term (compileT mt)]
compileB (Par (Lib.TSingle mt) mb) = Dot (compileT mt) (compileB mb)

-----------------------------
-- Boudol's Substitutions --
```

```
-------------------------------


finalSub :: Term -> Term
finalSub (Subs mt mb s) = finalSub (sub $ Subs mt mb s)
finalSub t = t
```