Synthesis of Programs from Linear Types

Maria Inês Melo e Sousa
Master's degree in Computer Science
Departamento de Ciência de Computadores
2021

Supervisor
António Mário da Silva Marcos Florido, Associate Professor,
Faculdade de Ciências da Universidade do Porto

Co-Supervisor
Sandra Maria Mendes Alves, Assistant Professor,
Faculdade de Ciências da Universidade do Porto

The President of the Jury,

Porto, ______/______/_________

Abstract

The synthesis of programs from linear types is a subject matter that has had a growing interest in recent years, and, therefore, has seen strong developments. This work aims to study and implement a program synthesis system from annotated types with computational resources, i.e., producing a code given a type and inductively synthesize well-typed programs. These type systems are based on linear types, originated from linear logic, and are currently available in programming languages such as Linear Haskell.

Resumo

A síntese de programas a partir de tipos lineares é um assunto que tem tido um interesse crescente nos últimos anos e, portanto, tem visto um forte desenvolvimento. Este trabalho visa estudar e implementar um sistema de síntese de programas a partir de tipos anotados com recursos computacionais, ou seja, produzir código partindo de um tipo e sintetizando indutivamente programas bem tipados. Estes sistemas de tipos são baseados em tipos lineares, com origem na lógica linear, e atualmente estão disponíveis em linguagens como o Linear Haskell.

Acknowledgments

First and foremost, I would like to thank my supervisors and teachers Mário Florido and Sandra Alves for the guidance and support that they gave me through this year, for the availability, explanations, assiduous monitoring and correction of my development and writing of this dissertation, and for the encouragement and effort put in me.

Thank you to my friends who accompanied me on this journey, in special to Teófilo, Pires, Guilherme, and Joana, that were always present. And, to my roommates and friends, Matilde and Cynthia, that always provided a pleasant and fun environment.

Additionally, I would like to acknowledge my family, especially my parents and sister for their encouragement and unconditional support, and my family friend, Luisa Mota Vieira, for her support and advices.

Last but not least, a special thanks to Miguel for all his support, company and understanding.

Contents

List of Figures

Listings

Chapter 1

Introduction

Program Synthesis is a subject whose purpose is to generate automatically programs from certain specifications, in this case type information with resource annotations. This has been a subject that has increased in popularity and, consequently, growing through studies carried out in recent years. This thesis is intended to carry out a study and implementation of a system for Program Synthesis from annotated types resorting to computational resources, that is, producing code in a system that works as an inversion of Type Inference (Type Inhabitation), starting from a type and inductively synthesizing well-typed subterms. Our first approach is based on Terms with Graded Types [1].

1.1 Motivation

Formal type systems are extremely important nowadays in computer science research because, with these systems, one can prove program properties in a sound setting. We studied type systems based on linear types, which come from Linear Logic, some of them available in linear languages such as Linear Haskell [2]. A function to be linear consumes its arguments exactly once. So one important program property is the number of times an argument (resource) is consumed. One solution is the use of graded modal types that use both linear and graded types, i.e., the resources are annotated with a grade or multiplicity, which states the number of times that the resource still can be used.

The main motivation for this dissertation is to check how difficult it would be to implement these systems in Prolog following a Type Inhabitation approach [3, 4]. Another motivation is the fact of enabling resources usage control, by having an initial bound on the number of times the resource can be consumed, which may help remove several bugs.

1.2 Objectives

Through this work we want to study and implement a Program Synthesis system from annotated types, using computational resources. The implementation is performed in the Prolog language. Therefore, the main objective is to implement a Program Synthesis system for Terms with Graded Types [1], and a Program Inference system for Graded Linear Types [1] and Linear Haskell [2]. Finally we want to create a new Program Synthesis system for Partial Typed Terms.

1.3 Contribution

In order to achieve all the objectives stated, our approach and contribution are the following:

- Do a review study for the λ-calculus and simple types. The intention is to gain knowledge with the concepts involved in λ-calculus, to be able to develop this work.
- Study the simple type systems, to comprehend the structure and language of a system à la Curry and a system à la Church, and their main differences. The study reviews the type problems, like Type Inhabitation, Type Inference, and Type Checking, to know their differences and where do they fit, in which system.
- Study and describe the main concepts of the type system Graded Linear Types, and implement it. This is essential to understand how type systems can provide an infrastructure for the Program Syntheses systems. This system focus on the linearity of the assumptions, which are annotated with grades.
- Describe and implement a subset of Linear Haskell (Haskell for linear types). The extension focus on the linearity of the function arrow, which is annotated with a multiplicity, which is similar to the grade.
- Study and implement the Program Synthesis system, the system of Terms with Graded Types. This is the main contribution of this dissertation. It makes use of the same syntax language of the system of Graded Linear Types.
- Describe and implement the rules based on Type Inhabitation, for the new notion of Partial Typed Terms, and fill a special term associated with a well typed type.

1.4 Organization

This present dissertation is organized into five chapters, including this introduction aimed at contextualizing the reader to the present topic of Synthesized Programs from linear types, and the approach that was made. Chapter 2 begins by reviewing the main concepts of the λ-calculus and its simple types, by presenting their definitions and some examples. Chapter 3 provides a description of two type systems: the Graded Linear Types system, and the Linear Haskell system. There they are explained in more detail to better understand their mechanism and inference rules. Chapter 4 is about Program Synthesis, which is the main focus of this dissertation. It begins by describing the Terms with Graded Types system, and his inference rules and respective code implementation. Next, it explains the new notion of Typed Partial Terms and why is used, and describe its implementation rules. Finally, in Chapter 5 the conclusions are provided with future prospects.

Chapter 2

Lambda-calculus and Simple Types

This chapter is an approach to the λ-calculus and its simple types, following the theoretical research in [5]. Here, the basis, which guides the work performed through the dissertation, is explained in more detail by introducing some properties and definitions of the λ-calculus, and its simple types theory.

2.1 Lambda-calculus

The λ-calculus is a formal system created by the mathematician Alonzo Church [6]. Through this formal system of mathematical logic, Church defined the computable functions that would serve as a model for functional programming languages (e.g. *Haskell*, *ML*).

The terms of λ-calculus are constructed from an infinite alphabet of type variables and denotes functions abstraction and functions application. Those functions can be applied to any arguments, including the functions themselves, making the λ-calculus a *type-free theory*.

Definition 2.1.1 (Syntax)**.** There are three forms of defining the terms in λ-calculus:

$$\begin{array}{rl} x & \text{(Variable)} \\ (\lambda x.t_1) & \text{(Abstraction)} \\ (t_1 t_2) & \text{(Application)} \end{array}$$

These rules define a λ-term as a variable x, an abstraction function $(\lambda x.t_1)$, with a parameter x and a body t_1, and an application function $(t_1 t_2)$, that represents a t_1 applied to an argument t_2. Note that x, t_1, and t_2 are λ-terms.

Abstractions are right-associative, and applications are left-associative, which allows the following abbreviations:

$$
\begin{array}{lcl}
(\lambda x_1 \ldots x_n.t) & \equiv & (\lambda x_1.(\ldots(\lambda x_n.t))) \\
(t_1 t_2 \ldots t_n) & \equiv & (\ldots(t_1 t_2)\ldots t_n)
\end{array}
$$

The variables of a term can be classified as free variables or bound variables.

Definition 2.1.2 (Free and Bound Variables)**.** If an occurrence of a variable x in a term t appears in a subterm of the form $\lambda x.t$, then it is a bound occurrence. Otherwise, it is a free occurrence.

Definition 2.1.3. The set of free variables of t, $\mathrm{fv}(t)$, is defined as follows:

$$
\begin{array}{lcl}
\mathrm{fv}(x) & = & \{x\} \\
\mathrm{fv}(\lambda x.t) & = & \mathrm{fv}(t)\backslash\{x\} \\
\mathrm{fv}(t_1 t_2) & = & \mathrm{fv}(t_1) \cup \mathrm{fv}(t_2)
\end{array}
$$

Definition 2.1.4. The set of bound variables of t, $\mathrm{bv}(t)$, is defined as follows:

$$
\begin{array}{lcl}
\mathrm{bv}(x) & = & \emptyset \\
\mathrm{bv}(\lambda x.t) & = & \mathrm{bv}(t) \cup \{x\} \\
\mathrm{bv}(t_1 t_2) & = & \mathrm{bv}(t_1) \cup \mathrm{bv}(t_2)
\end{array}
$$

Example 2.1.1. The same variable can occur in two ways, bound and free:

$$(\lambda xy.xzy)y$$

In the expression above, the first occurrence of y, which appears in the body of the subterm $\lambda xy.xzy$, is bound to λy, whereas the second occurrence of y is free.

After the classification of free and bound variables, the definition of the substitution function arises.

Definition 2.1.5 (Substitution)**.** The result of the substitution of free occurrences x in t by u, denoted by $t[u/x]$, is defined as follows:

$$
\begin{array}{lcl}
y[u/x] & = & \begin{cases} u & \text{if } y \equiv x \\ y & \text{otherwise} \end{cases} \\
(\lambda y.t)[u/x] & = & \begin{cases} (\lambda y.t) & \text{if } y \equiv x \\ (\lambda y.t[u/x]) & \text{otherwise} \end{cases} \\
(t_1 t_2)[u/x] & = & (t_1[u/x] t_2[u/x])
\end{array}
$$

Definition 2.1.6 (β-conversion)**.** The main computational rule of λ-calculus is β-conversion.

$$\beta : \underbrace{(\lambda x.t_1)t_2}_{\beta-redex} \quad \to_\beta \quad \underbrace{t_1[t_2/x]}_{\beta-contractum}$$

The expression $(\lambda x.t_1)t_2$, on the left side of the rule, is called a β-*redex* (reducible expression), and the $t_1[t_2/x]$ is its β-*contractum*.

A term t_1 reduces to t_2 if t_2 is obtained by replacing a redex in t_1 with its contractum. Let $\to$ and $\to^*$ be binary relations, then:

- t_1 reduces to t_2 in one step, and it is written like $t_1 \to t_2$.
- t_1 reduces to t_2 in many steps, and it is written like $t_1 \to^* t_2$.

Some care is needed with substitution, as the following example illustrates.

Example 2.1.2. Considering the λ-term $\lambda xy.x$, for any terms t_1 and t_2, the result should be:

$$(\lambda xy.x)t_1t_2 \to^* t_1$$

However, if $(t_1 \equiv y)$, the expression will be:

$$(\lambda xy.x)yt_2 \to^* t_2$$

In this situation, when performing the substitution in the λ-term, the free occurrence of y in t_1 is captured by a bound occurrence of y, hence the variable capture problem arises. In order to avoid this, the substitution in a λ-term should only be made if the bound occurrences of the λ-term are different from the free occurrences of t_1. Thus, if there are variables occurrences in common, one needs to perform α-conversion.

Definition 2.1.7 (α-conversion)**.** The following rule is called α-conversion,

$$\lambda x.t \to_\alpha \lambda y.t[y/x],$$

provided that y does not occur free in t.

Example 2.1.3.

$$(\lambda x.xw)x \to_\alpha (\lambda y.yw)x$$

To avoid the issue of variable capture, the convention of Barendregt [7] will be adopted. This convention will be followed, as it assumes that the sets of free and bound variables are always distinct in any context.

Definition 2.1.8 (η-conversion)**.** The notion of η-conversion is given by the following rule:

$$\lambda x.tx \to_\eta t, \text{ with } x \notin \mathrm{fv}(t)$$

Definition 2.1.9 (Normalization)**.** A λ-term is in normal form if it does not contain any redex as a subterm. A term is normalizable if it reaches a normal form.

A λ-term will reach normal form if all its subterms that are not in normal form are erased by reductions. A strategy that ensures that a normalizable term reaches its normal form is a normalizing strategy.

Definition 2.1.10. The reduction strategy in normal order, denoted by F_L, is defined as follows:

$$F_L \quad = \quad \begin{cases} t_1 & \text{if } t_1 \text{ is in normal form.} \\ t_2 & \text{if } t_1 \to_\beta t_2 \text{, reducing the leftmost redex in } t_1. \end{cases}$$

This reduction strategy is normalizable, i.e., if the term has a normal form, then, when using normal order reduction, it will reach the normal form.

A term may or may not have, and hence, attain a normal form, which is analogous to the execution of a program, for example, which may either reach its end, returning a result, or enter a loop.

2.2 Simple Types

In the previous section, the λ-calculus syntax was presented, along with different notions of reduction. Now, a simple typed formulation, the simple types of λ-calculus, and the Type Inhabitation will be introduced.

For the simple typed formulation, are referred two systems and their properties: the system *à la Curry* and the system *à la Church*.

2.2.1 Types à la Curry

The type system à la Curry assigns elements, of a given set $\mathbb{T}$ of types, to the type-free λ-terms.

Definition 2.2.1. Given an infinite set of type variables $\mathbb{V}$, denoted by X and Y, the set $\mathbb{T}$, of types, is inductively defined as follows:

$$\begin{array}{lcl} X, Y \in \mathbb{V} & \Rightarrow & X, Y \in \mathbb{T} \\ A, B \in \mathbb{T} & \Rightarrow & (A \to B) \in \mathbb{T} \end{array}$$

Notation: The arrow type is right-associative, that is: $A_1 \to A_2 \to \cdots \to A_n = (A_1 \to (A_2 \to \cdots \to (A_{n-1} \to A_n) \ldots))$

Definition 2.2.2. Let t be a λ-term, x a term variable in $\mathbb{V}$, and A a type in $\mathbb{T}$, then:

- A *statement* or a *type-assignment* is of the form $t : A$, that is $t \in A$. It has a *predicate* that is the type A and a *subject*, the λ-term t.
- A *declaration* or an *assumption* is a statement in which the subject is a term variable, i.e., $x : A$.
- A *type-context* or *basis* is a set of *declarations* $\{x_1 : A_1, \ldots, x_n : A_n\}$ with distinct variables as subjects. Γ can be seen as a partial function, such that $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = A_i$, where $1 \leq i \leq n$.
- A *judgment* is of the form $\Gamma \vdash_{\text{Curry}} t : A$, where Γ is a set of assumptions, and it is pronounced like "t has type A given the basis Γ".

Definition 2.2.3 (The simple type system à la Curry)**.** In Curry, a statement $t : A$ given a basis Γ can be produced, if $\Gamma \vdash_{\text{Curry}} t : A$ is obtained from the following inference rules:

$$\begin{array}{lcll} (x : A) \in \Gamma & \Rightarrow & \Gamma \vdash_{\text{Curry}} x : A & \text{(Axiom)} \\ \Gamma, x : A \vdash_{\text{Curry}} t : B & \Rightarrow & \Gamma \vdash_{\text{Curry}} (\lambda x.t) : (A \to B) & \text{(Abs)} \\ \Gamma \vdash_{\text{Curry}} t_1 : (A \to B), \Gamma \vdash_{\text{Curry}} t_2 : A & \Rightarrow & \Gamma \vdash_{\text{Curry}} (t_1 t_2) : B & \text{(App)} \end{array}$$

Notation: The notation $\Gamma, x : A$ represents the set $\Gamma \cup \{x : A\}$, where x does not appear in Γ.

Usually, those are represented through inference derivation rules:

$$\frac{}{\Gamma, x : A \vdash_{\text{Curry}} x : A}\ \text{(Axiom)} \qquad \frac{\Gamma, x : A \vdash_{\text{Curry}} t : B}{\Gamma \vdash_{\text{Curry}} (\lambda x.t) : (A \to B)}\ \text{(Abs)}$$

$$\frac{\Gamma \vdash_{\text{Curry}} t_1 : (A \to B) \qquad \Gamma \vdash_{\text{Curry}} t_2 : A}{\Gamma \vdash_{\text{Curry}} (t_1 t_2) : B}\ \text{(App)}$$

Example 2.2.1. For instance, given the λ-term $\lambda xyz.y(\lambda u.u)$, in which $\Gamma = \{x : X, y : ((Y_1 \to Y_1) \to Y_2), z : Z\}$, the Curry simple type system produces the following derivation:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \vdash_{\text{Curry}} y : ((Y_1 \to Y_1) \to Y_2) \qquad \cfrac{\Gamma \cup \{u : Y_1\} \vdash_{\text{Curry}} u : Y_1}{\Gamma \vdash_{\text{Curry}} (\lambda u.u) : Y_1 \to Y_1}\ \text{(Abs)}}{\{x : X, y : (Y_1 \to Y_1) \to Y_2, z : Z\} \vdash_{\text{Curry}} y(\lambda u.u) : Y_2}\ \text{(App)}}{\{x : X, y : (Y_1 \to Y_1) \to Y_2\} \vdash_{\text{Curry}} \lambda z.y(\lambda u.u) : Z \to Y_2}\ \text{(Abs)}}{\{x : X\} \vdash_{\text{Curry}} \lambda yz.y(\lambda u.u) : ((Y_1 \to Y_1) \to Y_2) \to Z \to Y_2}\ \text{(Abs)}}{\vdash_{\text{Curry}} \lambda xyz.y(\lambda u.u) : X \to ((Y_1 \to Y_1) \to Y_2) \to Z \to Y_2}\ \text{(Abs)}$$

Definition 2.2.4. Consider a basis $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$, then:

- If V_0 is a set of variables, then $\Gamma \upharpoonright V_0 = \{x : \Gamma(x) \mid x \in V_0\}$.
- If $A, B \in \mathbb{T}$, the result of substituting, by B, all occurrences of variable X in A, is denoted by $A[B/X]$. Stretching this substitution notion to a basis, it follows that:

$$\Gamma[B/X] = \{x_1 : A_1[B/X], \ldots, x_n : A_n[B/X]\},\ \text{if } \Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}.$$

Now, some properties of $\lambda \rightarrow$-Curry are presented. The details and proofs can be found in [5].

These first lemmas analyze the importance of a basis to infer a type assignment.

Lemma 2.2.1 (Basis Lemmas)**.** Let Γ be a basis:

- If Γ' is a basis, such that $\Gamma \subseteq \Gamma'$, then $\Gamma \vdash_{\text{Curry}} t : A \Rightarrow \Gamma' \vdash_{\text{Curry}} t : A$.
- If $\Gamma \vdash_{\text{Curry}} t : A$, then $\text{fv}(t) \subseteq dom(\Gamma)$.
- If $\Gamma \vdash_{\text{Curry}} t : A$, then $\Gamma \upharpoonright \text{fv}(t) \vdash_{\text{Curry}} t : A$.

This lemma examines how terms of different forms get typed.

Lemma 2.2.2 (Generation Lemma)**.**

- If $\Gamma \vdash_{\text{Curry}} x : A$, then $(x : A) \in \Gamma$.
- If $\Gamma \vdash_{\text{Curry}} \lambda x.t : C$, then $\exists\, A, B\ [\Gamma \cup \{x : A\} \vdash_{\text{Curry}} t : B$ and $C \equiv (A \rightarrow B)]$.
- If $\Gamma \vdash_{\text{Curry}} t_1 t_2 : B$, then $\exists\, A\ [\Gamma \vdash_{\text{Curry}} t_1 : (A \rightarrow B)$ and $\Gamma \vdash_{\text{Curry}} t_2 : A]$.

The following lemmas hold for substitution.

Lemma 2.2.3 (Substitution Lemmas)**.**

- If $\Gamma \vdash_{\text{Curry}} t : A$, then $\Gamma[B/X] \vdash_{\text{Curry}} t : A[B/X]$.
- If $\Gamma \cup \{x : A\} \vdash_{\text{Curry}} t_1 : B$ and $\Gamma \vdash_{\text{Curry}} t_2 : A$, then $\Gamma \vdash_{\text{Curry}} t_1[t_2/x] : B$.

Proposition 2.2.1 (Typability of subterms)**.** Let t_2 be a subterm of t_1, so if $\Gamma \vdash_{\text{Curry}} t_1 : A$ then $\exists\, \Gamma', A'$: $\Gamma' \vdash_{\text{Curry}} t_2 : A'$, that is, if t_1 has a type, for some Γ and A, then all the subterms of t_1 also have a type.

Theorem 2.2.1 (Subject reduction theorem)**.** Suppose that t_1 is a λ-term and $t_1 \rightarrow^*_\beta t_2$, then

$$\Gamma \vdash_{\text{Curry}} t_1 : A \Rightarrow \Gamma \vdash_{\text{Curry}} t_2 : A$$

2.2.2 Types à la Church

Both the systems à la Curry and à la Church assign elements, of a given set $\mathbb{T}$ of types, to the type-free λ-terms, however, in addition to that, in the type system à la Church, types are also assigned explicitly to type annotated terms.

Example 2.2.2. For example, in a type system à la Curry, the following statement is achieved: $\vdash_{\text{Curry}} (\lambda x.x) : (A \to A)$, whereas in a type system à la Church, it would be: $\vdash_{\text{Church}} (\lambda x : A.x) : (A \to A)$.

Definition 2.2.5. Given the set $\mathbb{T}$ of types and the set of term variables $\mathbb{V}$, denoted by x, the pseudo-terms $\Lambda_{\mathbb{T}}$, also called type annotated λ-terms, denoted by t_1, t_2, are defined as follow:

$$t \in \Lambda_{\mathbb{T}},\ A \in \mathbb{T} := x \mid t_1 t_2 \mid \lambda x : A.t$$

It should be noted that the syntactic abbreviations, which are used in the λ-calculus, are also used in the typed λ-calculus.

Definition 2.2.6 (The simple type system à la Church)**.** In Church, a statement $t : A$ is derivable from the basis Γ, notated as $\Gamma \vdash_{\text{Church}} t : A$, if it can be produced from the following rules:

$$\begin{array}{lcll}
(x : A) \in \Gamma & \Rightarrow & \Gamma \vdash_{\text{Church}} x : A & \text{(Axiom)} \\
\Gamma, x : A \vdash_{\text{Church}} t : B & \Rightarrow & \Gamma \vdash_{\text{Church}} (\lambda x : A.t) : (A \to B) & \text{(Abstraction)} \\
\Gamma \vdash_{\text{Church}} t_1 : (A \to B), \Gamma \vdash_{\text{Church}} t_2 : A & \Rightarrow & \Gamma \vdash_{\text{Church}} (t_1 t_2) : B & \text{(Application)}
\end{array}$$

Usually, those are represented through inference derivation rules:

$$\frac{}{\Gamma, x : A \vdash_{\text{Church}} x : A}\,\text{(Axiom)} \qquad \frac{\Gamma, x : A \vdash_{\text{Church}} t : B}{\Gamma \vdash_{\text{Church}} (\lambda x : A.t) : (A \to B)}\,\text{(Abs)}$$

$$\frac{\Gamma \vdash_{\text{Church}} t_1 : (A \to B) \qquad \Gamma \vdash_{\text{Church}} t_2 : A}{\Gamma \vdash_{\text{Church}} (t_1 t_2) : B}\,\text{(App)}$$

Example 2.2.3. For instance, given the λ-term $(\lambda x : (X \to X)(\lambda y : Y.y))(\lambda x : X.x)$, the Church simple type system produces the following derivation:

$$\frac{\dfrac{\dfrac{\{x : X \to X, y : Y\} \vdash_{\text{Church}} y : Y}{\{x : X \to X\} \vdash_{\text{Church}} (\lambda y : Y.y) : (Y \to Y)}\,\text{(Abs)}}{\vdash_{\text{Church}} (\lambda x : (X \to X)(\lambda y : Y.y)) : (X \to X) \to (Y \to Y)}\,\text{(Abs)} \qquad \dfrac{\{x : X\} \vdash_{\text{Church}} x : X}{\vdash_{\text{Church}} (\lambda x : X.x) : (X \to X)}\,\text{(Abs)}}{\vdash_{\text{Church}} (\lambda x : (X \to X)(\lambda y : Y.y))(\lambda x : X.x) : Y \to Y}\,\text{(App)}$$

Definition 2.2.7. On $\Lambda_{\mathbb{T}}$, the binary relations *one-step β-conversion*, denoted by $\to_\beta$, and *many-steps β-conversion*, denoted by $\to_\beta^*$, are generated by the contraction rule:

$$(\lambda x : A.t_1)t_2 \to_\beta t_1[t_2/x].$$

Example 2.2.4.

$$(\lambda x : X.\lambda y : Y.x)(\lambda z : Z.zz) \rightarrow_\beta (\lambda y : Y.\lambda z : Z.zz)$$

Now, some properties of $\lambda \rightarrow$-Church are presented. The details and proofs can be found in [5].

These first lemmas analyze the importance of a basis to infer a type assignment.

Lemma 2.2.4 (Basis Lemmas)**.** Let Γ be a basis:

- If Γ' is a basis, such that $\Gamma \subseteq \Gamma'$, then $\Gamma \vdash_{\text{Church}} t : A \Rightarrow \Gamma' \vdash_{\text{Church}} t : A$.
- If $\Gamma \vdash_{\text{Church}} t : A$, then $\text{fv}(t) \subseteq dom(\Gamma)$.
- If $\Gamma \vdash_{\text{Church}} t : A$, then $\Gamma \restriction \text{fv}(t) \vdash_{\text{Church}} t : A$.

This lemma examine how terms of different forms get typed.

Lemma 2.2.5 (Generation Lemma)**.**

- If $\Gamma \vdash_{\text{Church}} x : A$, then $(x : A) \in \Gamma$.
- If $\Gamma \vdash_{\text{Church}} \lambda x : A.t : C$, then $\exists\, B\ [\Gamma \cup \{x : A\} \vdash_{\text{Church}} t : B$ and $C = (A \rightarrow B)]$.
- If $\Gamma \vdash_{\text{Church}} t_1 t_2 : B$, then $\exists\, A\ [\Gamma \vdash_{\text{Church}} t_1 : (A \rightarrow B)$ and $\Gamma \vdash_{\text{Church}} t_2 : A]$.

The following lemmas hold for substitution.

Lemma 2.2.6 (Substitution Lemmas)**.**

- If $\Gamma \vdash_{\text{Church}} t : A$, then $\Gamma[B/X] \vdash_{\text{Church}} t[B/X] : A[B/X]$.
- If $\Gamma \cup \{x : A\} \vdash_{\text{Church}} t_1 : B$ and $\Gamma \vdash_{\text{Church}} t_2 : A$, then $\Gamma \vdash_{\text{Church}} t_1[t_2/x] : B$.

Proposition 2.2.2 (Typability of subterms)**.** Let t_2 be a subterm of t_1, so if $\Gamma \vdash_{\text{Church}} t_1 : A$ then $\exists\, \Gamma', A'$: $\Gamma' \vdash_{\text{Church}} t_2 : A'$, that is, if t_1 has a type, for some Γ and A, then all the subterms of t_1 also have a type.

Theorem 2.2.2 (Subject reduction theorem)**.** Suppose that t_1 is a λ-term and $t_1 \rightarrow^*_\beta t_2$, then:

$$\Gamma \vdash_{\text{Church}} t_1 : A \Rightarrow \Gamma \vdash_{\text{Church}} t_2 : A.$$

In these next lemmas, the equivalence between different types assigned to the same λ-term can be witnessed.

Lemma 2.2.7 (Uniqueness of type lemmas)**.**

- If $\Gamma \vdash_{\text{Church}} t : A$ and $\Gamma \vdash_{\text{Church}} t : A'$ then $A \equiv A'$.
- If $\Gamma \vdash_{\text{Church}} t_1 : A$, $\Gamma \vdash_{\text{Church}} t_2 : A'$, and $t_1 =_\beta t_2$ then $A \equiv A'$.

Note: When it is clear from the context which type system is used, the annotations $\vdash_{\text{Curry}}$ and $\vdash_{\text{Church}}$ are omitted, therefore, only the $\vdash$ will be used.

Now that the definition of important notions of the systems à la Curry and à la Church is finished, several notions of Type Inhabitaion will be introduced, in order to ease the interpretation of the following chapters.

2.2.3 Type Inhabitation

When it comes to type systems, there are three typical main questions:

- Given a closed term t and a type A, does t have type A, denoted by $\vdash t : A$?.
- Given a closed term t, is there a type A, such that $\vdash t : A$, denoted by $\vdash t :?$.
- Given type A, is there a closed term t, such that $\vdash t : A$, denoted by $\vdash ? : A$.

These problems are respectively known as: *Type Checking*, *Type Inference*, and *Type Inhabitation*.

The solutions to a Type Inhabitation problem $\vdash ? : A$, are called the type inhabitants of type A. This section follows the analysis found in [3] and [4].

Definition 2.2.8. Given a β-normal inhabitant t of type A, there is one type-assignment-deduction that assigns the type A to the λ-term t. In the deduction of the form $\vdash t : A$, for each λ-subterm and variable, it is assigned a type, and the result of this process, during the deduction, is called *typed-term* and denoted by t^A.

mi tw earphones lite

Example 2.2.5. Given a λ-term $t = \lambda xyz.zy$, it follows that:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash_{\text{Curry}} z : B \to C \qquad \Gamma \vdash_{\text{Curry}} y : B}{\Gamma = \{x : A, y : B, z : B \to C\} \vdash_{\text{Curry}} zy : C}\,(\text{App})}{\{x : A, y : B\} \vdash_{\text{Curry}} \lambda z.zy : (B \to C) \to C}\,(\text{Abs})}{\{x : A\} \vdash_{\text{Curry}} \lambda yz.zy : B \to (B \to C) \to C}\,(\text{Abs})}{\vdash_{\text{Curry}} \lambda xyz.zy : A \to B \to (B \to C) \to C}\,(\text{Abs})$$

Let $X = A \to B \to (B \to C) \to C$, for the λ-term $t = \lambda xyz.zy$, then the type inhabitant is:

$$t^X = (\lambda x^A y^B z^{B \to C}.(z^{B \to C} y^B)^C)^{A \to B \to (B \to C) \to C}$$

Note that Type Inhabitation is an inversion of Type Inference, such that, it starts from a type and synthesizes well-typed subterms.

This chapter covers the background of type-directed program synthesis, which is the main focus of this dissertation. It reviews the main concepts of λ-calculus so that it can be interpreted, as well its simple types, with a focus on the two different systems à la Curry and à la Church, and on the analysis of Type Inhabitation.

Chapter 3

Linear Type Systems

Type Inhabitation is the basis for program synthesis, where the goal is to extract code (terms) from specifications (types). There is a lot of work in Type Inhabitation for the λ-calculus [4, 8–11], which corresponds to proof search work for intuitionistic logic, by the Curry-Howard isomorphism [12]. For type systems that deal explicitly with resources, the corresponding logic, through the Curry-Howard isomorphism, is the logic of resources, known as linear logic [13].

This chapter presents two type systems related by the Curry-Howard isomorphism with linear logic. Propositions in linear logic are resources that must be used exactly once. Non-linear propositions, propositions that can be used more than once, can be denoted using the exponential operator !, also called bang.

There are several core-type systems based on linear logic explained in [14–17]. Here, the focus is on the *Graded Linear Types* [1] and *Linear Haskell* [2].

In the following sections, it will be formally introduced the type systems and a top-level implementation in Prolog. The Prolog language was chosen because, in some cases, type derivation is non deterministic, and the Prolog backtracking search engine fits naturally in this framework.

3.1 Graded Linear Types

Graded Linear λ-calculus follows the system à la Curry and it is a core linear functional language, where assumptions are annotated with a grade. These grades are integers describing the use of variables. In this case, they count the number of times the variable is used. For instance, the assumption $x : [A]_3$ means that x can be used, with type A, three times.

Definition 3.1.1 (Grammar of Types)**.** The grammar of types, denoted by A, B, in graded linear types, is defined as follows:

$$A, B = A \multimap B \mid A \otimes B \mid A \oplus B \mid 1 \mid \Box_r A$$

This grammar is constituted by: *linear functions*, denoted by $A \multimap B$, that, when consuming A, produce B; *multiplicative conjunction*, characterized by $A \otimes B$ and *additive disjunction*, denoted by $A \oplus B$, the first one represents both A and B types, and the second represents a choice, i.e., it has to choose either A or B; an *unit* 1, representing the unity; and the *graded modality*, designated by $\Box_r A$, which represents an indexed set of type operators, where r ranges over the elements of an algebra structure, parameterizing the calculus.

Definition 3.1.2 (Grammar of Terms)**.** Given a term, denoted by t, the grammar of terms is defined as follows:

$$
\begin{array}{rll}
t = & x & \text{(Variable)} \\
& \mid \lambda x.t & \text{(Abstraction)} \\
& \mid t_1 t_2 & \text{(Application)} \\
& \mid [t] & \text{(Construct)} \\
& \mid \textbf{let } [x] = t_1 \textbf{ in } t_2 & \text{(Let)} \\
& \mid \langle t_1, t_2 \rangle & \text{(Pair Construct)} \\
& \mid \textbf{let } \langle x_1, x_2 \rangle = t_1 \textbf{ in } t_2 & \text{(Let Pair)} \\
& \mid () & \text{(Empty)} \\
& \mid \textbf{let } () = t_1 \textbf{ in } t_2 & \text{(Let Empty)} \\
& \mid \textbf{inl } t & \text{(Inl)} \\
& \mid \textbf{inr } t & \text{(Inr)} \\
& \mid \textbf{case } t_1 \textbf{ of inl } x_1 \to t_1 | \textbf{inr } x_2 \to t_3 & \text{(Case)}
\end{array}
$$

The first three lines define the λ-calculus, as usual. The *Construct* syntax constructs a term typed with a graded modal type $\Box_r A$, by raising a term t to the graded modality. The **Let** $[x] = t_1$ **in** t_2 eliminates a term typed with a graded modal value t_1, binding a graded variable x in the scope of t_2. The *Pair Construct* syntax constructs a pair, whereas the *Let Pair* syntax destructs the pair. As it is a linear calculus, both components of the pair must be used. The *Empty* syntax is used for the inhabitant of multiplicative unit 1, and the *Let Empty* syntax destructs the inhabitant of multiplicative unit 1. The *Inl* and *Inr* syntaxes tag the elements to be able to indicate where they come from. The *Case* syntax is applied in the sum types to identify the constructors.

Definition 3.1.3 (Grammar of Contexts)**.** Contexts Γ are defined as follows:

$$\Gamma = \emptyset \mid \Gamma, x : A \mid \Gamma, x : [A]_r$$

The contexts can be *empty* $\emptyset$, or sets of *assumptions* that can be linear, denoted by A, used exactly once, or intuitionistic (graded), denoted by $[A]$, used any number of times. However, the intuitionistic assumption is usually denoted by $[A]_r$, to specify the number of times (r) it could be used.

As previously mentioned, $\Gamma, x : A$ means the union of contexts, denoted by $\Gamma \bigcup \{x : A\}$, where x does not appear in Γ.

There are many operations on contexts to capture the non-linear data flow grading.

Definition 3.1.4 (Context Addition)**.** Given Γ_1 and Γ_2, the context addition is defined by ordered cases matching inductively on the structure of Γ_2, as follows:

$$\Gamma_1 + \Gamma_2 \quad = \quad \begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ ((\Gamma_1', \Gamma_1'') + \Gamma_2'), x : [A]_{(r+s)} & \Gamma_2 = \Gamma_2', x : [A]_s \ \wedge \ \Gamma_1 = \Gamma_1', x : [A]_r, \Gamma_1'' \\ (\Gamma_1 + \Gamma_2'), x : A & \Gamma_2 = \Gamma_2', x : A \ \wedge \ x : A \notin \Gamma_1 \end{cases}$$

The context addition, denoted by $\Gamma_1 + \Gamma_2$, combines contexts that come from typing multiple subterms in a rule, and it is undefined if Γ_1 and Γ_2 overlap on their linear assumptions.

Example 3.1.1. For instance, consider a context $\Gamma_1 = \{x : A\}$ and a context $\Gamma_2 = \{x : B\}$, then the context addition between Γ_1 and Γ_2 is undefined since it has the same linear variable associated with different types. Nevertheless, if $\Gamma_1 = \{x : [A]_1, y : [B]_1\}$ and $\Gamma_2 = \{x : [A]_3\}$, the result of this context addition will be $\{x : [A]_{(1+3)}, y : [B]_1\}$.

The context addition $\Gamma_1 + \Gamma_2$ will be used in *App, Let* $\Box$*, Let* 1*, Pair, Let Pair* and *Case* inference rules.

Definition 3.1.5 (Partial great-lower bound of contexts)**.** Assuming that there is an order relation $\sqsubseteq$ defined on the set of grades, where $r \sqcup s$ is the great-lower bound of r and s in $r \sqsubseteq s$, then the great-lower bound of contexts, denoted by $\Gamma_1 \sqcup \Gamma_2$, is defined as follows:

$$\Gamma_1 \sqcup \Gamma_2 \quad = \quad \begin{cases} \emptyset & \Gamma_1 = \emptyset \ \wedge \ \Gamma_2 = \emptyset \\ (\emptyset \sqcup \Gamma_2'), x : [A]_{0 \sqcup s} & \Gamma_1 = \emptyset \ \wedge \ \Gamma_2 = \Gamma_2', x : [A]_s \\ (\Gamma_1' \sqcup (\Gamma_2', \Gamma_2'')), x : A & \Gamma_1 = \Gamma_1', x : A \ \wedge \ \Gamma_2 = \Gamma_2', x : A, \Gamma_2'' \\ (\Gamma_1' \sqcup (\Gamma_2', \Gamma_2'')), x : [A]_{r \sqcup s} & \Gamma_1 = \Gamma_1', x : [A]_r \ \wedge \ \Gamma_2 = \Gamma_2', x : [A]_s, \Gamma_2'' \end{cases}$$

Example 3.1.2. For instance, consider a context $\Gamma_1 = \{x : [A]_2, y : [B]_3, z : [C]_5\}$ and a context $\Gamma_2 = \{x : [A]_4, y : [B]_1\}$, then the great-lower bound of the two contexts Γ_1 and Γ_2 is $\{x : [A]_4, y : [B]_3, z : [C]_5\}$.

The great-lower bound of two contexts $\Gamma_1 \sqcup \Gamma_2$ will be used in *Case* inference rule.

Definition 3.1.6 (Scalar context multiplication)**.** Given a grade r and a context, the scalar context multiplication is defined as follows:

$$r * \emptyset = \emptyset \qquad r * (\Gamma, x : [A]_s) = (r * \Gamma), x : [A]_{(r*s)}$$

Example 3.1.3. For instance, consider a grade $r = 3$ and a context $\Gamma = \{x : [A]_1, y : [B]_2\}$, then the scalar context multiplication is equal to $\{x : [A]_{(3*1)}, y : [B]_{(3*2)}\}$.

The scalar context multiplication will be used in the *Pr* inference rule, which is applied to promote the grade r into the assumptions, through the scalar context multiplication between the grade r and the ambient Γ, that must be graded, $[\Gamma]$.

Definition 3.1.7 (Typing rules of the Graded Linear λ-calculus)**.** The typing rules are defined in Figure 3.1.

$$\frac{}{x : A \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ (\text{Abs}) \qquad \frac{\Gamma_1 \vdash t_1 : A \multimap B \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash t_1 t_2 : B}\ (\text{App})$$

$$\frac{\Gamma \vdash t : A}{\Gamma, [\Delta]_0 \vdash t : A}\ (\text{Weak}) \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma, x : [A]_1 \vdash t : B}\ (\text{Der})$$

$$\frac{[\Gamma] \vdash t : A}{r * [\Gamma] \vdash [t] : \Box_r A}\ (\text{Pr}) \qquad \frac{\Gamma_1 \vdash t_1 : \Box_r A \qquad \Gamma_2, x : [A]_r \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ [x] = t_1\ \textbf{in}\ t_2 : B}\ (\text{Let}\ \Box)$$

$$\frac{}{\emptyset \vdash () : 1}\ (1) \qquad \frac{\Gamma_1 \vdash t_1 : 1 \qquad \Gamma_2 \vdash t_2 : A}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ () = t_1\ \textbf{in}\ t_2 : A}\ (\text{Let 1})$$

$$\frac{\Gamma_1 \vdash t_1 : A \qquad \Gamma_2 \vdash t_2 : B}{\Gamma_1 + \Gamma_2 \vdash \langle t_1, t_2 \rangle : A \otimes B}\ (\text{Pair}) \qquad \frac{\Gamma_1 \vdash t_1 : A \otimes B \qquad \Gamma_2, x_1 : A, x_2 : B \vdash t_2 : C}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ \langle x_1, x_2 \rangle = t_1\ \textbf{in}\ t_2 : C}\ (\text{Let Pair})$$

$$\frac{\Gamma, x : [A]_r, \Gamma' \vdash t : A \qquad r \sqsubseteq s}{\Gamma, x : [A]_s, \Gamma' \vdash t : A}\ (\text{Approx}) \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \textbf{inl}\ t : A \oplus B}\ (\text{Inl}) \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \textbf{inr}\ t : A \oplus B}\ (\text{Inr})$$

$$\frac{\Gamma_1 \vdash t_1 : A \oplus B \qquad \Gamma_2, x_1 : A \vdash t_2 : C \qquad \Gamma_3, x_2 : B \vdash t_3 : C}{\Gamma_1 + (\Gamma_2 \sqcup \Gamma_3) \vdash \textbf{case}\ t_1\ \textbf{of inl}\ x_1 \to t_2 |\ \textbf{inr}\ x_2 \to t_3 : C}\ (\text{Case})$$

Figure 3.1: Typing rules of the Graded Linear Types.

In the typing rules of graded linear λ-calculus, the first three rules type the linear λ-calculus, as usual. The *Weak* rule expresses that assumptions graded by 0 may be discarded. For instance, the $[\Delta]_0$ denotes a context with a set of only graded assumptions, graded by 0. The *Der* (Dereliction) grants that the linear assumptions, denoted by $x : A$, can be converted on graded assumptions $x : [A]_1$, with grade 1.

The *Pr* (Promotion) rule promotes the graded modality into the assumption, by applying the scalar context multiplication between the graded context $[\Gamma]$ and the grade r, making assumptions usable r times, and the *Let* $\Box$ rule removes the graded modal value $\Box_r A$ and converts it into a graded assumption $x : [A]_r$. The *1* rule is used for the inhabitant of multiplicative unit 1, and the *Let 1* rule destructs the inhabitant of multiplicative unit 1. The *Pair* rule adds the contexts that type the subterms of the pair $\langle t_1, t_2 \rangle$, and the *Let Pair* rule types the pair elimination by binding the pair component to linear variables in the body of the term t_2.

The *Approx* (Approximation) rule converts a grade s in a grade r if r approximates s ($r \sqsubseteq s$). The *Inl* and *Inr* rules tag where the elements came from for the sum type $A \oplus B$. The *Case* rule removes the sums by inducing the great-lower bound of the contexts to type the two branches of the **case**.

Example 3.1.4. Given the context $\Gamma = \{y : A\}$, and the term $\mathbf{let}[x] = [y]$ **in** $x : [A]_3$, the typing rules of the graded linear λ-calculus produces the following derivation:

$$\dfrac{\dfrac{\dfrac{\{y : A\} \vdash y : A}{\{y : [A]_1\} \vdash y : [A]_1}\ (\text{Der})}{3 * \{y : [A]_1\} \vdash [y] : \square_3 A}\ (\text{Pr}) \qquad \dfrac{\dfrac{\dfrac{\{x : A\} \vdash x : A}{\{x : [A]_1\} \vdash x : [A]_1 \quad 1 \sqsubseteq 2}\ (\text{Der})}{\{x : [A]_2\} \vdash x : [A]_2 \quad 2 \sqsubseteq 3}\ (\text{Approx})}{\{x : [A]_3\} \vdash x : [A]_3}\ (\text{Approx})}{\{y : [A]_3\} \vdash \mathbf{let}[x] = [y]\ \mathbf{in}\ x : [A]_3}\ (\text{Let}\ \square)$$

3.1.1 Implementation

In this sub-section, the top-level predicates of the implementation in Prolog of some of the previous typing rules algorithm are presented. The complete code implementation of the implemented rules can be consulted in Appendix A.

Type Completion Rule

```
typeC(In_Context,T,A,Out_Context):-
    type(In_Context,T,A),
    completion(In_Context,T,Out_Context).
```

Listing 3.1: Type Completion Rule of the Graded Linear Types.

In *Type Completion* rule, the code implementation (Listing 3.1) receives an input context *In_Context* and a term T, and must return the value of the type A and the value of the output context *Out_Context*. Therefore, it starts to call the predicate *type*, with the input context *In_Context*, the term T and type A, then, after receiving the output of this predicate (the value of type A), it calls the predicate *completion* (Listing 3.6 and Listing 3.7) with the input context *In_Context*, the term T, and the output context *Out_Context*, to receive the value of the output context.

Var Rule

The *Var* rule, represented in Figure 3.1, is implemented in Listing 3.2. It receives an input context *In_Context* and a Variable term X, and must return the value of type A. For that to happen, the first predicate *atom* checks if the X is a term variable, and the second predicate selects the

```
type(In_Context,X,A) :-
    atom(X),
    select((X,A),In_Context,[]).
```

Listing 3.2: Var Rule of the Graded Linear Types.

linear assumption (X, A) of the input context $In\_Context$, returning an empty context, and it is through this selection that it extracts the value of type A that was in the input context.

Note: The *select* predicate, whenever it selects an assumption from some context, removes it from that context and creates a new context equal to the context from which the assumption was removed.

Example 3.1.5. Given the input `typeC([(x,grdAssump(M,a))],x,A,Out_Context)`, the output produced is composed by the graded assumption type `A=grdAssump(M,a)` and the output context `Out_Context=[(x,grdAssump(M,a))]`.

Abs Rule

```
type(In_Context,lam(X,T),impl(A,B)) :-
    type([(X,A)|In_Context],T,B).
```

Listing 3.3: Abs Rule of the Graded Linear Types.

The *Abs* rule, shown in Figure 3.1, is implemented in Listing 3.3. It receives an input context $In\_Context$ and an Abstraction term $lam(X, T)$, and must return the value of type $impl(A, B)$. Thus, the predicate *type* is called with the input context $In\_Context$, extended with a fresh linear assumption (X, A), the term T, and the type B, to receive the value of type B.

Example 3.1.6. Given the input `typeC([],lam(x,lam(y,app(y,x))),A,Out_Context)`, the output produced is composed by the type `A=impl(A1,impl(impl(A1,B1),B1))` and the output context `Out_Context=[]`.

App Rule

The *App* rule, shown in Figure 3.1, is implemented in Listing 3.4. It receives an input context $In\_Context$ and an Application term $app(T1, T2)$, and must return the value of type B. For that to happen, the first predicate *type* has as arguments the new context $In\_Context1$, as input context, the term $T1$, and the type $impl(A, B)$, whose value must be returned. In the third line, the predicate *type* has the new context $In\_Context2$, as input context, the term $T2$, and the type A, whose value must be returned. Finally, it applies the context addition predicate *cntxtAdd*

```
type(In_Context,app(T1,T2),B) :-
    type(In_Context1,T1,impl(A,B)),
    type(In_Context2,T2,A),
    cntxtAdd(In_Context1,In_Context2,In_Context).
```

Listing 3.4: App Rule of the Graded Linear Types.

between the two input contexts $In\_Context1$ and $In\_Context2$, and returns the result in the input context $In\_Context$.

Example 3.1.7. Given the input `typeC([C,(x,impl(a,b))],app(x,y),A,Out_Context)`, the output produced is composed by the part of the input context `C=(y,a)`, the type `A=b`, and the output context `Out_Context=[(y,a),(x,impl(a,b))]`.

Let $\Box$ Rule

```
1 type(In_Context,let(grdTerm(X),T1,T2),B) :-
2     type(In_Context1,T1,grdType(R,A)),
3     type(In_Context2,T2,B),
4     select((X,grdAssump(R,A)),In_Context2,In_Context3),
5     cntxtAdd(In_Context1,In_Context2,In_Context).
```

Listing 3.5: Let Rule of the Graded Linear Types.

The *Let* $\Box$ rule, shown in Figure 3.1, is implemented in Listing 3.5. It receives an input context $In\_Context$ and a Let term $let(grdTerm(X), T1, T2)$, and must return the value of type B. For that to happen, the first predicate *type* has as arguments the new context $In\_context1$, as input context, the term $T1$, and the type $grdType(R, A)$, whose value must be returned. In the third line, the *type* has the new context $In\_Context2$, as input context, the term $T2$, and the type B, whose value must be returned. Then, it selects the graded assumption $(X, grdAssump(R, A))$ of the input context $In\_Context2$ and returns the rest of the input context in the new context $In\_Context3$. Finally, it applies the context addition predicate, *cntxtAdd*, between the two input contexts $In\_Context1$ and $In\_Context2$, and returns the result in the input context $In\_Context$.

Example 3.1.8. Given the input `typeC(C,let(grdTerm(x),y,x),A,Out_Context)`, the output produced is composed by the input context `C=Out_Context`, the graded assumption type `A=grdAssump(M1, A1)`, and the output context `Out_Context=[(x,grdAssump(M1,A1)),(y,grdType(M1,A1))]`.

```
1 completion(In_Context,lam(X,T),Out_Context1):-
2     !,
3     completion([X|In_Context],T,Out_Context),
4     select(X,Out_Context,Out_Context1).
```

Listing 3.6: Completion1 Rule of the Graded Linear Types.

Completion Rule

The *Completion* rule has three ways to proceed. The first way of the procedure is implemented in Listing 3.6. It receives an input context $In\_Context$ and an Abstraction term $lam(X, T)$, and must return the value of the output context $Out\_Context1$. For that to happen, if the term received is of the form $lam(X, T)$, then calls the cut !, which prevents backtracking and finding extra solutions, when finishing this rule. Next, in the third line, the predicate *completion* is called with the input context $In\_Context$, extended with a fresh term X, the term T, and the output context $Out\_Context$, whose value must be returned. Then, it selects the term X of the output context $Out\_Context$ and records the remnant in the output context $Out\_Context1$.

Example 3.1.9. Given the input `completion([(x,impl(a,b)),(y,b)],lam(x,lam(y,x)),Out_Context)`, the output produced is composed by the output context `Out_Context=[(x,impl(a,b)),(y,b)]`.

```
completion(In_Context,app(T,U),Out_Context2):-
    !,
    completion(In_Context,T,Out_Context1),
    completion(In_Context,U,Out_Context2).
completion(In_Context,X,In_Context).
```

Listing 3.7: Completion2 Rule of the Graded Linear Types.

When the previous rule fails, before reaching the cut !, it then passes to the next two implemented in Listing 3.7. In the first rule of Listing 3.7, it receives an input context $In\_Context$ and an Application term $app(T, U)$, and must return the value of the output context $Out\_Context2$. For that to happen, if the term received is of the form TU, it calls the cut !, and then the next two predicates *completion* with the same input context $In\_Context$, each with one term, the first one has the term T, and the second the term U, and with new output contexts, the first has the new output context $Out\_Context1$, the second has the new output context $Out\_Context2$, which values, of the output contexts, must be returned. Finally, the second rule of the Listing 3.7 receives an input context $In\_Context$ and a Variable term X, and returns the input context $In\_Context$.

Example 3.1.10. Given the input `completion([(x,impl(a,b)),(y,b)],app(x,y),Out_Context)`, the output produced is composed by the output context `Out_Context=[(x,impl(a,b)),(y,b)]`.

The Graded Linear Types follows the system à la Curry, where the type-free λ-terms are assigned, with types. However, in the next section, the Linear Haskell, which follows the system à la Church, will be enlightened.

3.2 Linear Haskell

More directly based on linear logic, is the Linear Haskell, a system that extends Haskell with linear types and follows the system à la Church. Here, is defined the subset of Linear Haskell treated in the dissertation.

Definition 3.2.1 (Grammar of Multiplicities)**.** The grammar of multiplicities, denoted by π, μ, in the Linear Haskell, is defined as follows:

$$\pi, \mu = 1 \mid \omega \mid p \mid \pi + \mu \mid \pi \cdot \mu$$

In Linear Haskell, functions are annotated with a multiplicity that defines how many times the function consumes its input, similar to the grade. The multiplicity 1 represents a linear function, i.e., it can consume exactly once its input, and the multiplicity ω represents an unrestricted function, such that it can consume an infinite number of times its input. The p is a multiplicity type parameter, that can be 1, or ω, and there are the sum $\pi + \mu$ and product $\pi \cdot \mu$ of multiplicities. With this definition, it is also defined an algebra with the multiplicities: the $+$ and $\cdot$, which are associative and commutative relations, the 1, which is the unit of the $\cdot$, and the $\cdot$, that distributes over the $+$.

Definition 3.2.2 (Equivalence of Multiplicities)**.** The equivalence given by the algebraic properties was extended, with the following rules:

- The result of $\omega \cdot \omega$ is equal to ω.
- $1 + 1 = 1 + \omega = \omega + \omega = \omega$

Definition 3.2.3 (Grammar of Types)**.** Given a set of type variables $\mathbb{V}$, denoted by X and Y, the grammar of types $\mathbb{T}$, denoted by A, B, in the Linear Haskell is defined as follows:

$$A, B = X \mid A \to_{\pi} B$$

This grammar has the types with multiplicity-annotated arrows.

Definition 3.2.4 (Grammar of Terms)**.** Given a term, denoted by e, s, t, or u, the grammar of terms is defined as follows:

$$\begin{aligned} e, s, t, u = \;& x && (Variable) \\ & \mid \lambda_\pi(x : A).t && \text{(Abstraction)} \\ & \mid ts && \text{(Application)} \\ & \mid \mathbf{let}_\pi\ x_1 : A_1 = t_1 \ldots x_n : A_n = t_n\ \mathbf{in}\ u && \text{(Let)} \end{aligned}$$

In the grammar of terms, the *Abstraction* is now annotated with its multiplicity π, and the *Let* syntax, which is annotated with multiplicity π, performs pattern matching.

Definition 3.2.5 (Grammar of Contexts)**.** Given a judgment of the form $\Gamma \vdash t : A$, Γ ranges over contexts, and their grammar is defined as follows:

$$\Gamma = x :_\mu A, \Gamma \mid \breve{}$$

The judgment $\Gamma \vdash t : A$ implies that the term $t : A$ can be consumed only once. However, the judgment $\Gamma \vdash x :_\mu A$ will consume each binding $x :_\mu A$ in Γ, with multiplicity μ. An empty context is denoted by $\breve{}$.

Definition 3.2.6 (Context Addition)**.** Given two contexts, the context addition is defined as follows:

$$\begin{aligned} (x :_\pi A, \Gamma) + (x :_\mu A, \Delta) &= x :_{\pi+\mu} A, (\Gamma + \Delta) \\ (x :_\pi A, \Gamma) + \Delta &= x :_\pi A, \Gamma + \Delta \qquad (\text{if } x \notin \Delta) \\ () + \Delta &= \Delta \end{aligned}$$

Example 3.2.1. For instance, consider a context $\Gamma_1 = \{x : [A]_\omega, y : [B]_1\}$ and a context $\Gamma_2 = \{x : [A]_1\}$, the context addition between Γ_1 and Γ_2 is $\{x : [A]_\omega, y : [B]_1\}$.

The scalar context multiplication will be used in *Var, App* and *Let* inference rules.

Definition 3.2.7 (Context Scaling)**.** Given a multiplicity and a context, the context scaling is defined as follows:

$$\pi(x :_\mu A, \Gamma) \quad = \quad x :_{\pi\mu} A, \pi\Gamma$$

Example 3.2.2. For instance, consider a multiplicity $\pi = 1$ and a context $\Gamma = \{x : [A]_\omega, y : [B]_1\}$, then the context scaling is $\{x : [A]_\omega, y : [B]_1\}$, but if $\pi = \omega$ for the same context Γ, then the context scaling is equal to $\{x : [A]_\omega, y : [B]_\omega\}$.

The scalar context multiplication will be used in *Var, App* and *Let* inference rules.

Definition 3.2.8. The operations of addition and multiplication of multiplicities and contexts are defined as follows:

$$\begin{aligned} \Gamma + \Delta &= \Delta + \Gamma & (\pi + \mu)\Gamma &= \pi\Gamma + \mu\Gamma & 1\Gamma &= \Gamma \\ \pi(\Gamma + \Delta) &= \pi\Gamma + \pi\Delta & (\pi\mu)\Gamma &= \pi(\mu\Gamma) \end{aligned}$$

Definition 3.2.9 (Typing rules of the Linear Haskell)**.** The typing rules are defined in Figure 3.2.

$$\frac{}{\omega\Gamma + x :_1 A \vdash x : A}\ (\text{Var}) \qquad \frac{\Gamma, x :_\pi A \vdash t : B}{\Gamma \vdash \lambda_\pi(x : A).t : A \to_\pi B}\ (\text{Abs})$$

$$\frac{\Gamma \vdash t : A \to_\pi B \qquad \Delta \vdash u : A}{\Gamma + \pi\Delta \vdash tu : B}\ (\text{App})$$

$$\frac{\Gamma_i \vdash t_i : A_i \qquad \Delta, x_1 :_\pi A_1 \ldots x_n :_\pi A_n \vdash u : C}{\Delta + \pi\sum_i \Gamma_i \vdash \mathbf{let}_\pi\ x_1 : A_1 = t_1 \ldots x_n : A_n = t_n\ \mathbf{in}\ u : C}\ (\text{Let})$$

Figure 3.2: Typing rules of the Linear Haskell.

In the typing rules of Linear Haskell, there is: the *Var* rule, that expresses that contexts may be weakened with variables of multiplicity ω; the *Abs* (Abstraction) rule is explicitly annotated with multiplicity π, in $\lambda_\pi(x : A).t$, and its function is to add to the environment Γ the assumption $(x :_\pi A)$, before checking the body t of the abstraction; and the *App* (Application) rule which consumes t once, yielding the multiplicities in Γ, and u once, yielding the multiplicities in Δ. However, if the multiplicity π on the function arrow $A \to_\pi B$, is ω, then the function consumes its arguments ω times. Thus, all the u free variables are also used with multiplicity ω, represented by scaling the multiplicities in Δ by π, and then add all the multiplicities in Γ and $\pi\Delta$. Lastly, the *Let* rule is a combination of the *Abs* and *App* rules, where each **let** binding is explicitly annotated with its multiplicity.

Example 3.2.3. For instance, given the context $\Gamma = \{y : (A \to_1 B)\}$, and the term $\mathbf{let}_1\ x : (A \to_1 B) = y\ \mathbf{in}\ x : (A \to_1 B)$, the typing rules of the Linear Haskell produces the following derivation:

$$\frac{\{y : (A \to_1 B)\} \vdash y : (A \to_1 B) \qquad \{x :_1 (A \to_1 B)\} \vdash x : (A \to_1 B)}{1\{y : (A \to_1 B)\} \vdash \mathbf{let}_1\ x : (A \to_1 B) = y\ \mathbf{in}\ x : (A \to_1 B)}\ (\text{Let})$$

3.2.1 Implementation

In this sub-section, the top-level predicates of the implementation in Prolog of the previous typing rules algorithm are displayed. The complete code implementation of the implemented rules can be consulted in Appendix B.

Type Completion Rule

In *Type Completion* rule the code implementation, Listing 3.8, does the same as it does in the Type Completion rule of the Graded Linear Types 3.1. This rule calls the predicate rule *type*

```
typeC(In_Context,T,A,Out_Context):-
    type(In_Context,T,A),
    completion(In_Context,T,Out_Context).
```

Listing 3.8: Type Completion Rule of the Linear Haskell.

to get the value of the type argument A, and then it calls the predicate *completion* to get the output context.

Var Rule

```
type(In_Context,X,A) :-
    atom(X),
    !,
    cntxtScale(omega,In_Context2,In_Context1),
    cntxtAdd(In_Context1,[(X,1,A)],In_Context).
```

Listing 3.9: Var Rule of the Linear Haskell.

The *Var* rule, shown in Figure 3.2, is implemented in Listing 3.9. It receives an input context $In\_Context$ and a Variable term X, and must return the value of type A. For that to happen, the first predicate *atom* checks if the X is a variable term, the second it is the cut !, and the third and fourth are context predicates. In the first context predicate, the context scaling *contxtScale* is applied to the multiplicity omega (ω) and to the new input context $In\_Context2$, and the output result is returned in the new context $In\_Context1$. Finally, in the last predicate the context addition *cntxtAdd* is applied between the context $In\_Context1$ and the linear assumption $(X, 1, A)$, and the output result returned is the input context itself, $In\_Context$.

Example 3.2.4. Given the input `typeC([(x,M,a)],x,A,Out_Context)`, the output produced is composed by two results. The first one has the multiplicity `M=1`, the type `A=a`, and the output context `Out_Context=[(x,1,a)]`. The second has the multiplicity `M=omega`, the type `A=a`, and the output context `Out_Context=[(x,omega,a)]`.

Abs Rule

```
type(In_Context,lam(M,(X,A),T),impl(M,A,B)) :-
    !,
    type([(X,M,A)|In_Context],T,B).
```

Listing 3.10: Abs Rule of the Linear Haskell.

The *Abs* rule, shown in Figure 3.2, is implemented in Listing 3.10. It receives an input context $In\_Context$ and an Abstraction term $lam(M,(X,A),T)$, and must return the value of type $impl(M,A,B)$. For that to happen, the first predicate it is the cut !, and then, the predicate *type* is called with the input context $In\_Context$, extended with a fresh assumption (X,M,A), the term T, and the type B, whose value must be returned.

Example 3.2.5. Given the input
`typeC([],lam(1,(x,B),lam(1,(y,b),app(x,y))),A,Out_Context)`, the output produced is composed by the type `B=impl(1,b,A1)`, the type
`A=impl(1,impl(1,b,A1),impl(1,b,A1))`, and the output context
`Out_Context=[]`.

App Rule

```
type(In_Context,app(T,U),B) :-
    type(In_Context1,T,impl(M,A,B)),
    type(In_Context2,U,A),
    cntxtScale(M,In_Context2,In_Context3),
    cntxtAdd(In_Context1,In_Context3,In_Context).
```

Listing 3.11: App Rule of the Linear Haskell.

The *App* rule, shown in Figure 3.2, is implemented in Listing 3.11. Just like the *App* rule code implementation of the Graded linear Types, Listing 3.4, in the Linear Haskell the *App* rule is just the same, with the exception of its very own characteristics, with regard to the multiplicities. It receives the same arguments, and returns the same argument type. However, the first predicate *type* has as arguments: the new context $In\_Context1$, as input context, the term T, and the type $impl(M,A,B)$, whose value must be returned. In the third line, the predicate *type* has the new context $In\_Context2$, as input context, the term U, and the type A, whose value must be returned. Finally, it applies the context addition predicate, *cntxtAdd*, between the two input contexts $In\_Context1$ and $In\_Context2$, and returns the result in the input context itself, $In\_Context$.

Example 3.2.6. Given the input
`typeC([(x,1,impl(1,a,b)),(y,1,a)],app(x,y),A,Out_Context)`, the output produced is composed by the type `A=b` and the output context
`Out_Context=[(x,1,impl(1,a,b)),(y,1,a)]`.

Let Rule

The *Let* rule, shown in Figure 3.2, is implemented in Listing 3.12. It receives an input context $In\_Context$ and a Let term $let(M,(X,A),T1,U)$, and must return the value of type B. For

```
1 type(In_Context,let(M,(X,A),T1,U),B) :-
2     type(In_Context1,T1,A),
3     type([(X,M,A)|In_Context2],U,B),
4     cntxtScale(M,In_Context1,In_Context3),
5     cntxtAdd(In_Context3,In_Context2,In_Context).
```

Listing 3.12: Let Rule of the Linear Haskell.

that to happen, the first predicate *type* has as arguments: the new context $In\_Context1$, as input context, the term $T1$, and the type A, whose value must be returned. In the third line, the *type* has the new context $In\_Context2$, as input context, extended with a fresh assumption (X, M, A), the term U, and the type B, whose value must be returned. Then, the context scaling, *contxtScale*, is applied to the multiplicity M and the context $In\_Context1$ and the result output is saved in the new context $In\_Context3$. Finally, in the last predicate the context addition, *cntxtAdd*, is applied between the context $In\_Context3$ and the context $In\_Context2$ and the output result returned is the input context itself, $In\_Context$.

Example 3.2.7. Given the input
`typeC(In_Context,let(1,(x,a),y,x),A,Out_Context)`, the output produced is composed by the type `A=a` and the output context and input context:
`Out_Context=[(y,1,a)], In_Context=Out_context`.

Completion Rule

```
1 completion(In_Context,lam(M,(X,A),T),Out_Context1):-
2     !,
3     completion([(X,1,A)|In_Context],T,Out_Context),
4     select((X,1,A),Out_Context,Out_Context1).
5 completion(In_Context,app(T,U),Out_Context2):-
6     !,
7     completion(In_Context,T,Out_Context1),
8     completion(In_Context,U,Out_Context2).
9 completion(In_Context,X,In_Context).
```

Listing 3.13: Completion Rule of the Linear Haskell.

The *Completion* rule of the Linear Haskell, Listing 3.13, has some similarity to the Completion rule at the Graded Linear Types, Listing 3.6 and Listing 3.7, with some exceptions. In the first Completion rule, it receives an input context $In\_Context$ and an Abstraction term $lam(M, (X, A), T)$, and must return the value of the output context $Out\_Context$. For that to happen, if the term received is of the form $lam(M, (X, A), T)$, then calls the cut !. Next, in the third line, the predicate *completion* is called, with the input context $In\_Context$, extended with a fresh linear assumption $(X, 1, A)$, the term T, and the output context *Out_Context*, whose value must be returned. Then, it selects the linear assumption $(X, 1, A)$ of the output context *Out_Context* and records the remnant in the output context *Out_Context*1. In the next two Completion rules, line five and line nine, it follows the Completion rule of the Graded Linear Types, in Listing 3.7.

3.2.2 Limitations

There are still some limitations, in the implementation of the Linear Haskell system, for the Let rule. For instance, for the example given in Example 3.2.7, the output should have been the output mentioned, but, instead of that output, it was an infinite output:

The output is composed of an infinite number of results, where has always the same and correct type `A=a`, but different input contexts and output contexts:

```
In_Context=Out_Context, Out_Context=[(y, 1, a)|_34126],
In_Context=Out_Context, Out_Context=[_34220,  (y, 1, a)|_35380],
In_Context=Out_Context, Out_Context=[_34220, _35378,  (y, 1, a)|_36544],
...
```

Note: The underscore numbers are variables that represent free assumptions.

This happened because of Prolog bactracking, that, sometimes, enters in an infinite loop to search all the output results, and due to time limitations we could not solve this problem yet.

The Linear Type Systems chapter covers the characteristics and top-level implementation of the two linear type systems used in this dissertation. As for these two linear type systems, one (Graded Linear Types system) follows the Type Inference problem since, given the environment and the term, it tries to find the type of that term, while the other (Linear Haskell system) follows the Type Checking problem since, given the environment, the term and the type, it tries to check if that term has that type associated. In the next chapter, it is presented the inverse of the Type Inference problem, the Type Inhabitation problem, where, given the environment and the type, it tries to reach the term of that type.

Chapter 4

Program Synthesis

In this chapter, it is exhibited the main contribution of this dissertation: Terms with Graded Types and Partial Typed Terms. In the first, it is considered some definitions, explained its inference rules, and displayed and explained the top-level implementation. In the Partial Typed Terms, it is presented the top-level code implementation.

4.1 Terms with Graded Types

Terms with Graded Types follow the system à la Curry and have the same grammar as the Graded Linear Types in Section 3.1. So it is a core linear functional language, where assumptions are also annotated with a grade.

In this system for program synthesis given a judgment in form $\Gamma \vdash A \Rightarrow t; \Delta$, the program synthesis receives an input, with a context Γ and a type A, and produces ($\Rightarrow$) the respective output consisting of a term t and a context Δ.

Definition 4.1.1 (Partial least-lower bound of contexts)**.** Assuming that there is an order relation $\sqsubseteq$ defined on the set of grades, where $r \sqcap s$ is the least-lower bound of r and s in $r \sqsubseteq s$, then the least-lower bound of contexts, denoted by $\Gamma_1 \sqcap \Gamma_2$, is defined as follows:

$$\Gamma_1 \sqcap \Gamma_2 \quad = \begin{cases} \emptyset & \Gamma_1 = \emptyset \ \wedge \ \Gamma_2 = \emptyset \\ (\emptyset \sqcap \Gamma_2'), x : [A]_{0 \sqcap s} & \Gamma_1 = \emptyset \ \wedge \ \Gamma_2 = \Gamma_2', x : [A]_s \\ (\Gamma_1' \sqcap (\Gamma_2', \Gamma_2'')), x : A & \Gamma_1 = \Gamma_1', x : A \ \wedge \ \Gamma_2 = \Gamma_2', x : A, \Gamma_2'' \\ (\Gamma_1' \sqcap (\Gamma_2', \Gamma_2'')), x : [A]_{r \sqcap s} & \Gamma_1 = \Gamma_1', x : [A]_r \ \wedge \ \Gamma_2 = \Gamma_2', x : [A]_s, \Gamma_2'' \end{cases}$$

Example 4.1.1. For instance, consider a context $\Gamma_1 = \{y : [B]_4\}$ and a context $\Gamma_2 = \{y : [B]_3, x : [A]_2, z : [C]_2\}$, then the least-lower bound of the two contexts Γ_1 and Γ_2 is $\{y : [B]_3, x : [A]_0, z : [C]_0\}$.

The least-lower bound of two contexts $\Gamma_1 \sqcap \Gamma_2$ will be used in $L \oplus$ inference rule, which is used to synthesize expressions of type **case**.

Definition 4.1.2 (Context subtraction)**.** Given Γ_1 and Γ_2, the context subtraction is defined as follows:

$$\Gamma_1 - \Gamma_2 \quad = \quad \begin{cases} \Gamma_1 & \Gamma_2 = \emptyset \\ (\Gamma_1', \Gamma_1'') - \Gamma_2' & \Gamma_2 = \Gamma_2', x : A \ \wedge \ \Gamma_1 = \Gamma_1', x : A, \Gamma_1'' \\ ((\Gamma_1', \Gamma_1'') - \Gamma_2'), x : [A]_q & \Gamma_2 = \Gamma_2', x : [A]_s \ \wedge \ \Gamma_1 = \Gamma_1', x : [A]_r, \Gamma_1'' \\ & \wedge \ \exists q.r \sqsupseteq q + s \ \wedge \ \forall q'.r \sqsupseteq q' + s \Rightarrow q \sqsupseteq q' \end{cases}$$

The context subtraction, denoted by $\Gamma_1 - \Gamma_2$, quantifies a variable q to express the subtraction result of grades on the right, with those on the left.

Example 4.1.2. For instance, consider a context $\Gamma_1 = \{x : [A]_5\}$ and a context $\Gamma_2 = \{x : [A]_3, z : [C]_2\}$, then the subtraction between the two contexts Γ_1 and Γ_2 is $\{x : [A]_0, z : [C]_2\}$, $\{x : [A]_1, z : [C]_2\}$, $\{x : [A]_2, z : [C]_2\}$.

The context subtraction will be used in $R\,\Box$ inference rule, which is applied to synthesize a promotion $[t]$ for the graded modality type $\Box_r A$, if it is possible to synthesize a linear term t from type A.

Now, the synthesis rules and their explanation are introduced, following the presentation in [1]. Each subterm has a right R rule and left L rule, which introduces the type in the conclusions, or in the hypotheses, respectively, i.e., in *sequent calculus* [18, 19] these R and L rules are like the constructors and the deconstructors. The right rules construct a synthesis to reach the required goal, while the left rules deconstruct the assumptions. The complete system can be found in Figure 4.1.

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow x;\ \Gamma}\ (\text{LinVar}) \qquad \frac{\exists s.r \sqsupseteq s + 1}{\Gamma, x : [A]_r \vdash A \Rightarrow x;\ \Gamma, x : [A]_s}\ (\text{GrVar})$$

The *LinVar* rule verifies if there is a linear assumption $x : A$ in the context input for a given type A, and then, if it is verified, produces an output with the synthesized term x and the context Γ without the x since it has been used.

Example 4.1.3 (LinVar)**.** For instance, consider an input with a context $\Gamma = \{y : B, x : A\}$ and a type A, then the output produced must be a term $t = x$ and the context, without the assumption $x : A$, $\Gamma = \{y : B\}$.

The *GrVar* rule verifies if there is a graded assumption $x : [A]_r$ in the context input given a type A. If verified, tests if there exists a grade s, such that $s + 1$ approximates the grade r, and produces the output, which is composed with the synthesized term x, and the output context that has the context Γ and the new graded assumption context $x : [A]_s$, that can be used s times more.

Example 4.1.4 (GrVar)**.** For instance, consider an input with a context $\Gamma = \{y : [B]_2, x : [A]_3\}$ and a type A, then the output produced is composed by three results, all with the same term $t = x$, but different output contexts (Δ), where, in each result, Δ is equal to $\{x : [A]_0, y : [B]_2\}$, $\{x : [A]_1, y : [B]_2\}$ and $\{x : [A]_2, y : [B]_2\}$, respectively.

$$\frac{\Gamma, x : A \vdash B \Rightarrow t;\ \Delta \qquad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow \lambda x.t;\ \Delta}\ (\mathrm{R} \multimap)$$

For the $R \multimap$ rule the $\lambda x.t$ is synthesized from $A \multimap B$, if t can be synthesized from B, with a fresh linear assumption $x : A$ extending the input context Γ, and to guarantee that the x is used precisely once (linearly) by t, it must not appear in the output context Δ.

Example 4.1.5 (R $\multimap$)**.** For instance, consider an input with a context $\Gamma = \emptyset$ and a type $(A \multimap (B \multimap C)) \multimap (A \multimap (B \multimap C))$, then the output produced is composed by three results, where, in each result, the term t is equal to $\lambda x.x$, $\lambda xyz.xyz$, and $\lambda xy.xy$, respectively, and the output context is always the same through the results, $\Delta = \emptyset$.

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow t_1;\ \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow t_2;\ \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow [(x_1 t_2)/x_2]t_1;\ \Delta_2}\ (\mathrm{L} \multimap)$$

The $L \multimap$ rule synthesizes the term $[(x_1 t_2)/x_2]t_1$ for type C, through two constructions. The first construction synthesizes the term t_1 for type C, having the input context extended with a fresh linear assumption $x_2 : B$, taking into account the result of x_1, which produces the output context Δ_1. To guarantee that the x_2 is used precisely once (linearly) by t_1, it must not appear in the output context Δ_1. In the second construction, the term t_2 is synthesized from type A, under the input context Δ_1. Finally, the term $[(x_1 t_2)/x_2]t_1$, means that the term x_2 is substituted in t_1 by the Application term $x_1 t_2$.

Example 4.1.6 (L $\multimap$)**.** For instance, consider an input with a context $\Gamma = \{x : (A \multimap B), y : A\}$ and a type B, then the output produced is composed by a term $t = xy$, and an output context $\Delta_2 = \emptyset$.

$$\frac{\Gamma, x : [A]_s, y : A \vdash B \Rightarrow t;\ \Delta, x : [A]_{s'} \qquad y \notin |\Delta| \qquad \exists s.r \sqsupseteq s + 1}{\Gamma, x : [A]_r \vdash B \Rightarrow [x/y]t;\ \Delta, x : [A]_{s'}}\ (\mathrm{Der})$$

The *Der* rule synthesizes a term $[x/y]t$ for the goal type B, having the input context extended with the fresh graded assumption $x : [A]_r$. This happens, if it synthesizes a term t for type B, with the input context extended with a fresh graded assumption $x : [A]_s$ and linear assumption $y : A$. The $r \sqsupseteq s + 1$ updates the number of times the term can be used, as it has already been used once, and the term y should not appear in the output context Δ, since it is linear and, therefore, has already been used once.

Example 4.1.7 (Der)**.** For instance, consider an input with a context $\Gamma = \{x : [A]_2, y : B\}$ and a type A, then the output produced is composed by two results, all with the same term $t = x$, but different contexts, where, in each result, Δ is equal to $\{x : [A]_0, y : B\}$ and $\{x : [A]_1, y : B\}$, respectively.

$$\frac{\Gamma \vdash A \Rightarrow t;\ \Delta}{\Gamma \vdash \Box_r A \Rightarrow [t];\ \Gamma - r * (\Gamma - \Delta)}\ (\text{R}\ \Box) \qquad \frac{\Gamma, x_2 : [A]_r \vdash B \Rightarrow t;\ \Delta, x_2 : [A]_s \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow \textbf{let}\ [x_2] = x_1\ \textbf{in}\ t;\ \Delta}\ (\text{L}\ \Box)$$

The $R\ \Box$ rule synthesizes a construct term $[t]$, for the graded modality type $\Box_r A$, if it synthesizes the linear term t for type A, producing the output context Δ for this premise. With the output context Δ, produces the final output context applying the context subtraction between context Γ and the scalar context multiplication among the grade r and the context subtraction $\Gamma - \Delta$.

Example 4.1.8 (R $\Box$)**.** For instance, consider an input with a context $\Gamma = \{x : A, y : B\}$ and a graded type $\Box_2 A$, then the output produced is composed by a graded term $t = [x]$, and an output context $\Delta_2 = \{y : B\}$.

The $L\ \Box$ rule synthesizes a term **let** $[x_2] = x_1$ **in** t for type B, with the input context extended with a fresh graded modality $x_1 : \Box_r A$. This happens, if it synthesizes a term t for type B, with the input context extended with a fresh graded assumption $x_2 : [A]_r$, producing an output context Δ extended with the fresh graded assumption $x_2 : [A]_s$, with the premise that $0 \sqsubseteq s$. From this, it returns the output Δ.

Example 4.1.9 (L $\Box$)**.** For instance, consider an input with a context $\Gamma = \{x : \Box_2 A, y : B\}$ and a type A, then the output produced is composed by a term t equals to **let** $[z] = x$ **in** z, and an output context $\Delta = \{y : B\}$.

$$\frac{\Gamma \vdash A \Rightarrow t_1;\ \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow t_2;\ \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle;\ \Delta_2}\ (\text{R}\ \otimes)$$

The $R \otimes$ rule synthesizes the term $\langle t_1, t_2 \rangle$ from the type $A \otimes B$, through two constructions. The first construction synthesizes the term t_1 from A producing an output context Δ_1. The second construction synthesizes the term t_2 from B, with the input context Δ_1, producing the output context Δ_2.

Example 4.1.10 (R $\otimes$)**.** For instance, consider an input with a context $\Gamma = \{x : [A]_2, y : B, z : C\}$ and a type $A \otimes B$, then the output produced is composed by two results, all with the same term $t = \langle x, y \rangle$, but different contexts, where, in each result, Δ_2 is equal to $\{x : [A]_0, z : C\}$ and $\{x : [A]_1, z : C\}$, respectively.

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow t_2;\ \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow \textbf{let}\ \langle x_1, x_2 \rangle = x_3\ \textbf{in}\ t_2;\ \Delta}\ (\text{L}\ \otimes)$$

The $L \otimes$ rule synthesizes the term **let** $\langle x_1, x_2 \rangle = x_3$ **in** t_2 for type C, with the input context extended with a fresh assumption $x_3 : A \otimes B$, through one construction. The construction synthesizes the term t_2 for type C, having the input context extended with the fresh linear assumptions $x_1 : A$ and $x_2 : B$, and producing the output context Δ. To guarantee that the x_1 and x_2 are used precisely once (linearly) by t_2, they must not appear in the output context Δ.

Example 4.1.11 (L $\otimes$)**.** For instance, consider an input with a context $\Gamma = \{x : (A \otimes B), y : B, z : C\}$ and a type $A \otimes B$, then the output produced is composed by two results, where, in each result, the term t is equal to x, and **let** $\langle x_1, x_2 \rangle = x$ **in** $\langle x_1, x_2 \rangle$, respectively, and the output context is always the same through the results, $\Delta = \{y : B, z : C\}$.

$$\frac{}{\Gamma \vdash 1 \Rightarrow ();\ \Gamma}\ (\text{R1}) \qquad \frac{\Gamma \vdash C \Rightarrow t;\ \Delta}{\Gamma, x : 1 \vdash C \Rightarrow \textbf{let}\ () = x\ \textbf{in}\ t;\ \Delta}\ (\text{L1})$$

The *R1* rule synthesizes the term () for the unit type 1 and returns the input context Γ as output context.

Example 4.1.12 (R1)**.** For instance, consider an input with a context $\Gamma = \{x : A, y : B\}$ and a type 1, then the output produced is composed by the term $t = ()$ and the output context $\Gamma = \{x : A, y : B\}$.

The *L1* rule synthesizes the term **let** $() = x$ **in** t from C, with the input context extended with a fresh linear assumption $x : 1$, if t can be synthesized from C, producing the output context Δ.

Example 4.1.13 (L1)**.** For instance, consider an input with a context $\Gamma = \{x : 1, y : B\}$ and a type B, then the output produced is composed by two results. The first one has a term $t = y$ and an output context $\Delta = \{x : 1\}$, and the second one has a term t equal to **let** $() = x$ **in** y and an output context $\Delta = \emptyset$.

$$\frac{\Gamma \vdash A \Rightarrow t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow \textbf{inl}\ t;\ \Delta}\ (\text{R}\ \oplus_1) \qquad \frac{\Gamma \vdash B \Rightarrow t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow \textbf{inr}\ t;\ \Delta}\ (\text{R}\ \oplus_2)$$

The $R \oplus_1$ rule synthesizes the term **inl** t from $A \oplus B$, if t can be synthesized from A (left), producing the output context Δ.

The $R \oplus_2$ rule synthesizes the term **inr** t from $A \oplus B$, if t can be synthesized from B (right), producing the output context Δ.

Example 4.1.14 (R $\oplus_1$ and R $\oplus_2$)**.** For instance, consider an input with a context $\Gamma = \{x : A, y : B\}$ and a type $A \oplus B$, then the output produced is composed by two results. The first one (R $\oplus_1$) has a term $t =$ **inl** x and an output context $\Delta = \{y : B\}$, and the second one (R $\oplus_2$) has a term $t =$ **inr** y and an output context $\Delta = \{x : A\}$.

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow t_1;\ \Delta_1 \qquad \Gamma, x_3 : B \vdash C \Rightarrow t_2;\ \Delta_2 \qquad x_2 \notin |\Delta_1| \qquad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow \textbf{case } x_1 \textbf{ of inl } x_2 \to t_1 |\ \textbf{inr } x_3 \to t_2;\ \Delta_1 \sqcap \Delta_2}\ (\mathrm{L} \oplus)$$

The $L \oplus$ rule synthesizes the term **case** x_1 **of inl** $x_2 \to t_1|$ **inr** $x_3 \to t_2$ for type C, with the input context extended with a fresh linear assumption $x_1 : A \oplus B$, through two constructions. The first construction synthesizes the term t_1 for type C, with the input context extended with a fresh linear assumption $x_2 : A$. Taking into account the result of x_1, this first construction produces the output context Δ_1. To guarantee that the x_2 is used precisely once (linearly) by t_1, it must not appear in the output context Δ_1. In the second construction, the term t_2 is synthesized for type C, with the input context extended with a fresh linear assumption $x_3 : B$. Taking into account the result of x_1, this second construction produces the output context Δ_2. To guarantee that the x_3 is used precisely once (linearly) by t_2, it must not appear in the output context Δ_2. Finally, the output context of this rule is the least-lower bound between the contexts Δ_1 and Δ_2.

Example 4.1.15 (L $\oplus$)**.** For instance, consider an input with a context $\Gamma = \{x : (A \oplus A)\}$ and a type A, then the output produced is composed by a term t equals to **case** x **of inl** $x_2 \to x_2|$ **inr** $x_3 \to x_3$ and an output context $\Delta_1 \sqcap \Delta_2 = \emptyset$.

$$\frac{}{\Gamma, x : A \vdash A \Rightarrow x;\ \Gamma}\ (\mathrm{LinVar}) \qquad \frac{\exists s. r \sqsupseteq s + 1}{\Gamma, x : [A]_r \vdash A \Rightarrow x;\ \Gamma, x : [A]_s}\ (\mathrm{GrVar})$$

$$\frac{\Gamma, x : A \vdash B \Rightarrow t;\ \Delta \qquad x \notin |\Delta|}{\Gamma \vdash A \multimap B \Rightarrow \lambda x.t;\ \Delta}\ (\mathrm{R} \multimap)$$

$$\frac{\Gamma, x_2 : B \vdash C \Rightarrow t_1;\ \Delta_1 \qquad x_2 \notin |\Delta_1| \qquad \Delta_1 \vdash A \Rightarrow t_2;\ \Delta_2}{\Gamma, x_1 : A \multimap B \vdash C \Rightarrow [(x_1 t_2)/x_2]t_1;\ \Delta_2}\ (\mathrm{L} \multimap)$$

$$\frac{\Gamma, x : [A]_s, y : A \vdash B \Rightarrow t;\ \Delta, x : [A]_{s'} \qquad y \notin |\Delta| \qquad \exists s. r \sqsupseteq s + 1}{\Gamma, x : [A]_r \vdash B \Rightarrow [x/y]t;\ \Delta, x : [A]_{s'}}\ (\mathrm{Der})$$

$$\frac{\Gamma \vdash A \Rightarrow t;\ \Delta}{\Gamma \vdash \Box_r A \Rightarrow [t];\ \Gamma - r * (\Gamma - \Delta)}\ (\mathrm{R}\ \Box) \qquad \frac{\Gamma, x_2 : [A]_r \vdash B \Rightarrow t;\ \Delta, x_2 : [A]_s \qquad 0 \sqsubseteq s}{\Gamma, x_1 : \Box_r A \vdash B \Rightarrow \textbf{let } [x_2] = x_1 \textbf{ in } t;\ \Delta}\ (\mathrm{L}\ \Box)$$

$$\frac{\Gamma \vdash A \Rightarrow t_1;\ \Delta_1 \qquad \Delta_1 \vdash B \Rightarrow t_2;\ \Delta_2}{\Gamma \vdash A \otimes B \Rightarrow \langle t_1, t_2 \rangle;\ \Delta_2}\ (\mathrm{R} \otimes)$$

$$\frac{\Gamma, x_1 : A, x_2 : B \vdash C \Rightarrow t_2;\ \Delta \qquad x_1 \notin |\Delta| \qquad x_2 \notin |\Delta|}{\Gamma, x_3 : A \otimes B \vdash C \Rightarrow \textbf{let } \langle x_1, x_2 \rangle = x_3 \textbf{ in } t_2;\ \Delta}\ (\mathrm{L} \otimes)$$

$$\frac{}{\Gamma \vdash 1 \Rightarrow ();\ \Gamma}\ (\mathrm{R1}) \qquad \frac{\Gamma \vdash C \Rightarrow t;\ \Delta}{\Gamma, x : 1 \vdash C \Rightarrow \textbf{let } () = x \textbf{ in } t;\ \Delta}\ (\mathrm{L1})$$

$$\frac{\Gamma \vdash A \Rightarrow t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow \textbf{inl } t;\ \Delta}\ (\mathrm{R} \oplus_1) \qquad \frac{\Gamma \vdash B \Rightarrow t;\ \Delta}{\Gamma \vdash A \oplus B \Rightarrow \textbf{inr } t;\ \Delta}\ (\mathrm{R} \oplus_2)$$

$$\frac{\Gamma, x_2 : A \vdash C \Rightarrow t_1;\ \Delta_1 \qquad \Gamma, x_3 : B \vdash C \Rightarrow t_2;\ \Delta_2 \qquad x_2 \notin |\Delta_1| \qquad x_3 \notin |\Delta_2|}{\Gamma, x_1 : A \oplus B \vdash C \Rightarrow \textbf{case } x_1 \textbf{ of inl } x_2 \to t_1 |\ \textbf{inr } x_3 \to t_2;\ \Delta_1 \sqcap \Delta_2}\ (\mathrm{L} \oplus)$$

Figure 4.1: Synthesis rules.

4.1.1 Implementation

In this sub-section, it is displayed the top level predicates of the implementation in Prolog of the previous synthesis algorithm. The complete code implementation of the implemented rules can be consulted in Appendix C.

LinVar Rule

```
synthesis(In_Context,A,X,Out_Context) :-
    nonvar(A),
    var(X),
    select((X,A),In_Context,Out_Context).
```

Listing 4.1: LinVar Rule.

The *LinVar* rule, shown in Figure 4.1, is implemented in Listing 4.1. It receives an input context $In\_Context$ and a type A, and must return the value of the Variable term X and the value of the output context $Out\_Context$. For that to happen, the predicates begin to verify if the A is a type and X a term variable, and selects the linear assumption (X, A) of the input context $In\_Context$, recording the remnant in the output context $Out\_Context$.

Note: As previously mentioned, the *select* predicate, whenever it selects an assumption from some context, removes it from that context and creates a new context equal to the context from which the assumption was removed.

Example 4.1.16. Given the input `synthesis([(y,b),(x,a)],a,T,Out_Context)`, the output produced is composed by the term `T=x` and the output context `Out_Context=[(y,b)]`.

GrVar Rule

```
synthesis(In_Context,A,T,[(T,grdAssump(S,A))|Out_Context]) :-
    select((T,grdAssump(R,A)),In_Context,Out_Context),
    R #>= S+1,
    S #>= 0,
    indomain(S).
```

Listing 4.2: GrVar Rule.

The *GrVar* rule, shown in Figure 4.1, is implemented in Listing 4.2. It receives an input context $In\_Context$ and a type A, and must return the value of the term T and the value of the output context $Out\_Context$, with the extended fresh graded assumption $(T, grdAssump(S, A))$. For that to happen, the predicate selects the graded assumption $(T, grdAssump(R, A))$, from the input context *In_Context*, recording the remnant in the output context *Out_Context*. Then, it checks if the grade R is greater or equal to the grade $S+1$, and if the new grade S is a positive number. Finally, checks if the grade S is finite and bind the grade S to all the values of his domain on backtracking (*indomain(S)*).

Example 4.1.17. Given the input
`synthesis([(y,grdAssump(2,b)),(x,grdAssump(3,a))],a,T,Out_Context)`, the output produced is composed by three results, all with the same term `T=x`, but different output contexts, where, in each result, the `Out_Context` is equal to
`[(x,grdAssump(0,a)),(y,grdAssump(2,b))]`,
`[(x,grdAssump(1,a)),(y,grdAssump(2,b))]`, and
`[(x,grdAssump(2,a)),(y,grdAssump(2,b))]`, respectively.

R $\multimap$ Rule

```
synthesis(In_Context,impl(A,B),lam(X,T),Out_Context) :-
    synthesis([(X,A)|In_Context],B,T,Out_Context),
    \+(belongs((X,_),Out_Context)).
```

Listing 4.3: R $\multimap$ Rule.

The $R \multimap$ rule, shown in Figure 4.1, is implemented in Listing 4.3. It receives an input context *In_Context* and a type $impl(A, B)$, and must return the value of the Abstraction term $lam(X, T)$ and the value of the output context *Out_Context*. For that to happen, the first predicate *synthesis* is called with the input context *In_Context*, extended with the fresh linear assumption (X, A), the type B, the term T, and the *Out_Context*, whose value, of the last two, must be returned. Then, it checks if an assumption with a term X, for any type, does not appear in the output context *Out_Context*.

Example 4.1.18. Given the input
`synthesis([],impl(impl(a,impl(b,c)),impl(a,impl(b,c))),T,Out_Context)`, the output produced is composed by three results, where, in each result, the term `T` is equal to
`lam(x,x)`, `lam(x,lam(y,lam(z,app(app(x,y),z))))`, and to
`lam(x,lam(y,app(x,y)))`, respectively, and the output context is always the same through the results, `Out_Context=[]`

L $\multimap$ Rule

The $L \multimap$ rule, shown in Figure 4.1, is implemented in Listing 4.4. It receives an input context *In_Context* and a type C, and must return the value of the term T and the value of the output context *Out_Context*. For that to happen, the first predicate selects the assumption

```
1 synthesis(In_Context,C,T,Out_Context) :-
2     select((X1,impl(A,B)),In_Context,In_Context1),
3     synthesis([(X2,B)|In_Context1],C,T1,Out_Context1),
4     var(X2),
5     \+(belongs((X2,_),Out_Context1)),
6     synthesis(Out_Context1,A,T2,Out_Context),
7     subs(X2,app(X1,T2),T1,T).
```

Listing 4.4: L $\multimap$ Rule.

$(X1, impl(A, B))$ from the input context $In\_Context$, recording the remnant in the new context $In\_Context1$. Then, it calls the function *synthesis* with the input context $In\_Context1$, extended with a linear assumption $(X2, B)$, the type C, and return the value of the new term $T1$ and the value of the new output context $Out\_Context1$. In the line 4 and line 5, it verifies if the term $X2$ is a variable, and tests if the assumption with the term $X2$, for any type, does not belong to the $Out\_Context1$. Next, it calls the function *synthesis* with the $Out\_Context1$ as input context, the type A, and returns the value of the new term $T2$ and the value of the output context $Out\_Context$. Finally, the predicate *subs* substitutes the term $X2$ by the application term $app(X1, T2)$, in $T1$ and save it in the term T.

Example 4.1.19. Given the input `synthesis([(x,impl(a,b)),(y,a)],b,T,Out_Context)`, the output produced is composed by the term `T=app(x,y)` and the output context `Out_context=[]`.

Der Rule

```
synthesis(In_Context,B,T,Out_Context) :-
    select((X,grdAssump(R,A)),In_Context,In_Context1),
    R #>= S+1,
    S #>= 0,
    indomain(S),
    synthesis([(Y,A),(X,grdAssump(S,A))|In_Context1],B,T1,
  Out_Context),
    \+(belongs((Y,_),Out_Context)),
    subs(Y,X,T1,T).
```

Listing 4.5: Der Rule.

The *Der* rule, shown in Figure 4.1, is implemented in Listing 4.5. It receives an input context $In\_Context$ and a type B, and must return the value of the term T and the value of the output context $Out\_Context$. For that to happen, the first predicate selects the graded assumption $(X, grdAssump(R, A))$ from the input context $In\_Context$, recording the remnant in the new context $In\_Context1$. The next three lines of predicates (lines 3-5) do the exact same thing that the $GrVar$ rule do in Listing 4.2 (lines 3-5). Next, the predicate *synthesis* is called with the input context $In\_Context$, extended with the linear assumption (Y, A) and with the graded assumption $(X, grdAssump(S, A))$, the type B, and return the value of the new term $T1$ and the value of the output context $Out\_Context$. Then, it verifies if the assumption with the term Y, for any type, does not belong in the output context $Out\_Context$. Finally, the predicate *subs* substitutes the term Y, by the term X in $T1$ and record it in the term T.

Example 4.1.20. Given the input
`synthesis([(x,grdAssump(2,a)),(y,grdAssump(2,b))],a,T,Out_Context)`, the output produced is composed by two results, all with the same term `T=x`, but different output contexts, where, in each result, the `Out_Context` is equal to
`[(x,grdAssump(0,a)),(y,grdAssump(2,b))]`, and
`[(x,grdAssump(1,a)),(y,grdAssump(2,b))]`, respectively.

R □ Rule

```
1 synthesis(In_Context,grdType(R,A),grdTerm(T),Out_Context) :-
2     synthesis(In_Context,A,T,Out_Context1),
3     cntxtSub(In_Context,Out_Context1,Out_Sub),
4     cntxtMult(R,Out_Sub,Out_Mult),
5     cntxtSub(In_Context,Out_Mult,Out_Context).
```

Listing 4.6: R □ Rule.

The $R\,\square$ rule, shown in Figure 4.1, is implemented in Listing 4.6. It receives an input context $In\_Context$ and a graded type $grdType(R, A)$, and must return the value of the Construct term $grdTerm(T)$ and the value of the output context $Out\_Context$. For that to happen, the first predicate *synthesis* it is called with the input context $In\_Context$, the type A, and must return the value of the term T and the value of the new context $Out\_Context1$. Next, the context subtraction predicate, *cntxtSub*, is applied between the input context $In\_Context$ and the output context $Out\_Context1$, and the result is recorded in the new context Out_Sub. In the fourth line, the predicate *cntxtMult* applies the context multiplication between the grade R and the output context Out_Sub, recording the result in the new context Out_Mult. Finally, the predicate *cntxtSub* is used again between the input context $In\_Context$ and the output context Out_Mult, and the result is returned in the output context $Out\_Context$.

Example 4.1.21. Given the input
`synthesis([(x,a),(y,b)],grdType(2,a),T,Out_Context)`, the output produced is composed by the graded term `T=grdTerm(x)` and the output context
`Out_Context=[(y,b)].`

L □ Rule

```
1 synthesis(In_Context,B,let(grdTerm(X2),X1,T),Out_Context) :-
2     select((X1,grdType(R,A)),In_Context,In_Context1),
3     R #>= S+1,
4     S #>= 0,
5     indomain(S),
6     synthesis([(X2,grdAssump(R,A))|In_Context1],B,T,Out_Context1),
7     select((X2,grdAssump(S,A)),Out_Context1,Out_Context).
```

Listing 4.7: L □ Rule.

The $L\ \Box$ rule, shown in Figure 4.1, is implemented in Listing 4.7. It receives an input context $In\_Context$ and a type B, and must return the value of the Let term $let(grdTerm(X2), X1, T)$ and the value of the output context $Out\_Context$. For that to happen, the first predicate selects the graded assumption $(X1, grdType(R, A))$ from the input context $In\_Context$, recording the remnant in the new context $In\_Context1$. The next three lines of predicates (lines 3-5) do the exact same thing that the $GrVar$ rule do in Listing 4.2 (lines 3-5). Next, the predicate *synthesis* is called with the input context $In\_Context1$, extended with the fresh graded assumption $(X2, grdAssump(R, A))$, the type B, and must return the value of the term T and the value of the new context $Out\_Context1$. Finally, the last predicate selects the graded assumption $(X2, grdAssump(S, A))$ of the output context $Out\_Context1$, and returns the remnant of the context in the output context $Out\_Context$.

Example 4.1.22. Given the input `synthesis([(x,grdType(2,a)),(y,b)],a,T,Out_Context)`, the output produced is composed by the term `T=let(grdTerm(x1),x,x1)` and the output context `Out_Context=[(y,b)]`.

R ⊗ Rule

```
synthesis(In_Context,product(A,B),pair(T1,T2),Out_Context) :-
    synthesis(In_Context,A,T1,Out_Context1),
    synthesis(Out_Context1,B,T2,Out_Context).
```

Listing 4.8: R ⊗ Rule.

The $R \otimes$ rule, shown in Figure 4.1, is implemented in Listing 4.8. It receives an input context $In\_Context$ and a product type $product(A, B)$, and must return the value of the Pair term $pair(T1, T2)$ and the value of the output context $Out\_Context$. For that to happen, the first predicate *synthesis* is called with the input context $In\_Context$, the type A, and must return the values of the term $T1$ and the value of the new output context $Out\_Context1$. The second predicate *synthesis* is called with the context $Out\_Context1$, the type B, and must return the value of the term $T2$ and the value of the output context $Out\_Context$.

Example 4.1.23. Given the input
`synthesis([(x,grdAssump(2,a)),(y,b),(z,c)],product(a,b),T,Out_Context)`,
the output produced is composed by two results, all with the same term `T=pair(x,y)`, but different output contexts, where, in each result, `Out_Context` is equal to
`[(x,grdAssump(0,a)),(z,c)]`, and
`[(x,grdAssump(1,a)),(z,c)]`, respectively.

L ⊗ Rule

```
synthesis(In_Context,C,let(pair(X1,X2),X3,T2),Out_Context) :-
    select((X3,product(A,B)),In_Context,In_Context1),
    synthesis([(X1,A),(X2,B)|In_Context1],C,T2,Out_Context),
    \+(belongs((X1,_),Out_Context)),
    \+(belongs((X2,_),Out_Context)).
```

Listing 4.9: L ⊗ Rule.

The $L \otimes$ rule, shown in Figure 4.1, is implemented in Listing 4.9. It receives an input context $In\_Context$ and a type C, and must return the value of the Let Pair term $let(pair(X1, X2), X3, T2)$ and the value of the output context $Out\_Context$. For that to happen, the first predicate selects the linear assumption $(X3, product(A, B))$ from the $In\_Context$, and save the remnant in the new context $In\_Context1$. Next, it calls the predicate *synthesis* with the context $In\_Context1$, extended with the linear assumptions $(X1, A)$ and $(X2, B)$, the type C, and returns the value of the term $T2$ and the value of the output context $Out\_Context$. The last two lines, verify if the assumptions with the terms $X1$ and $X2$, for any types, do not appear in the output context $Out\_Context$.

Example 4.1.24. Given the input
`synthesis([(x,product(a,b)),(y,b),(z,c)],product(a,b),T,Out_Context)`, the output produced is composed by two results, where, in each result, the term `T` is equal to `x`, and `let(pair(x1,x2),x,pair(x1,x2))`, respectively, and the output context is always the same through the results, `Out_Context=[(y,b),(z,c)]`.

R1 Rule

The $R1$ rule, shown in Figure 4.1, is implemented in Listing 4.10. It receives an input context $In\_Context$ and a type *unit*, and must return an Empty term *empty* and the input context itself, $In\_Context$, as output context.

```
synthesis(In_Context,unit,empty,In_Context).
```

Listing 4.10: R1 Rule.

Example 4.1.25. Given the input `synthesis([(x,a),(y,b)],unit,T,Out_Context)`, the output produced is composed by the term `T=empty` and the output context `Out_Context=[(x,a),(y,b)]`.

L1 Rule

```
synthesis(In_Context,C,let(empty,X,T1),Out_Context) :-
    select((X,unit),In_Context,In_Context1),
    synthesis(In_Context1,C,T1,Out_Context).
```

Listing 4.11: L1 Rule.

The $L1$ rule, shown in Figure 4.1, is implemented in Listing 4.11. It receives an input context $In\_Context$ and a type C, and must return the value of the Let Empty term $let(empty, X, T1)$ and the value of the output context $Out\_Context$. For that to happen, the first predicate selects the linear assumption $(X, unit)$ from the input context $In\_Context$ and records the remnant in a new context $In\_Context1$. Next, it calls the *synthesis* with the input context $In\_Context1$, the type C, and returns the value of the term $T1$ and the value of the $Out\_Context$.

Example 4.1.26. Given the input `synthesis([(x,unit),(y,b)],b,T,Out_Context)`, the output produced is composed by two results. The first one has the term `T=y` and the output context `Out_Context=[(x,unit)]`, and the second has the term `T=let(empty,x,y)` and the output context `Out_Context=[]`.

R $\oplus_1$ Rule

```
synthesis(In_Context,or(A,B),inl(T),Out_Context) :-
    synthesis(In_Context,A,T,Out_Context).
```

Listing 4.12: R $\oplus_1$ Rule.

The $R \oplus_1$ rule, shown in Figure 4.1, is implemented in Listing 4.12. It receives an input context $In\_Context$ and a type $or(A, B)$, and must return the value of the Inl term $inl(T)$ and the value of the output context $Out\_Context$. For that to happen, the predicate *synthesis* is called with the $In\_Context$ as the input context, the type A, and must return the value of the term T and the value of the $Out\_Context$.

R $\oplus_2$ Rule

```
synthesis(In_Context,or(A,B),inr(T),Out_Context) :-
    synthesis(In_Context,B,T,Out_Context).
```

Listing 4.13: R $\oplus_2$ Rule.

The $R \oplus_2$ rule, shown in Figure 4.1, is implemented in Listing 4.13. It receives an input context *In_Context* and a type $or(A, B)$, and must return the value of the Inr term $inr(T)$ and the value of the output context *Out_Context*. For that to happen, the predicate *synthesis* is called with the *In_Context* as the input context, the type B, and must return the value of the term T and the value of the *Out_Context*.

Example 4.1.27. Given the input `synthesis([(x,a),(y,b)],or(a,b),T,Out_Context)`, the output produced is composed by two results. The first one has the term `T=inl(x)` and the output context `Out_Context=[(y,b)]`, and the second has the term `T=inr(y)` and the output context `Out_Context=[(x,a)]`.

L $\oplus$ Rule

```
1 synthesis(In_Context,C,case(X1,inl(X2),T1,inr(X3),T2),Out_Context
    ) :-
2     select((X1,or(A,B)),In_Context,In_Context1),
3     synthesis([(X2,A)|In_Context1],C,T1,Out_Context1),
4     synthesis([(X3,B)|In_Context1],C,T2,Out_Context2),
5     \+(belongs((X2,_),Out_Context1)),
6     \+(belongs((X3,_),Out_Context2)),
7     cntxtLowBound(Out_Context1,Out_Context2,Out_Context).
```

Listing 4.14: L $\oplus$ Rule.

The $L \oplus$ rule, shown in Figure 4.1, is implemented in Listing 4.14. It receives an input context *In_Context* and a type C, and must return the value of the Case term $case(X1, inl(X2), T1, inr(X3), T2)$ and the value of the output context *Out_Context*. For that to happen, the first predicate selects the assumption $(X1, or(A, B))$ from the *In_Context* and records the remnant in a new context *In_Context*1. Next, the predicate *synthesis* is called with the input context *In_Context*1, extended with a fresh linear assumption $(X2, A)$, the type C, and must return the value of the term $T1$ and the value of the new context *Out_Context*1. In the fourth line, the predicate *synthesis* is called with the input context *In_Context*1, extended with a fresh linear assumption $(X3, B)$, the type C, and must return the value of the term $T2$, and the value of the new output context *Out_Context*2. In the lines 5 and 6, it verifies if the assumptions with the terms $X2$ and $X3$, for any type, do not appear in the output contexts $Out\_Context1$ and $Out\_Context2$, respectively. Finally, the predicate *cntxtLowBound* applies the *partial least-lower bound of contexts* between the contexts $Out\_Context1$ and $Out\_Context2$ and returns the result in the output context $Out\_Context$.

Example 4.1.28. Given the input `synthesis([(x,or(a,a))],a,T,Out_Context)`, the output produced is composed by the term `T=case(x,inl(x1),x1,inr(x2),x2)`, and the output context `Out_Context=[]`.

The Terms with Graded Types follows the system à la Curry, where the type-free λ-terms are assigned with types, and it is a problem of Type Inhabitation, since, given a type, it tries to find out which term it is associated to that type.

In this dissertation, so far, we have shown the coding of three systems, two of which given a term return the type of that term (Graded Linear Types and Linear Haskell), and in the third program, Terms with Graded Types, given a type returns the term for that type. Thus, for graded types, there is always a way to obtain a type or a term. However, for Linear Haskell, it can only get the type, for a given term, but never the other way around. In the next section we will present an implementation that allows obtaining a term from a given linear type. This is a first attempt to synthesize terms typed in Linear Haskell.

4.2 Partial Typed Terms

This section presents an original work consisting of a synthesis algorithm for a language with Partial Typed Terms, based on Linear Haskell. For this language with Partial Typed Terms, the system will not be presented formally, but there are new syntactic elements.

In this section, the predicates of the implementation in Prolog of this system are presented. The complete code implementation of the implemented rules can be consulted in Appendix B.

This program synthesis implementation is performed for a subset of Linear Haskell. A new type of terms was added, which is the *Hole* term, for which we call the synthesis algorithm for Terms with Graded Types. Therefore, for this subset of Linear Haskell it is possible to get the term for a given type, filling the *Hole* term.

Completion Rule

The forth way of the *Completion* rule of the Linear Haskell 3.2.1, is implemented in Listing 4.15. This procedure is for the terms that are of the form $hole(X, A)$. It receives an input context $In\_Context$ and a new kind of term (a Hole term) $hole(X, A)$, and returns the value of the output context $Out\_Context$. For that to happen, it first calls the *remvInt* and, then, the *eraseMult* predicates. The first one, *remvInt*, is used to remove the integers (multiplicities) of

```
completion(In_Context,hole(X,A),Out_Context):-
    remvInt(In_Context,NewContext),
    eraseMult(A,A1),
    ground(A1),
    !,
    synthesis(NewContext,A1,X,Out_Context).
```

Listing 4.15: Completion Rule of the Linear Haskell.

the input context $In\_Context$ and records the new context in the $New\_Context$, whereas, the second, *eraseMult*, is used to remove the multiplicities specifically from a type A and save the new type in $A1$. After removing all the multiplicities, it runs the predicate *ground*, which checks if the type A is composed by only bound variables. In the next line, the ! certificates that, in the end of this rule predicates, it does not continue running to the next rules. Finally, it calls the function *synthesis* of the Program Synthesis, chapter 4.1.1, with the input context $In\_Context$, the type A, and return the value of the Variable term X and the value of the output context $Out\_Context$.

Example 4.2.1. Given the input `completion([(x,1,impl(1,a,b)),(y,1,b)],lam(1,(x,impl(1,a,b)),lam(1,(y,b),app(hole(X,impl(1,a,b)),y))),Out_Context)`, the output produced is composed by two results, where, in each result the term `X` is equal to `x` , and `lam(z,app(x,z))`, respectively, and the output context is always the same through the results, `Out_Context=[(x,1,impl(1,a,b)),(y,1,b)]`.

Example 4.2.2. Given the input `completion([(x,1,impl(1,a,b)),(y,1,b)],app(hole(X,impl(1,a,b)),y),Out_Context)`, the output produced is composed by two results, where, in each result the term `X` is equal to `x` , and `lam(z,app(x,z))`, respectively, and the output context is always the same through the results, `Out_Context=[(x,1,impl(1,a,b)),(y,1,b)]`.

This rule is included in Linear Haskell, and, therefore, that is why it is called through the Type Completion rule in Listing 3.8, and which consequently executes the Completion rule, that calls the synthesis rules of the Terms with Graded Types, to fill out the term X.

Hole Rule

```
type(In_Context,hole(X,A),A).
```

Listing 4.16: Hole Rule of the Linear Haskell.

The *Hole* rule is implemented in Listing 4.16. It receives an input context $In\_Context$, and a type A, and returns the Hole term $hole(X, A)$, with his arguments values filled. This is only successful if the term in question is a linear term, with multiplicity 1.

Example 4.2.3. Given the input
`typeC([(x,1,impl(1,a,b)),(y,1,a)],hole(X,A),impl(1,a,b),Out_Context)`, the output produced is composed by two results, where, in each result the term `X` is equal to `x`, and `lam(z, app(x,z))`, respectively, and the type is always the same through the results, as well as the output context, `A=impl(1,a,b)` and `Out_Context=[(y,a)]`.

In future work the aim is to arrive at a system, this can be generalized, to fill not only one term, but several. Whenever the programmer easily knows which the multiplicity is, it is not necessary to write the term, in case it has multiplicity 1, i.e, if it is linear.

To summarise, in this chapter it was introduced the Terms with Graded Types system, which follows the problem of Type Inhabitation, since given a type it should find the respective term for that type. It was introduced and explained its inference rules and top-level implementation. It was also presented the Partial Typed Term that follows the same problem type, Type Inhabitation, and fills the "holes" of the Hole term with the respective term for the given type. For that to result, it resorts to the Terms with Graded Types system. For the Partial Typed Terms, it was revealed its top-level code implementation.

Chapter 5

Final Remarks

In this last chapter, we will summary what was done, the main goals and contributions, explaining the main concepts that should have been retained throughout the reading of this dissertation. We finish with the future work.

The study carried out trough this dissertation intended to create synthesis of programs from linear types. For this we began to study some previous works: [2] and [1]. Then we began by studying the systems of Graded Linear Types and starting to implement it, and then make the same, to implement the main system, the system for Terms with Graded Types, which makes use of the syntax of a language of the previous system. Next we studied and implemented the system for Linear Haskell, and finally we implemented Program Synthesis for the Partial Typed Terms.

Through this study, we could verify similarities and differences between different systems. While the type system, denoted by Graded Linear Types, follows the type system à la Curry because it assigns types to the free λ-terms, and has assumptions annotated with grades, denoted by natural numbers that tell how often the assumptions can be used, the type system, denoted by Linear Haskell, follows the type systems à la Church, as types are explicitly assigned to annotated type terms, and use multiplicities, which are annotated in the type arrows and in the Abstract and Let terms. Instead of grades, the multiplicities are denoted by 1, or ω (means an infinite number), to express how many times it can be used. Our implementation of these systems given a term is able to discover which type is associated to that term. Graded Linear Type system was easier to implement, due to its lower complexity, as it has only grade annotated in the assumptions, instead of the other system, which has the multiplicities, which can also be infinite (ω), annotated both in types and in terms.

We also implemented a new notion of Partial Typed Terms, that applied ideas from the Linear Haskell, following the Type Inhabitation problem approach: given a type, it founds the term associated to the given type, and fills the "holes" (typed sub-terms in fault) with the respective term and type.

The main focus of our work was the system with Graded Types, which follows the types à la Curry approach. For this system we also implemented a program synthesis algorithm following the Type Inhabitation problem that given a type, discovers the term associated with that type.

With regard to the Partial Typed Terms, for the time being, only the implementation has been performed. It is not expected to be difficult from the top level of the implementation to do the opposite and get to a formal system, but due to the time limitations, this part was not done.

Another problem occurred due to the use of the Prolog language. Although it is very useful, due to its backtracking framework, it can sometimes be disadvantageous, as it runs backtracking infinitely and ends up in a loop. This was a problem that occurred sometimes, and took time to correct, and ended up subsisting, at least in one case, for lack of time to correct it.

Regarding future studies, there are several ways to improve this dissertation. Still, the main objectives would be to finish rectifying any problems that may exist in the implementation code, and formalize the system and demonstrate the correction for the Partial Typed Terms.

To conclude, taking into account all the positive and negative aspects of this dissertation, I think the result was positive. The main objectives have been completed and the results are in sight. I hope this thesis may help someone that wants to know in more detail systems for the Synthesis of Programs from Linear Types.

Bibliography

[1] J. Hughes and D. Orchard. Resourceful program synthesis from graded linear types. In Maribel Fernández, editor, *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*, volume 12561 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 2020. doi: 10.1007/978-3-030-68446-4_8. URL https://doi.org/10.1007/978-3-030-68446-4_8.

[2] J. P. Bernardy, M. Boespflug, R. R. Newton, S. Peyton Jones, and A. Spiwack. Linear haskell: Practical linearity in a higher-order polymorphic language. *Proc. ACM Program. Lang.*, 2 (POPL), December 2017. doi: 10.1145/3158093. URL https://doi.org/10.1145/3158093.

[3] S. Broda and L. Damas. On long normal inhabitants of a type. *J. Log. Comput.*, 15(3): 353–390, 2005. doi: 10.1093/logcom/exi016. URL https://doi.org/10.1093/logcom/exi016.

[4] S. Broda and L. Damas. Counting a type's principal inhabitants. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings*, volume 1581 of *Lecture Notes in Computer Science*, pages 69–82. Springer, 1999. doi: 10.1007/3-540-48959-2_7. URL https://doi.org/10.1007/3-540-48959-2_7.

[5] H. Barendregt. *Lambda Calculi with Types*, page 117–309. Oxford University Press, Inc., USA, 1993. ISBN 0198537611.

[6] H. Barendregt and E. Barendsen. Introduction to lambda calculus. *Nieuw archief voor wisenkunde*, 4:337–372, 01 1984.

[7] H. Barendregt. The lambda calculus - its syntax and semantics. In *Studies in logic and the foundations of mathematics*, volume 103, page 654. North-Holland, 2nd edition, 1984. ISBN 9780444875082.

[8] C. B. Ben-Yelles. *Type assignment in the lambda-calculus: Syntax and semantics*. PhD thesis, University College of Swansea, UK, 1979.

[9] M. Takahashi, Y. Akama, and S. Hirokawa. Normal proofs and their grammar. *Information and Computation*, 125(2):144–153, 1996. doi: 10.1006/inco.1996.0027. URL https://doi.org/10.1006/inco.1996.0027.

[10] M.W. Bunder. Proof finding algorithms for implicational logics. *Theoretical Computer Science*, 232(1–2):165 – 186, 2000. doi: 10.1016/S0304-3975(99)00174-7. URL https://doi.org/10.1016/S0304-3975(99)00174-7.

[11] P. Urzyczyn. Inhabitation in typed lambda-calculi (A syntactic approach). In *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*, pages 373–389, 1997. doi: 10.1007/3-540-62688-3_47. URL https://doi.org/10.1007/3-540-62688-3_47.

[12] H. B Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584–590, 1934. ISSN 0027-8424. doi: 10.1073/pnas.20.11.584. URL https://www.pnas.org/content/20/11/584.

[13] P. Wadler. A taste of linear logic. In Andrzej M. Borzyszkowski and Stefan Sokolowski, editors, *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93, Gdansk, Poland, August 30 - September 3, 1993, Proceedings*, volume 711 of *Lecture Notes in Computer Science*, pages 185–210. Springer, 1993. doi: 10.1007/3-540-57182-5_12. URL https://doi.org/10.1007/3-540-57182-5_12.

[14] P. Wadler. Linear types can change the world! In Manfred Broy and Cliff B. Jones, editors, *Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990*, page 561. North-Holland, 1990.

[15] M. Ian. Lilac: A functional programming language based on linear logic. *J. Funct. Program.*, 4(4):395–433, 1994. doi: 10.1017/S0956796800001131. URL https://doi.org/10.1017/S0956796800001131.

[16] H. Sören. Linear functional programming. In T. Johnsson, Simon L. Peyton Jones, and K. Karlsson, editors, *Proceedings of the Workshop on Implementation of Lazy Functional Languages*, pages 13–32, 1988.

[17] A. Samson. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111 (1&2):3–57, 1993. doi: 10.1016/0304-3975(93)90181-R. URL https://doi.org/10.1016/0304-3975(93)90181-R.

[18] G. Gentzen. Untersuchungen über das logische schließen. i, 1935. URL https://link.springer.com/article/10.1007/BF01201353#citeas.

[19] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory, Second Edition*, volume 43 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2000. ISBN 978-0-521-77911-1.

Appendix A

Graded Linear Types

The implementation presented is referring to the chapter 3, section 3.1.

```
:- use_module(library(simplex)).
:- use_module(library(pairs)).
:- use_module(library(clpfd)).

%%%%%%%%%%%%%%%%%%%
%   Functions
%%%%%%%%%%%%%%%%%%%
belongsAdd(_,L) :-
    var(L),
    !,
    fail.
belongsAdd((X,_),[(Y,_)|R]) :-
    X == Y,
    !.
belongsAdd(X,[Y|R]) :-
    belongsAdd(X,R).

sortX(C1,C1):-
    var(C1),
    !.
sortX(C1,C):-
    sort(C1,C).
```

```
%%%%%%%%%%%%%%%%%%%
%    Contexts
%%%%%%%%%%%%%%%%%%%
% Addition
cntxtAdd(C1,C2,C3):-
    var(C2),
    !,
    cntxtAdd(C2,C1,C3).
cntxtAdd(C,[],C):- !.
cntxtAdd(C1,C2,[(X,grdAssump(R+S,A))|C3]) :-
    select((X,grdAssump(R,A)),C1,NewC1),
    select((X,grdAssump(S,A)),C2,NewC2),
    !,
    cntxtAdd(NewC1,NewC2,C3).
cntxtAdd(C1,C2,[(X,A)|C3]) :-
    select((X,A),C2,NewC2),
    \+(belongsAdd((X,A),C1)),
    !,
    cntxtAdd(C1,NewC2,C3).
cntxtAdd(C1,C2,C) :-
    select((X,A),C2,NewC2),
    \+(belongsAdd((X,A),C1)),
    !,
    select((X,A),C,C3),
    cntxtAdd(C1,NewC2,C3).
cntxtAdd(C1,C2,C3) :-
    !,
    sortX(C3,NewC3),
    cntxtAdd(C2,C1,NewC3).

%%%%%%%%%%%%%%%%%%%
%    Top Level
%%%%%%%%%%%%%%%%%%%
typeC(In_Context,T,A,Out_Context):-
    type(In_Context,T,A),
    completion(In_Context,T,Out_Context).
```

```
% Completion
completion(In_Context,lam(X,T),Out_Context1):-
    !,
    completion([X|In_Context],T,Out_Context),
    select(X,Out_Context,Out_Context1).
completion(In_Context,app(T,U),Out_Context2):-
    !,
    completion(In_Context,T,Out_Context1),
    completion(In_Context,U,Out_Context2).
completion(In_Context,X,In_Context).

%%%%%%%%%%%%%%%%%%%
%   Typing Rules
%%%%%%%%%%%%%%%%%%%
% Var
type(In_Context,X,A) :-
    atom(X),
    select((X,A),In_Context,[]).

% Abs
type(In_Context,lam(X,T),impl(A,B)) :-
    type([(X,A)|In_Context],T,B).

% App
type(In_Context,app(T1,T2),B) :-
    type(In_Context1,T1,impl(A,B)),
    type(In_Context2,T2,A),
    cntxtAdd(In_Context1,In_Context2,In_Context).

% Let Graded
type(In_Context,let(grdTerm(X),T1,T2),B) :-
    type(In_Context1,T1,grdType(R,A)),
    type(In_Context2,T2,B),
    select((X,grdAssump(R,A)),In_Context2,In_Context3),
    cntxtAdd(In_Context1,In_Context2,In_Context).
```

Appendix B

Linear Haskell

The implementation presented is referring to the chapter 3, section 3.2, and to chapter 4, section 4.2.

```
:- use_module(library(simplex)).
:- use_module(library(pairs)).
:- use_module(library(clpfd)).

:- consult(termsGrTypes).
%%%%%%%%%%%%%%%%%%%
%   Functions
%%%%%%%%%%%%%%%%%%%
multplScale(1,1,1).
multplScale(1,omega,omega).
multplScale(omega,1,omega).
multplScale(omega,omega,omega).

multplAdd(1,1,omega).
multplAdd(1,omega,omega).
multplAdd(omega,1,omega).
multplAdd(omega,omega,omega).
```

```
belongsAdd(_,L) :-
    var(L),
    !,
    fail.
belongsAdd((X,_),[(Y,_)|R]) :-
    X == Y,
    !.
belongsAdd(X,[Y|R]) :-
    belongsAdd(X,R).

sortX(C1,C1):-
    var(C1),
    !.
sortX(C1,[(X,M,A)]):-
    select((X,M,A),C1,C),
    var(C),
    !.
sortX(C1,C):-
    sort(C1,C).

remvInt([], []).
remvInt([(A,X,B)|T], [(A,B)|T2]):-
    atom(B),
    !,
    remvInt(T,T2).
remvInt([(A,X,impl(M,C,D))|T], [(A,impl(C,D))|T2]):-
    remvInt(T,T2).

eraseMult(T,T):-
    var(T),
    !.
eraseMult(impl(M,A,B),impl(A1,B1)):-
    !,
    eraseMult(A,A1),
    eraseMult(B,B1).
eraseMult(T,T).
```

```
%%%%%%%%%%%%%%%%%%%
%   Contexts
%%%%%%%%%%%%%%%%%%%
% Addition
cntxtAdd(C1,C2,C3):-
    sortX(C1,NewC1),
    sortX(C2,NewC2),
    sortX(C3,NewC3),
    cntxtAddX(NewC1,NewC2,NewC3).
cntxtAddX(C1,C2,C3):-
    var(C1),
    !,
    cntxtAddX(C2,C1,C3).
cntxtAddX([],C,C):- !.

cntxtAddX([(X,M1,A)|C1], C2, C) :-
    \+belongsAdd((X,M2,A),C2),
    select((X,M1,A),C,C3),
    cntxtAddX(C1, C2, C3).

cntxtAddX([(X,M1,A)|C1], C2, C) :-
    \+belongsAdd((X,M2,A),C2),
    select((X,M1,A),C,C3),
    cntxtAddX(C1, C2, C3).

cntxtAddX([(X,M1,A)|C1],C2,[(X,omega,A)|C]):-
    select((X,M2,A),C2,C3),
    !,
    multplAdd(M1,M2,omega),
    cntxtAddX(C1,C3,C).

% Scaling
cntxtScale(M,[],[]).
cntxtScale(M,[(X,M1,A)|C],[(X,M2,A)|C2]):-
    multplScale(M,M1,M2),
    cntxtScale(M,C,C2),
    !.

%%%%%%%%%%%%%%%%%%%
%   Top Level
%%%%%%%%%%%%%%%%%%%
typeC(In_Context,T,A,Out_Context):-
    type(In_Context,T,A),
    completion(In_Context,T,Out_Context).
```

```
% Completion
completion(In_Context,hole(X,A),Out_Context):-
    remvInt(In_Context,NewContext),
    eraseMult(A,A1),
    ground(A1),
    !,
    synthesis(NewContext,A1,X,Out_Context).
completion(In_Context,lam(M,(X,A),T),Out_Context1):-
    !,
    completion([(X,1,A)|In_Context],T,Out_Context),
    select((X,1,A),Out_Context,Out_Context1).
completion(In_Context,app(T,U),Out_Context2):-
    !,
    completion(In_Context,T,Out_Context1),
    completion(In_Context,U,Out_Context2).
completion(In_Context,X,In_Context).

%%%%%%%%%%%%%%%%%%%
%   Typing Rules
%%%%%%%%%%%%%%%%%%%
% Var
type(In_Context,X,A) :-
    atom(X),
    !,
    cntxtScale(omega,In_Context2,In_Context1),
    cntxtAdd(In_Context1,[(X,1,A)],In_Context).

% Hole
type(In_Context,hole(X,A),A).

% Abs
type(In_Context,lam(M,(X,A),T),impl(M,A,B)) :-
    !,
    type([(X,M,A)|In_Context],T,B).

% App
type(In_Context,app(T,U),B) :-
    type(In_Context1,T,impl(M,A,B)),
    type(In_Context2,U,A),
    cntxtScale(M,In_Context2,In_Context3),
    cntxtAdd(In_Context1,In_Context3,In_Context).
```

```
% Let
type(In_Context,let(M,(X,A),T1,U),B) :-
    type(In_Context1,T1,A),
    type([(X,M,A)|In_Context2],U,B),
    cntxtScale(M,In_Context1,In_Context3),
    cntxtAdd(In_Context3,In_Context2,In_Context).
```

Appendix C

Terms with Graded Types

The implementation presented is referring to the chapter 4, section 4.1.

```
:- use_module(library(simplex)).
:- use_module(library(pairs)).
:- use_module(library(clpfd)).

%%%%%%%%%%%%%%%%%%%
%   Functions
%%%%%%%%%%%%%%%%%%%
subs(X,Q,X1,Q) :-
    var(X),
    X == X1,
    !.
subs(X,Q,Y,Y):-
    var(X),
    var(Y),
    !.
subs(X,Q,lam(X1,T),lam(X1,T)) :-
    var(X),
    X == X1,
    !.
subs(X,Q,lam(Y,T),lam(Y,TNew)) :-
    var(X),
    !,
    subs(X,Q,T,TNew).
```

```
subs(X,Q,app(M,N),app(MNew,NNew)) :-
    var(X),
    !,
    subs(X,Q,M,MNew),
    subs(X,Q,N,NNew).

belongs((X,_),[(Y,_)|R]) :-
    X == Y,
    !.
belongs(X,[Y|R]) :-
    belongs(X,R).

%%%%%%%%%%%%%%%%%%%
%   Contexts
%%%%%%%%%%%%%%%%%%%

% Subtraction
cntxtSub(T1,[],T1).
cntxtSub([],T2,T2).
cntxtSub(T1,T2,T3) :-
    select((X,A),T1,NewT1),
    select((X,A),T2,NewT2),
    cntxtSub(NewT1,NewT2,T3).
cntxtSub(T1,T2,[(X,grdAssump(Q,A))|T3]) :-
    select((X,grdAssump(R,A)),T1,NewT1),
    select((X,grdAssump(S,A)),T2,NewT2),
    R #>= Q+S,
    (R #>= Q1+S -> Q #>= Q1; fail),
    Q #>= 0,
    indomain(Q),
    cntxtSub(NewT1,NewT2,T3).

% Multiplication
cntxtMult(R,[],[]).
cntxtMult(R,[(X,grdAssump(S,A))|List],[(X,grdAssump(S*R,A))|List1
   ]) :-
    !,
    cntxtMult(R,List,List1).
cntxtMult(R,[(X,A)|List],[(X,A)|List1]) :-
    cntxtMult(R,List,List1).
```

```
% Partial least-lower bound
cntxtLowBound([],[],[]).
cntxtLowBound([],T2,[(X,grdAssump(0,A))|T3]) :-
   select((X,grdAssump(S,A)),T2,NewT2),
   cntxtLowBound([],NewT2,T3).
cntxtLowBound(T1,T2,[(X,A)|T3]) :-
    select((X,A),T1,NewT1),
    select((X,A),T2,NewT2),
    cntxtLowBound(NewT1,NewT2,T3).
cntxtLowBound(T1,T2,[(X,grdAssump(Min,A))|T3]) :-
    select((X,grdAssump(R,A)),T1, NewT1),
    select((X,grdAssump(S,A)),T2, NewT2),
    Min #= min(R,S),
    cntxtLowBound(NewT1,NewT2,T3),
    !.
cntxtLowBound(T1,[],T3) :-
    cntxtLowBound([],T1,T3),
    !.

%%%%%%%%%%%%%%%%%%%
%   Syntax Rules
%%%%%%%%%%%%%%%%%%%
% LinVar
synthesis(In_Context,A,X,Out_Context) :-
    nonvar(A),
    var(X),
    select((X,A),In_Context,Out_Context).

% GrVar
synthesis(In_Context,A,T,[(T,grdAssump(S,A))|Out_Context]) :-
    select((T,grdAssump(R,A)),In_Context,Out_Context),
    R #>= S+1,
    S #>= 0,
    indomain(S).

% R Implication
synthesis(In_Context,impl(A,B),lam(X,T),Out_Context) :-
    synthesis([(X,A)|In_Context],B,T,Out_Context),
    \+(belongs((X,_),Out_Context)).
```

```
% L Implication
synthesis(In_Context,C,T,Out_Context) :-
    select((X1,impl(A,B)),In_Context,In_Context1),
    synthesis([(X2,B)|In_Context1],C,T1,Out_Context1),
    var(X2),
    \+(belongs((X2,_),Out_Context1)),
    synthesis(Out_Context1,A,T2,Out_Context),
    subs(X2,app(X1,T2),T1,T).

% Der
synthesis(In_Context,B,T,Out_Context) :-
    select((X,grdAssump(R,A)),In_Context,In_Context1),
    S #>= 0,
    R #>= S+1,
    indomain(S),
    synthesis([(Y,A),(X,grdAssump(S,A))|In_Context1],B,T1,
   Out_Context),
    \+(belongs((Y,_),Out_Context)),
    subs(Y,X,T1,T).

% R Graded
synthesis(In_Context,grdType(R,A),grdTerm(T),Out_Context) :-
    synthesis(In_Context,A,T,Out_Context1),
    cntxtSub(In_Context,Out_Context1,Out_Sub),
    cntxtMult(R,Out_Sub,Out_Mult),
    cntxtSub(In_Context,Out_Mult,Out_Context).

% L Graded
synthesis(In_Context,B,let(grdTerm(X2),X1,T),Out_Context) :-
    select((X1,grdType(R,A)),In_Context,In_Context1),
    R #>= S+1,
    S #>= 0,
    indomain(S),
    synthesis([(X2,grdAssump(R,A))|In_Context1],B,T,Out_Context1)
   ,
    select((X2,grdAssump(S,A)),Out_Context1,Out_Context).

% R Product
synthesis(In_Context,product(A,B),pair(T1,T2),Out_Context) :-
    synthesis(In_Context,A,T1,Out_Context1),
    synthesis(Out_Context1,B,T2,Out_Context).
```

```
% L product
synthesis(In_Context,C,let(pair(X1,X2),X3,T2),Out_Context) :-
    select((X3,product(A,B)),In_Context,In_Context1),
    synthesis([(X1,A),(X2,B)|In_Context1],C,T2,Out_Context),
    \+(belongs((X1,_),Out_Context)),
    \+(belongs((X2,_),Out_Context)).

% R1
synthesis(In_Context,unit,empty,In_Context).

% L1
synthesis(In_Context,C,let(empty,X,T1),Out_Context) :-
    select((X,unit),In_Context,In_Context1),
    synthesis(In_Context1,C,T1,Out_Context).

% R Or1
synthesis(In_Context,or(A,B),inl(T),Out_Context) :-
    synthesis(In_Context,A,T,Out_Context).

% R Or2
synthesis(In_Context,or(A,B),inr(T),Out_Context) :-
    synthesis(In_Context,B,T,Out_Context).

% L Or
synthesis(In_Context,C,case(X1,inl(X2),T1,inr(X3),T2),Out_Context
   ) :-
    select((X1,or(A,B)),In_Context,In_Context1),
    synthesis([(X2,A)|In_Context1],C,T1,Out_Context1),
    synthesis([(X3,B)|In_Context1],C,T2,Out_Context2),
    \+(belongs((X2,_),Out_Context1)),
    \+(belongs((X3,_),Out_Context2)),
    cntxtLowBound(Out_Context1,Out_Context2,Out_Context).
```