FACULDADE DE CIÊNCIAS – June 2020

DEPARTAMENTO DE CIÊNCIA DE COMPUTADORES

Project Report

Quantitative Type Systems

Fábio Daniel Martins Reis

Supervisors:
Sandra Alves, Mário Florido

Table of Contents

Abstract

Quantitative types are types that provide quantitative information about the typed programs and have been used to obtain time and space complexity measures. Non-idempotent intersection types are an example of such types. Currently there are algorithms that verify if a given number of evaluation steps for a typed λ-expression is correct, but they do not calculate that number when only the expression is given. We came up with a novel type-inference algorithm that uses non-idempotent rank 2 intersection types to obtain the number of evaluation steps of a λ-expression while inferring its type.

1 Introduction

The ability to determine upper bounds for the number of execution steps of a program in compilation time is a relevant problem since it allows us to know in advance the computational resources needed to run the program.
Type systems are a powerful and successful tool of static program analysis that are used, for example, to detect errors in programs before running them. Quantitative type systems provide more than just qualitative information about programs and are particularly useful in contexts where we are interested in measuring the use of resources as they are related to the consumption of time and space in programs. That is the case for non-idempotent intersection types since they bound at the same time the number of evaluation steps and the size of the result, and it has been shown that the number of evaluation steps can be a reasonable time complexity measure.
Our research work led to the creation of a novel type-inference algorithm for the λ-calculus with non-idempotent intersection types that gives the number of evaluation steps of a program, while inferring its type.

A big part of this project consisted in the study of the existing type systems and its type-inference algorithms. I began by studying the untyped and simply typed λ-calculus [Barendregt, 1993], for which I implemented in Haskell an existing type-inference algorithm in order to understand how such an algorithm could be implemented. After that, I studied the intersection type system [Van Bakel, 1993, Jim, 1995] and finally the existing work that makes use of intersection types to obtain quantitative information on programs [Accattoli et al., 2018]. Finally, after studying the necessary background, I implemented an existing type-inference algorithm for the rank 2 intersection type system, which then led to the definition of a new algorithm that gives the number of evaluation steps of the typed λ-expression, which is our contribution to this area. Along with the main project I also implemented a parser for the λ-calculus using Happy.
The code developed during the project can be found in:
`https://github.com/toko18/QuantitativeTypeSystems-Project`

Organization of the report: in 2.1 we present the λ-calculus, a simple type system and a type-inference algorithm for the simply typed λ-calculus, and also cover essential definitions that are used in the entire report; in 2.2 we introduce the intersection types and a corresponding type system; in 3.1 we present a general overview of the final product we developed; in 3.2 we present the rank 2 intersection type system and corresponding type-inference algorithm; and in 3.3 we define our new type-inference algorithm for the rank 2 intersection type system.

2 State of the Art

2.1 Simple Types

We begin by introducing the λ-calculus (first introduced in [Church, 1932]) and some definitions and notations that will be used throughout the report. The definitions in this section can be found in [Barendregt, 1993] and [Hindley, 1997].

Notation We use x,x_1,x_2,... to range over a countable infinite set $\mathcal{V}$ of variables and M,M_1,M_2,... to range over the set Λ of lambda terms.

Definition 2.1.1 (Type-free λ-calculus) The terms of the type-free λ-calculus are defined by the following grammar:

$$M ::= x \mid (M_1 M_2) \mid (\lambda x M),$$

where a term of the form:

x	is called a *term variable*;
$(M_1 M_2)$	is called an *application*;
$(\lambda x M)$	is called an *abstraction*.

Example 2.1.2 Some examples of λ-terms are:

$$x;$$
$$(x_1 x_2);$$
$$(\lambda x_1 (x_1 x_2));$$
$$((\lambda x_1 (x_1 x_2)) x_3);$$
$$((\lambda x_3 ((\lambda x_1 (x_1 x_2)) x_3)) x_4).$$

An application $(M_1 M_2)$ can be seen as a function M_1 being applied to an argument M_2, and an abstraction $(\lambda x M)$ is a function definition and can be interpreted as 'the function that assigns to x the value M'.

Notation We use the following convention that lets us omit parentheses:

- outermost parentheses are not written;
- applications are left associative: $M_1 M_2 \dots M_n$ stands for $(\dots((M_1 M_2) M_3) \dots M_n)$;
- $\lambda x_1 x_2 \dots x_n . M$ stands for $(\lambda x_1 (\lambda x_2 (\dots (\lambda x_n (M)) \dots)))$.

Example 2.1.3 Using this convention, the examples in 2.1.2 may be written as follows:

$$x;$$
$$x_1 x_2;$$
$$\lambda x_1 . x_1 x_2;$$
$$(\lambda x_1 . x_1 x_2) x_3;$$
$$(\lambda x_3 . (\lambda x_1 . x_1 x_2) x_3) x_4.$$

Definition 2.1.4 (Free and bound variables) Every occurrence of a variable in a λ-term is either free or bound. In $\lambda x.M$, every occurrence of x in M is said to be *bound*. An occurrence of a variable is *free* if it is not bound.
The set $FV(M)$ of free variables of M is defined inductively as follows:

$$\begin{aligned} FV(x) &= \{x\}; \\ FV(M_1M_2) &= FV(M_1) \cup FV(M_2); \\ FV(\lambda x.M) &= FV(M) \setminus \{x\}. \end{aligned}$$

M is said to be a *closed λ-term* if it does not contain free variables ($FV(M) = \emptyset$).

Example 2.1.5 In the λ-term $(\lambda x_1x_2.x_1x_2x_3)x_4$, x_3 and x_4 occur as free variables, x_1 and x_2 occur as bound variables and $FV((\lambda x_1x_2.x_1x_2x_3)x_4) = \{x_3, x_4\}$.
In the λ-term $\lambda x_1x_2.x_1x_2x_2$, x_1 and x_2 occur both as bound variables and $FV(\lambda x_1x_2.x_1x_2x_2) = \emptyset$, so this term is closed.
In the λ-term $x_1(\lambda x_1x_2.x_1x_2x_3)$, x_1 and x_3 occur as free variables, x_1 and x_2 occur as bound variables and $FV(x_1(\lambda x_1x_2.x_1x_2x_3)) = \{x_1, x_3\}$. In this case, x_1 occurs both as a free variable (first occurrence) and as a bound variable (second occurrence). With the use of the notation bellow, a case like this one will never happen.

Notation (Variable convention) If M,M_1,...,M_n occur in a certain context (definition, proof, example, etc), then all bound variables are chosen to be different from the free variables.

Computing in the λ-calculus is performed using three conversion rules (α-conversion, β-reduction, η-reduction), which are term-rewriting procedures. We only focus on β-reduction since it is the one we will consider later on for counting evaluation steps of a program, where we can disregard α-conversions since we are using the variable convention described above.

Definition 2.1.6 (Substitution) $M_1[x \coloneqq M_2]$ is the result of substituting the term M_2 for each free occurrence of x in M_1 and can be inductively defined as follows:

$$\begin{aligned} x[x \coloneqq M] &= M; \\ x_1[x_2 \coloneqq M] &= x_1, \text{ if } x_1 \neq x_2; \\ (M_1M_2)[x \coloneqq M_3] &= (M_1[x \coloneqq M_3])(M_2[x \coloneqq M_3]); \\ (\lambda x.M_1)[x \coloneqq M_2] &= \lambda x.M_1; \\ (\lambda x_1.M_1)[x_2 \coloneqq M_2] &= \lambda x_1.(M_1[x_2 \coloneqq M_2]), \text{ if } x_1 \neq x_2. \end{aligned}$$

Example 2.1.7 If we apply the substitution $[x_3 \coloneqq x_5]$ to the first example in 2.1.5, we have:

$$\begin{aligned} ((\lambda x_1x_2.x_1x_2x_3)x_4)[x_3 \coloneqq x_5] &= ((\lambda x_1x_2.x_1x_2x_3)[x_3 \coloneqq x_5])(x_4[x_3 \coloneqq x_5]) \\ &= (\lambda x_1.((\lambda x_2.x_1x_2x_3)[x_3 \coloneqq x_5]))x_4 \\ &= (\lambda x_1x_2.((x_1x_2x_3)[x_3 \coloneqq x_5]))x_4 \\ &= (\lambda x_1x_2.((x_1x_2)[x_3 \coloneqq x_5])(x_3[x_3 \coloneqq x_5]))x_4 \\ &= (\lambda x_1x_2.(x_1[x_3 \coloneqq x_5])(x_2[x_3 \coloneqq x_5])x_5)x_4 \\ &= (\lambda x_1x_2.x_1x_2x_5)x_4 \end{aligned}$$

Definition 2.1.8 (β-reduction) β-reduction captures the notion of function application and the rule states that a term of the form $(\lambda x.M_1)M_2$ (called a *β-redex*) β-reduces to $M_1[x := M_2]$ (its *contractum*), notation:

$$(\lambda x.M_1)M_2 \to_\beta M_1[x := M_2].$$

Definition 2.1.9 (β-normal form) A term is said to be in β-normal form if it cannot be further reduced by the application of the β-reduction rule to its subterms. In other words, if a term does not contain any β-redex, it is said to be in β-normal form.

Example 2.1.10 The term $x_1((\lambda x_2.x_2x_3)x_4)$ is not in normal form since it contains the β-redex $(\lambda x_2.x_2x_3)x_4$. If we apply the β-reduction rule to that β-redex we get

$$(\lambda x_2.x_2x_3)x_4 \to_\beta (x_2x_3)[x_2 := x_4] = x_4x_3,$$

and so $x_1((\lambda x_2.x_2x_3)x_4)$ reduces to $x_1(x_4x_3)$.
The term $x_1(x_4x_3)$ is in β-normal form, since it does not contain any β-redex.

There are two main approaches for introducing types into the λ-calculus: 'à la Curry' (implicit typing paradigm) and 'à la Church' (explicit typing paradigm). We will be focusing on the Curry Type System, which was first introduced in [Curry, 1934] for the theory of combinators, and then modified for the λ-calculus in [Curry et al., 1958].

Notation We use α,α_1,α_2,... to range over a countable infinite set $\mathbb{V}$ of type variables and τ,τ_1,τ_2,... to range over the set $\mathbb{T}_0$ of simple types.

Definition 2.1.11 (Simple types) Simple types are defined by the following grammar:

$$\tau ::= \alpha \mid (\tau_1 \to \tau_2)$$

where a type of the form:

α	is called a *type variable*;
$\tau_1 \to \tau_2$	is called a *functional type*.

Notation Outermost parentheses are not written; by convention, '$\to$' associates to the right: $\tau_1 \to \tau_2 \to \cdots \to \tau_n$ stands for $(\tau_1 \to (\tau_2 \to \cdots \to (\tau_{n-1} \to \tau_n)\ldots))$.

Example 2.1.12 Some examples of simple types are:

$$\alpha;$$
$$\alpha_1 \to \alpha_2;$$
$$\alpha_1 \to \alpha_1 \to \alpha_2;$$
$$(\alpha_1 \to \alpha_1 \to \alpha_2) \to \alpha_3.$$

Definition 2.1.13

- A *statement* is of the form $M : \tau$, where the type τ is called the *predicate*, and the term M is called the *subject* of the statement.
- A *declaration* is a statement where the subject is a term variable.

- A *basis* Γ is a set of declarations where all subjects are distinct.

Definition 2.1.14 If $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is a basis, then:

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

Definition 2.1.15 (Curry Type System) In the Curry Type System, we say that M has type τ given the basis Γ, and write

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

if $\Gamma \vdash_{\mathcal{C}} M : \tau$ can be obtained from the following *derivation rules*:

$$\Gamma \cup \{x : \tau\} \vdash_{\mathcal{C}} x : \tau \qquad \text{(Axiom)}$$

$$\frac{\Gamma_x \cup \{x : \tau_1\} \vdash_{\mathcal{C}} M : \tau_2}{\Gamma_x \vdash_{\mathcal{C}} \lambda x.M : \tau_1 \to \tau_2} \qquad (\to \text{ Intro})$$

$$\frac{\Gamma \vdash_{\mathcal{C}} M_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_{\mathcal{C}} M_2 : \tau_1}{\Gamma \vdash_{\mathcal{C}} M_1 M_2 : \tau_2} \qquad (\to \text{ Elim})$$

Example 2.1.16 For the λ-term $\lambda x_1 x_2.x_1$ the following derivation is obtained:

$$\dfrac{\dfrac{\{x_1 : \tau_1, x_2 : \tau_2\} \vdash_{\mathcal{C}} x_1 : \tau_1}{\{x_1 : \tau_1\} \vdash_{\mathcal{C}} \lambda x_2.x_1 : \tau_2 \to \tau_1}}{\vdash_{\mathcal{C}} \lambda x_1 x_2.x_1 : \tau_1 \to \tau_2 \to \tau_1}$$

And for the λ-term $(\lambda x_1.x_1)x_2$ we obtain:

$$\dfrac{\dfrac{\{x_1 : \tau_2, x_2 : \tau_2\} \vdash_{\mathcal{C}} x_1 : \tau_2}{\{x_2 : \tau_2\} \vdash_{\mathcal{C}} \lambda x_1.x_1 : \tau_2 \to \tau_2} \qquad \{x_2 : \tau_2\} \vdash_{\mathcal{C}} x_2 : \tau_2}{\{x_2 : \tau_2\} \vdash_{\mathcal{C}} (\lambda x_1.x_1)x_2 : \tau_2}$$

Definition 2.1.17 (Type-substitution) We call *type-substitution* to

$$\mathbb{S} = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$$

where $\alpha_1, \ldots, \alpha_n$ are distinct type variables in $\mathbb{V}$ and $\tau_1, \ldots, \tau_n$ are types in $\mathbb{T}_0$. For any τ in $\mathbb{T}_0$, $\mathbb{S}(\tau)$ is the type obtained by simultaneously substituting α_i by τ_i, with $1 \leq i \leq n$, in τ.
The type $\mathbb{S}(\tau)$ is called an *instance* of the type τ.
The notion of type-substitution can be extended to bases in the following way:

$$\mathbb{S}(\Gamma) = \{x_1 : \mathbb{S}(\tau_1), \ldots, x_n : \mathbb{S}(\tau_n)\} \qquad \text{if } \Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

The basis $\mathbb{S}(\Gamma)$ is called an *instance* of the basis Γ.

Example 2.1.18 For $\Gamma = \{x_1 : \alpha_1 \to \alpha_2, x_2 : \alpha_1 \to \alpha_2 \to \alpha_1, x_3 : \alpha_3 \to \alpha_2\}$ and $\mathbb{S} = [\alpha_4/\alpha_1, \alpha_1 \to \alpha_1/\alpha_3]$, we have:

$$\begin{aligned}\mathbb{S}(\Gamma) &= \{x_1 : \mathbb{S}(\alpha_1 \to \alpha_2), x_2 : \mathbb{S}(\alpha_1 \to \alpha_2 \to \alpha_1), x_3 : \mathbb{S}(\alpha_3 \to \alpha_2)\} \\ &= \{x_1 : \alpha_4 \to \alpha_2, x_2 : \alpha_4 \to \alpha_2 \to \alpha_4, x_3 : (\alpha_1 \to \alpha_1) \to \alpha_2\}\end{aligned}$$

Definition 2.1.19 (Principal pair) A *principal pair* for a term M is a pair (Γ, τ) such that:

1. $\Gamma \vdash_{\mathcal{C}} M : \tau$;
2. If $\Gamma' \vdash_{\mathcal{C}} M : \tau'$, then $\exists \mathbb{S}.\ (\mathbb{S}(\Gamma) \subseteq \Gamma'$ and $\mathbb{S}(\tau) = \tau')$.

This definition is generalized for all type systems. A type system is said to have the *principal typing* property if for every term there exists a principal pair.

In the Curry Type System (and in other type systems), the decision problem of *typability* is: 'given a term M, decide whether there exists a basis Γ and a type τ such that $\Gamma \vdash_{\mathcal{C}} M : \tau$'. This problem is decidable and there exists an algorithm that given a term M, returns its principal pair (the Curry Type System has principal typings). Such an algorithm is the *Milner's Type-Inference Algorithm*, presented in [Milner, 1978].
This algorithm, as well as other type-inference algorithms for other type systems that we will discuss later on, is based on the generation and resolution of *constraints* (relations between types). In the case of *Milner's Type-Inference Algorithm*, the constraints are equations, which are solved by using *Robinson's unification algorithm* [Robinson, 1965] for the particular case where the terms we want to unify are simple types. We now describe these algorithms (Haskell implementations of them can be found in Appendix B).

Definition 2.1.20 (Unification problem) A *unification problem* is a finite set of equations $S = \{s_1 = t_1, \ldots, s_n = t_n\}$. A *unifier* (or *solution*) is a substitution $\mathbb{S}$, such that $\mathbb{S}(s_i) = \mathbb{S}(t_i)$, for $i = 1, \ldots, n$. We call $\mathbb{S}(s_i)$ (or $\mathbb{S}(t_i)$) a *common instance* of s_i and t_i. S is *unifiable* if it has at least one unifier. $\mathcal{U}(S)$ is the set of unifiers of S.

Example 2.1.21 The types $\alpha_1 \to \alpha_2 \to \alpha_1$ and $(\alpha_3 \to \alpha_3) \to \alpha_4$ are unifiable. For the substitution $\mathbb{S} = [(\alpha_3 \to \alpha_3)/\alpha_1, \alpha_2 \to (\alpha_3 \to \alpha_3)/\alpha_4]$, the common instance is $(\alpha_3 \to \alpha_3) \to \alpha_2 \to (\alpha_3 \to \alpha_3)$.

Definition 2.1.22 A substitution $\mathbb{S}$ is a *most general unifier* (mgu) of S if $\mathbb{S}$ is a least element of $\mathcal{U}(S)$. That is,

$$\mathbb{S} \in \mathcal{U}(S) \text{ and } \forall \mathbb{S}_1 \in \mathcal{U}(S).\ \exists \mathbb{S}_2.\ \mathbb{S}_1 = \mathbb{S}_2 \circ \mathbb{S}.$$

Example 2.1.23 Consider the types $\tau_1 = (\alpha_1 \to \alpha_1)$ and $\tau_2 = (\alpha_2 \to \alpha_3)$. The substitution $\mathbb{S}' = [(\alpha_4 \to \alpha_5)/\alpha_1, (\alpha_4 \to \alpha_5)/\alpha_2, (\alpha_4 \to \alpha_5)/\alpha_3]$ is a unifier of τ_1 and τ_2, but it is not the mgu.
The mgu of τ_1 and τ_2 is $\mathbb{S} = [\alpha_1/\alpha_2, \alpha_1/\alpha_3]$. The common instance of τ_1 and τ_2 by $\mathbb{S}'$, $(\alpha_4 \to \alpha_5) \to (\alpha_4 \to \alpha_5)$ is an instance of $(\alpha_1 \to \alpha_1)$.

Definition 2.1.24 A unification problem $S = \{\alpha_1 = t_1, \ldots, \alpha_n = t_n\}$ is in *solved form* if $\alpha_1, \ldots, \alpha_n$ are all pairwise distinct variables that do not occur in any of the t_i. In this case, we define $\mathbb{S}_S = [t_1/\alpha_1, \ldots, t_n/\alpha_n]$.

Definition 2.1.25 (Type unification) We define the following relation $\Rightarrow$ on type unification problems:

$$
\begin{array}{lcll}
\{\tau = \tau\} \cup S & \Rightarrow & S & \\
\{\tau_1 \to \tau_2 = \tau_3 \to \tau_4\} \cup S & \Rightarrow & \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup S & \\
\{\tau = \alpha\} \cup S & \Rightarrow & \{\alpha = \tau\} \cup S & \\
\{\alpha = \tau\} \cup S & \Rightarrow & \{\alpha = \tau\} \cup [\tau/\alpha](S) & \text{if } \alpha \in \mathit{fv}(S) \setminus \mathit{fv}(\tau) \\
\{\alpha = \tau\} \cup S & \Rightarrow & \mathit{FAIL} & \text{if } \alpha \in \mathit{fv}(\tau) \text{ and } \alpha \neq \tau
\end{array}
$$

where $[\tau/\alpha](S)$ corresponds to the notion of type-substitution extended to type unification problems. If $S = \{\tau_1 = \tau_1', \ldots, \tau_n = \tau_n'\}$, then $[\tau/\alpha](S) = \{[\tau/\alpha]\tau_1 = [\tau/\alpha]\tau_1', \ldots, [\tau/\alpha]\tau_n = [\tau/\alpha]\tau_n'\}$. And $\mathit{fv}(S)$ and $\mathit{fv}(\tau)$ are the sets of free type variables in S and τ, respectively. Since in our system all occurrences of type variables are free, $\mathit{fv}(S)$ and $\mathit{fv}(\tau)$ are the sets of type variables in S and τ, respectively.

Definition 2.1.26 (Unification algorithm) Let S be a unification problem. The unification function $\mathit{UNIFY}(S)$ is defined as:

function $\mathit{UNIFY}(S)$
  while $S \Rightarrow T$ **do**
    $S := T$;
  if S is in solved form **then**
    return $\mathbb{S}_S$;
  else
    FAIL;

Definition 2.1.27 (Type-inference algorithm) Let Γ be a basis, M a term, τ a type and *UNIFY* the function in Definition 2.1.26. The function $T_{\mathcal{C}}(M) = (\Gamma, \tau)$ defines a type-inference algorithm for the simply typed λ-calculus, in the following way:

1. If $M = x$, then $\Gamma = \{x : \alpha\}$ and $\tau = \alpha$, where α is a new variable;
2. If $M = M_1 M_2$, $T_{\mathcal{C}}(M_1) = (\Gamma_1, \tau_1)$ and $T_{\mathcal{C}}(M_2) = (\Gamma_2, \tau_2)$, then let $\{x_1, \ldots, x_n\} = \mathit{FV}(M_1) \cap \mathit{FV}(M_2)$, such that $x_1 : \delta_1, \ldots, x_n : \delta_n \in \Gamma_1$ and $x_1 : \delta_1', \ldots, x_n : \delta_n' \in \Gamma_2$. Then $T_{\mathcal{C}}(M) = (\mathbb{S}(\Gamma_1 \cup \Gamma_2), \mathbb{S}(\alpha))$, where $\mathbb{S} = \mathit{UNIFY}(\{\delta_1 = \delta_1', \ldots, \delta_n = \delta_n'\} \cup \{\tau_1 = \tau_2 \to \alpha\})$ (α is a new variable);
3. If $M = \lambda x.M_1$ and $T_{\mathcal{C}}(M_1) = (\Gamma_1, \tau_1)$, then:
   (a) If $x \notin \mathit{dom}(\Gamma_1)$, then $T_{\mathcal{C}}(M) = (\Gamma_1, \alpha \to \tau_1)$, where α is a new variable;
   (b) If $\{x : \tau\} \in \Gamma_1$, then $T_{\mathcal{C}}(M) = (\Gamma_1 \setminus \{x : \tau\}, \tau \to \tau_1)$.

2.2 Intersection Types

Even though typability in the Curry Type System is decidable and there is an algorithm that given a term, returns its principal pair, the system has some disadvantages when comparing to others, one of them being the large number of terms that cannot be typed. For example, in the Curry Type System

we cannot assign a type to the λ-term $\lambda x.xx$. This term, on the other hand, can be typed in systems that use Intersection Types, which allow terms to have more than one type. Such a system is the Coppo-Dezani Type System [Coppo and Dezani-Ciancaglini, 1980], which was one of the first to use intersection types, and a basis for subsequent systems.

Definition 2.2.1 (Intersection types) The intersection types $\sigma, \sigma_1, \sigma_2 \cdots \in \mathbb{T}$ of the Coppo-Dezani Type System are defined by the following grammar, where $n \geq 1$:

$$\sigma ::= \alpha \mid \sigma_1 \cap \cdots \cap \sigma_n \to \sigma$$

and $\sigma_1 \cap \cdots \cap \sigma_n$ is called a *sequence*.

Notation The intersection type constructor $\cap$ binds stronger than $\to$: $\alpha_1 \cap \alpha_2 \to \alpha_3$ stands for $(\alpha_1 \cap \alpha_2) \to \alpha_3$.

Example 2.2.2 Some examples of intersection types are:

$$\begin{gathered}\alpha; \\ \alpha_1 \to \alpha_2; \\ \alpha_1 \cap \alpha_2 \to \alpha_3; \\ (\alpha_1 \cap \alpha_2 \to \alpha_3) \to \alpha_4; \\ \alpha_1 \cap (\alpha_1 \to \alpha_2) \to \alpha_3.\end{gathered}$$

Definition 2.2.3 (Coppo-Dezani Type System) In the Coppo-Dezani Type System, we say that M has type σ given the basis Γ (where the predicates of declarations are sequences), and write

$$\Gamma \vdash_{\mathcal{CD}} M : \sigma,$$

if $\Gamma \vdash_{\mathcal{CD}} M : \sigma$ can be obtained from the following *derivation rules*, where $n \geq 1$ and $1 \leq i \leq n$:

$$\Gamma \cup \{x : \sigma_1 \cap \cdots \cap \sigma_n\} \vdash_{\mathcal{CD}} x : \sigma_i \qquad \text{(Axiom)}$$

$$\frac{\Gamma_x \cup \{x : \sigma_1 \cap \cdots \cap \sigma_n\} \vdash_{\mathcal{CD}} M : \sigma}{\Gamma_x \vdash_{\mathcal{CD}} \lambda x.M : \sigma_1 \cap \cdots \cap \sigma_n \to \sigma} \qquad (\to \text{Intro})$$

$$\frac{\Gamma \vdash_{\mathcal{CD}} M_1 : \sigma_1 \cap \cdots \cap \sigma_n \to \sigma \qquad \Gamma \vdash_{\mathcal{CD}} M_2 : \sigma_1 \cdots \Gamma \vdash_{\mathcal{CD}} M_2 : \sigma_n}{\Gamma \vdash_{\mathcal{CD}} M_1 M_2 : \sigma} \qquad (\to \text{Elim})$$

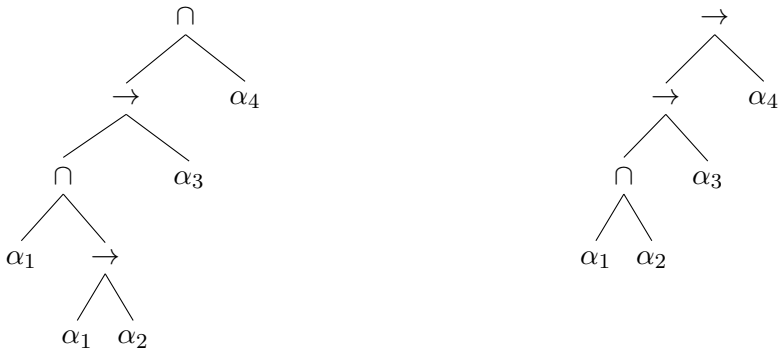
Example 2.2.4 For the λ-term $\lambda x.xx$ the following derivation is obtained:

$$\frac{\dfrac{\{x : \sigma_1 \cap (\sigma_1 \to \sigma_2)\} \vdash_{\mathcal{CD}} x : \sigma_1 \to \sigma_2 \qquad \{x : \sigma_1 \cap (\sigma_1 \to \sigma_2)\} \vdash_{\mathcal{CD}} x : \sigma_1}{\{x : \sigma_1 \cap (\sigma_1 \to \sigma_2)\} \vdash_{\mathcal{CD}} xx : \sigma_2}}{\vdash_{\mathcal{CD}} \lambda x.xx : \sigma_1 \cap (\sigma_1 \to \sigma_2) \to \sigma_2}$$

This system is a true extension of the Curry Type System, allowing term variables to have more than one type in the ($\to$ Intro) derivation rule and the right hand term to also have more than one type in the ($\to$ Elim) derivation rule.
The Coppo-Dezani Type System and other intersection type systems characterize termination – not only typed programs terminate, but all terminating programs are typable as well – which means

that typability is undecidable in these type systems. However, restrictions to these intersection type systems based on the *rank of types* suggested by Daniel Leivant in [Leivant, 1983] can make typability decidable. The *rank* of an intersection type can be easily determined by examining it in tree form: a type is of rank k if no path from the root of the type to an intersection type constructor $\cap$ passes to the left of k arrows (see examples below).

Example 2.2.5 The intersection type $(\alpha_1 \cap (\alpha_1 \to \alpha_2) \to \alpha_3) \cap \alpha_4$ (tree on the left) is a rank 2 type and $(\alpha_1 \cap \alpha_2 \to \alpha_3) \to \alpha_4$ (tree on the right) is a rank 3 type:

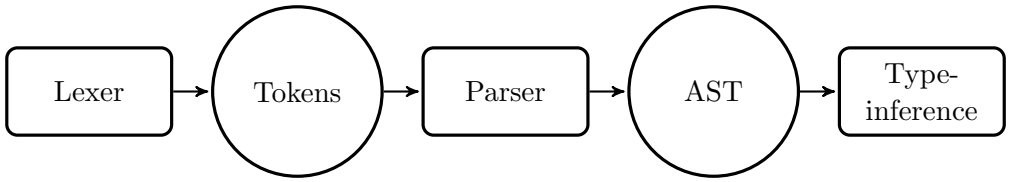


In the Rank 2 Intersection Type System introduced by Leivant, not only is typability decidable, but the system also has principal typings and, as in the Curry Type System for simple types, there is an algorithm that given a term, returns its principal pair. This is the type system on which we will focus our work and it will be described later on.

3 Implementation and Contribution

3.1 General Overview

Our final product, in addition to the type-inference algorithms for simple types, rank 2 intersection types and rank 2 intersection types with quantitative information, which are a tool of semantic analysis, it is also composed of a lexer and a parser that were made with Happy (a parser generator for Haskell). As shown in the scheme below, it first performs a lexical and a syntactical analysis on the input, which generate an Abstract Syntax Tree that is the input of the type-inference algorithms that perform the semantic analysis.



3.2 Rank 2 Intersection Types

As previously mentioned, the type-inference algorithm we implemented and modified is for the Rank 2 Intersection Type System. The type system was first introduced by Daniel Leivant in [Leivant, 1983],

where he briefly covered the type-inference algorithm. Different versions of the algorithm were later defined by Steffen van Bakel in [Van Bakel, 1993] and by Trevor Jim in [Jim, 1995].
We now present the type system, the subtype satisfaction problem and other definitions needed for the algorithm. The definitions in this section can be found in [Jim, 1995].

Notation From now on, we will use $\rho, \rho_1, \rho_2, \ldots$ to range over the set $\mathbb{T}_0$ of simple types, $\sigma, \sigma_1, \sigma_2, \ldots$ to range over the set $\mathbb{T}_1$ of rank 1 intersection types and $\tau, \tau_1, \tau_2, \ldots$ to range over the set $\mathbb{T}_2$ of rank 2 intersection types.

Definition 3.2.1 (Rank 2 intersection types) The rank 1 intersection types $\sigma, \sigma_1, \sigma_2, \cdots \in \mathbb{T}_1$ and the rank 2 intersection types $\tau, \tau_1, \tau_2, \cdots \in \mathbb{T}_2$ are defined by the following grammar:

$$\sigma ::= \rho \mid \sigma_1 \cap \sigma_2$$
$$\tau ::= \rho \mid \sigma \to \tau$$

Thus, rank 1 intersection types consist of intersections of simple types (sequences) and rank 2 intersection types are types possibly containing intersections, but only to the left of a single arrow. Simple types can be seen as rank 0 intersection types, since they do not contain intersections, and $\mathbb{T}_0 = \mathbb{T}_1 \cap \mathbb{T}_2$.

Notation We consider the intersection type constructor $\cap$ to be associative, commutative and non-idempotent (meaning that $\alpha \cap \alpha$ is not equivalent to α). So rank 1 intersection types can be seen as finite, non-empty multi-sets of simple types.

We are particularly interested on non-idempotent intersection types because they provide more quantitative information than the idempotent ones, as we will later discuss.

Definition 3.2.2

- We call *environment* to a basis where the predicates of declarations are sequences (rank 1 intersection types) and we use Γ to range over environments.
- If Γ_1 and Γ_2 are environments, the environment $\Gamma_1 + \Gamma_2$ is defined as follows:
  for each $x \in dom(\Gamma_1) \cup dom(\Gamma_2)$,

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \notin dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \notin dom(\Gamma_1) \\ \Gamma_1(x) \cap \Gamma_2(x) & \text{otherwise} \end{cases}$$

- The definition of *type-substitution* for $\mathbb{T}_1$ and $\mathbb{T}_2$ is similar to the one we described for $\mathbb{T}_0$, only differing on the types.

Definition 3.2.3 (Rank 2 Intersection Type System) The derivation rules of the Rank 2 Intersection Type System are the same of the Coppo-Dezani Type System, but the types are different since they are restricted to rank 2. In the Rank 2 Intersection Type System, we say that M has type τ given the environment Γ, and write

$$\Gamma \vdash M : \tau,$$

if $\Gamma \vdash M : \tau$ can be obtained from the following *derivation rules*, where $n \geq 1$ and $1 \leq i \leq n$:

$$\Gamma \cup \{x : \rho_1 \cap \cdots \cap \rho_n\} \vdash x : \rho_i \qquad \text{(Axiom)}$$

$$\frac{\Gamma_x \cup \{x : \sigma\} \vdash M : \tau}{\Gamma_x \vdash \lambda x.M : \sigma \to \tau} \qquad (\to \text{ Intro})$$

$$\frac{\Gamma \vdash M_1 : \rho_1 \cap \cdots \cap \rho_n \to \tau \qquad \Gamma \vdash M_2 : \rho_1 \cdots \Gamma \vdash M_2 : \rho_n}{\Gamma \vdash M_1 M_2 : \tau} \qquad (\to \text{ Elim})$$

Just like the Milner's Type-Inference Algorithm for the simply typed λ-calculus, the type-inference algorithm that we use for the rank 2 intersection type system is based on the generation and resolution of constraints. In this case, the constraints are inequations that define a subtype satisfaction problem, which will be first transformed into an equivalent unification problem and then solved by the unification algorithm previously defined.

Definition 3.2.4

- The relation $\leq_1$ is the least partial order on $\mathbb{T}_1$ closed under the following rule:

  If $\{\sigma'_1, \sigma'_2, \ldots, \sigma'_m\} \subseteq \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$, then $\sigma_1 \cap \sigma_2 \cap \cdots \cap \sigma_n \leq_1 \sigma'_1 \cap \sigma'_2 \cap \cdots \cap \sigma'_m$.

- The relation $\leq_2$ is the least partial order on $\mathbb{T}_2$ closed under the following rule:

  If $\sigma_1 \leq_1 \sigma_2$ and $\tau_2 \leq_2 \tau_1$, then $\sigma_2 \to \tau_2 \leq_2 \sigma_1 \to \tau_1$.

- Finally, the relation $\leq_{2,1}$ between $\mathbb{T}_2$ and $\mathbb{T}_1$ is the least relation closed under the following rule:

  If $\tau \leq_2 \rho_1, \tau \leq_2 \rho_2, \ldots, \tau \leq_2 \rho_n$, then $\tau \leq_{2,1} \rho_1 \cap \rho_2 \cap \cdots \cap \rho_n$.

  This relation expresses the notion of subtyping and $\tau \leq_{2,1} \sigma$ means that τ is a subtype of σ.

Definition 3.2.5 (Subtype satisfaction problem) Subtype satisfaction is a generalization of the unification problem. A $\leq_{2,1}$-*satisfaction problem* is a finite set P whose every element is either an equality between simple types or an inequality between a $\mathbb{T}_2$ type and a $\mathbb{T}_1$ type. A *solution* to P is a substitution $\mathbb{S}$, such that $\mathbb{S}(\tau) \leq_{2,1} \mathbb{S}(\sigma)$ for all inequalities $(\tau \leq \sigma) \in P$ and $\mathbb{S}(\rho_1) = \mathbb{S}(\rho_2)$ for all equalities $(\rho_1 = \rho_2) \in P$. $\mathcal{S}(P)$ is the set of solutions of P.

Definition 3.2.6 A substitution $\mathbb{S}$ is a *most general solution* (mgs) of P if

$$\mathbb{S} \in \mathcal{S}(P) \text{ and } \forall \mathbb{S}_1 \in \mathcal{S}(P).\, \exists \mathbb{S}_2.\, \mathbb{S}_1 = \mathbb{S}_2 \circ \mathbb{S}.$$

Definition 3.2.7 (Transformation of $\leq_{2,1}$-satisfaction problems) We define the following relation $\Rightarrow$ on $\leq_{2,1}$-satisfaction problems:

$$\begin{array}{lcl}
\{(\sigma \to \tau) \leq \alpha\} \cup P & \Rightarrow & \{\alpha_1 \leq \sigma, \tau \leq \alpha_2, \alpha = \alpha_1 \to \alpha_2\} \cup P \\
 & & \text{where } \alpha_1, \alpha_2 \text{ are new variables} \\
\{(\sigma \to \tau) \leq (\rho_1 \to \rho_2)\} \cup P & \Rightarrow & \{\rho_1 \leq \sigma, \tau \leq \rho_2\} \cup P \\
\{\tau \leq (\sigma_1 \cap \sigma_2)\} \cup P & \Rightarrow & \{\tau \leq \sigma_1, \tau \leq \sigma_2\} \cup P \\
\{\alpha \leq \rho\} \cup P & \Rightarrow & \{\alpha = \rho\} \cup P
\end{array}$$

Definition 3.2.8 ($\leq_{2,1}$-satisfaction problems transformation algorithm) Let P be a $\leq_{2,1}$-satisfaction problem and *UNIFY* the function in Definition 2.1.26. The function $MGS(P)$ that transforms P into an equivalent unification problem and returns its most general solution is defined as:

function $MGS(P)$
  while $P \Rightarrow T$ **do**
    $P \coloneqq T$;
  return $UNIFY(P)$;

Definition 3.2.9 (Type-inference algorithm) Let Γ be an environment, M a λ-term, τ a rank 2 intersection type and *MGS* the function in Definition 3.2.8. The function $T(M) = (\Gamma, \tau)$ defines a type-inference algorithm for the λ-calculus in the Rank 2 Intersection Type System, in the following way:

1. If $M = x$, then $\Gamma = \{x : \alpha\}$ and $\tau = \alpha$, where α is a new variable;
2. If $M = M_1 M_2$, then:
   (a) If $T(M_1) = (\Gamma_1, \alpha_1)$ and $T(M_2) = (\Gamma_2, \tau_2)$, then $T(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3))$, where $\mathbb{S} = MGS(\{\alpha_1 = \alpha_2 \to \alpha_3, \tau_2 \leq \alpha_2\})$ and α_2, α_3 are new variables;
   (b) If $T(M_1) = (\Gamma_1, (\cap_{i \in I} \sigma_i) \to \tau_1)$ and, for each $i \in I$, $T(M_2) = (\Gamma_i, \tau_i)$, then $T(M) = (\mathbb{S}(\Gamma_1 + \sum_{i \in I} \Gamma_i), \mathbb{S}(\tau_1))$, where $\mathbb{S} = MGS(\{\tau_i \leq \sigma_i \mid i \in I\})$;
3. If $M = \lambda x.M_1$ and $T(M_1) = (\Gamma_1, \tau_1)$ then:
   (a) If $x \notin dom(\Gamma_1)$, then $T(M) = (\Gamma_1, \alpha \to \tau_1)$, where α is a new variable;
   (b) If $\{x : \sigma\} \in \Gamma_1$, then $T(M) = (\Gamma_1 \setminus \{x : \sigma\}, \sigma \to \tau_1)$.

This type-inference algorithm and the function that transforms a $\leq 2,1$-satisfaction problem into a equivalent unification problem were implemented in Haskell and can be found in Appendix C. In the next section we will adapt the algorithm to extract quantitative information while inferring the types.

3.3 Quantitative Types

Quantitative types are types that, in addiction to the qualitative information that they give about a program, also provide some quantitative information that can be used as a time complexity measure, for example.
Intersection types, in particular the non-idempotent ones, are an example of types from which we can obtain quantitative information – they bound at the same time the number of evaluation steps and the size of the result. For that reason, and because there is already a type-inference algorithm for the rank 2 intersection type system, which is more powerful and where strictly more terms can be typed, when comparing to popular systems like ML's type system, rank 2 non-idempotent intersection types deserve our attention.
About idempotency, an intuitive example where we can see the adequacy of the non-idempotent intersection types over the idempotent ones, regarding quantitative information, is the following: while with non-idempotent intersection types, the term $\lambda f x.f(fx)$ is typed by $((\alpha_1 \to \alpha_1) \cap (\alpha_1 \to \alpha_1)) \to \alpha_1 \to \alpha_1$, in the idempotent system that type corresponds to $(\alpha_1 \to \alpha_1) \to \alpha_1 \to \alpha_1$, which is the

same result as we would obtain with simple types. Although both typings are correct, the type obtained with non-idempotent intersection types gives us the additional information that f occurs two times, while in the idempotent one this information is lost. On the other hand, the algorithm we implemented, described in Definition 3.2.9, infers the type $((\alpha_1 \to \alpha_2) \cap (\alpha_2 \to \alpha_3)) \to \alpha_1 \to \alpha_3$ for that same term, which means that our algorithm returns types that can be even more general than the non-idempotent version of the intersection types.

There is previous work that makes use of the non-idempotent intersection types with unlimited rank, to obtain quantitative information through type derivations. In [Accattoli et al., 2018], the authors define typing rules for several type systems, corresponding to different evaluation strategies, for which they are able to measure the number of steps taken by that strategy and the size of the term's normal form. They use a technique related to minimal typings named *tightening technique*, where rank 0 types can be what they call *tight constants*.
With the goal of having a type-inference algorithm for the rank 2 intersection type system that, while inferring the type of a term, could give a measure of the number of β-reductions that would have to be performed to get that term in normal form when using a certain evaluation strategy, we started by modifying the algorithm in Definition 3.2.9 using the ideas in [Accattoli et al., 2018]. But we soon abandoned this approach, when we realized that the Trevor Jim's algorithm we had implemented, already inferred non-idempotent intersection types that allowed us to use the constraints generated during the type-inference to obtain quantitative information.

With this idea in mind and after several testing and refinements, we ended up with a new type-inference algorithm for the λ-calculus with rank 2 non-idempotent intersection types that gives quantitative information about the number of β-reductions needed to obtain the typed term in normal form.
The final algorithm is defined bellow and the corresponding Haskell implementation can be found in Appendix D.

Definition 3.3.1 (New $\leq_{2,1}$-satisfaction problems transformation algorithm) Let P be a $\leq_{2,1}$-satisfaction problem and *UNIFY* the function in Definition 2.1.26. The function $QMGS(P) = (\mathbb{S}, C)$ that transforms P into an equivalent unification problem and returns its most general solution $\mathbb{S}$ and an integer C used for counting purposes, is defined as:

function $QMGS(P)$
  $C \coloneqq 0$;
  $E \coloneqq \emptyset$;
  while $P \neq \emptyset$ **do**
    if $P \Rightarrow T$ **then**
      if $P = \{(\sigma \to \tau) \leq (\rho_1 \to \rho_2)\} \cup P'$ **then**
        $C \coloneqq C + 1$;
      $P \coloneqq T$;
    else if $P = \{\alpha = \rho\} \cup P'$ **then**
      $P \coloneqq [\rho/\alpha](P')$;
      $E \coloneqq \{\alpha = \rho\} \cup E$;
  return $(UNIFY(E), C)$;

Definition 3.3.2 (New type-inference algorithm) Let Γ be an environment, M a λ-term, τ a rank 2 intersection type, C the quantitative measure and $QMGS$ the function in Definition 3.3.1. The function $QT(M) = (\Gamma, \tau, C)$ defines a type-inference algorithm (that gives a quantitative measure) for the λ-calculus in the Rank 2 Intersection Type System with non-idempotent types, in the following way:

1. If $M = x$, then $\Gamma = \{x : \alpha\}$, $\tau = \alpha$ and $C = 0$, where α is a new variable;
2. If $M = M_1 M_2$, then:
   (a) If $QT(M_1) = (\Gamma_1, \alpha_1, C_1)$ and $QT(M_2) = (\Gamma_2, \tau_2, C_2)$, then $QT(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3), C_1 + C_2)$, where $(\mathbb{S}, C_3) = QMGS(\{\alpha_1 = \alpha_2 \to \alpha_3, \tau_2 \leq \alpha_2\})$ and α_2, α_3 are new variables;
   (b) If $QT(M_1) = (\Gamma_1, (\cap_{i \in I} \sigma_i) \to \tau_1, C_1)$ and, for each $i \in I$, $QT(M_2) = (\Gamma_i, \tau_i, C_i)$, then $QT(M) = (\mathbb{S}(\Gamma_1 + \sum_{i \in I} \Gamma_i), \mathbb{S}(\tau_1), C)$, where $(\mathbb{S}, C_3) = QMGS(\{\tau_i \leq \sigma_i \mid i \in I\})$ and $C = C_1 + \sum_{i \in I} C_i + C_3 + 1$;
3. If $M = \lambda x.M_1$ and $QT(M_1) = (\Gamma_1, \tau_1, C_1)$ then:
   (a) If $x \notin dom(\Gamma_1)$, then $QT(M) = (\Gamma_1, \alpha \to \tau_1, C_1)$, where α is a new variable;
   (b) If $\{x : \sigma\} \in \Gamma_1$, then $QT(M) = (\Gamma_1 \setminus \{x : \sigma\}, \sigma \to \tau_1, C_1)$.

We believe that this measure C corresponds to the exact number of β-reductions needed to obtain the typed term in normal form, for a certain evaluation strategy. The idea of using the constraints to count the evaluation steps is very intuitive since these constraints are being generated when we have a term that is an application, which is the only term form that will possibly be β-reduced (depending on whether the left subterm is an abstraction or not). However, we cannot simply count the number of β-redexes in the term because new ones might appear when the term is reduced. Instead, what we want is to count all subterms that will eventually, in the evaluation path, act as abstractions that will be applied.
So the constraints that we are interested in counting are the ones of the form $(\sigma \to \tau) \leq (\rho_1 \to \rho_2)$ because it expresses that $(\sigma \to \tau)$ is a subtype of $(\rho_1 \to \rho_2)$, which means that the type $(\sigma \to \tau)$ is a functional type and the term associated with this type is one that will eventually be substituted and become (or already is) an abstraction that will be applied. Notice that these kind of constraints are only generated initially in the case 2.(b) of the type-inference function in Definition 3.3.2 (and then new ones may appear as the function in Definition 3.3.1 transforms the $\leq_{2,1}$-satisfaction problems) and the types of M_1 and M_2 are not added to the constraints, so we also need to increment the count when we match this case.
A less obvious change made to the $\leq_{2,1}$-satisfaction problems transformation algorithm, besides counting the number of the mentioned constraints, is the application of the type-substitution $\mathbb{S} = [\rho/\alpha]$ to all types in P whenever there is a new equation $(\alpha = \rho) \in P$, which is the same procedure done by the type unification algorithm when the fourth transformation in Definition 2.1.25 is applied. As an example of why this is also necessary in our algorithm, consider the term $(\lambda x_1.x_1x_1x_1)(\lambda x_2.x_2)$, for which the correct number of evaluation steps is 3. If we do not perform this substitutions, since $T(\lambda x_1.x_1x_1x_1) = (\{\}, (\alpha_2 \cap \alpha_1 \cap (\alpha_1 \to \alpha_2 \to \alpha_3)) \to \alpha_3)$ and $T(\lambda x_2.x_2) = (\{\}, \alpha'_i \to \alpha'_i)$, for $i \in \{1, 2, 3\}$, the constraints generated would be $P = \{(\alpha'_2 \to \alpha'_2) \leq \alpha_2, (\alpha'_1 \to \alpha'_1) \leq \alpha_1, (\alpha'_3 \to \alpha'_3) \leq (\alpha_1 \to \alpha_2 \to \alpha_3)\}$ and so the

constraint corresponding to the second occurrence of x_1 ($(\alpha'_1 \to \alpha'_1) \leq \alpha_1$) would not be contributing to the value of C (since it is not of the form $(\sigma \to \tau) \leq (\rho_1 \to \rho_2)$), but it should since after the first evaluation step, this occurrence of x_1 (as well as the others) is substituted by $(\lambda x_2.x_2)$, and will after be applied to the term in the position of the third occurrence of x_1 in the original term. And at some point, during the resolution of constraints, the equalities in the satisfaction problem would reveal that α_1 has to be equal to $\alpha_2 \to \alpha_3$ and so by performing the substitutions given by the equalities, the constraint corresponding to the second occurrence of x_1 would become $((\alpha'_1 \to \alpha'_1) \leq (\alpha_2 \to \alpha_3))$ and would correctly contribute to the increment of C, which would be 2 instead of 3 if the substitutions were not applied.

Some other examples of terms for which our algorithm correctly infers the type and gives the correct number of evaluation steps are the following:

- $QT(\lambda x.xx) = (\{\}, \alpha_2 \cap (\alpha_2 \to \alpha_3) \to \alpha_3, 0)$
- $QT((xy)y) = (\{x : \alpha_2 \to \alpha_5 \to \alpha_6, y : \alpha_5 \cap \alpha_2\}, \alpha_6, 0)$
- $QT((\lambda fx.f(fx))(\lambda x.x)) = (\{\}, \alpha_6 \to \alpha_6, 3)$
- $QT((\lambda x.x)(\lambda x.x)(\lambda x.x)) = (\{\}, \alpha_6 \to \alpha_6, 2)$
- $QT((\lambda xyz.xz(yz))(\lambda xy.x)) = (\{\}, (\alpha_6 \to \alpha_{11}) \to \alpha_6 \cap \alpha_9 \to \alpha_9, 3)$
- $QT((\lambda y.(\lambda x.xxx)y)(\lambda x.x)) = (\{\}, \alpha_{14} \to \alpha_{14}, 4)$

4 Final Remarks

Quantitative type systems, just like any other type system, are a powerful tool of static program analysis, but in addition to the qualitative information they also provide quantitative information about programs that can be used to estimate their time and space complexity in compile time, which allow us to know in advance the computational resources that will be required to run the program. Non-idempotent intersection types are an example of types used in this systems as quantitative information can be extracted from them.
The goals of this project were fulfilled as we, based on existing work on type-inference and intersection types, developed a novel type-inference algorithm that uses non-idempotent rank 2 intersection types to obtain the number of evaluation steps of a λ-expression while inferring its type. We did not, however, prove the correction of the algorithm nor determined for which evaluation strategy it is making the counting, which gives opportunity for future work.

Regarding my personal experience and performance on the making of this project, I must say that all the study that it required was very enriching as most of the concepts, besides the basics, were new to me and the fact that we came up with a new contribution made this project an actual research work. I believe that my personal contribution to the developing of the project was significant and that I solved most of the problems faced autonomously, but also with the help of the supervisors as this was a collaborative work.

References

[Accattoli et al., 2018] Accattoli, B., Graham-Lengrand, S., and Kesner, D. (2018). Tight typings and split bounds. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30.

[Barendregt, 1993] Barendregt, H. P. (1993). Lambda calculi with types. In *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*, page 117–309. Oxford University Press, Inc.

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

[Coppo and Dezani-Ciancaglini, 1980] Coppo, M. and Dezani-Ciancaglini, M. (1980). An extension of the basic functionality theory for the λ-calculus. *Notre Dame journal of formal logic*, 21(4):685–693.

[Curry, 1934] Curry, H. B. (1934). Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584.

[Curry et al., 1958] Curry, H. B., Feys, R., Craig, W., Hindley, J. R., and Seldin, J. P. (1958). *Combinatory logic*, volume 1. North-Holland Amsterdam.

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

[Jim, 1995] Jim, T. (1995). Rank 2 type systems and recursive definitions. *Massachusetts Institute of Technology, Cambridge, MA*.

[Leivant, 1983] Leivant, D. (1983). Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 88–98.

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

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41.

[Van Bakel, 1993] Van Bakel, S. J. (1993). *Intersection type disciplines in lambda calculus and applicative term rewriting systems*. PhD thesis.

A Lambda Calculus in Haskell

```
module LambdaCalculus where

data Term = Var TeVar
          | Abs TeVar Term
          | App Term Term
            deriving Eq

data TeVar = TeVar String
             deriving Eq

instance Show Term where
    show (Var x)      = show x
    show (Abs x m)    = ('(':'\':show x) ++ ('.':show m) ++ [')']
    show (App m1 m2) = ('(':show m1) ++ (' ':show m2) ++ [')']

instance Show TeVar where
    show (TeVar x) = id x
```

B Simple Types in Haskell

```haskell
module SimpleTypes where

import LambdaCalculus

data Type0 = TVar TyVar | TAp Type0 Type0
           deriving Eq

data TyVar = TyVar String
           deriving Eq

type EqSet = [(Type0, Type0)]

type Basis = [(TeVar, Type0)]

type Sub = (TyVar, Type0)

type Subst = [Sub]

data Unifier = Uni Subst | FAIL
             deriving (Eq, Show)

instance Show Type0 where
    show (TVar a)    = show a
    show (TAp t1 t2) = ('(':show t1) ++ ('-':'>':show t2) ++ [')']

instance Show TyVar where
    show (TyVar a) = id a

-------Type Unification-------

-- Given a Sub s=(a,t) and a simple type T0, replaces all free
-- occurrences of the type variable a in the type T0 with type t.
subst :: Sub -> Type0 -> Type0
subst (a1, t) (TVar a2) | a1 == a2  = t
                        | otherwise = TVar a2
subst s (TAp t1 t2)                 = TAp (subst s t1) (subst s t2)

-- Given a Sub s=(a,t) and an set of equations eqset, replaces all free
-- occurrences of the type variable a in the types in eqset with type t.
substE :: Sub -> EqSet -> EqSet
substE _ []             = []
substE s ((t1,t2):ts) = (t1',t2'):ts'
                      where t1' = subst s t1
                            t2' = subst s t2
                            ts' = substE s ts

-- Given a Sub s=(a,t) and an a type substitution subst, replaces all free
-- occurrences of the type variable a in the types in subst with type t.
substS :: Sub -> Subst -> Subst
substS _ []             = []
substS s ((t1,t2):ts) = (t1',t2'):ts'
                      where TVar t1' = subst s (TVar t1)
                            t2'      = subst s t2
                            ts'      = substS s ts

-- Checks if a type variable occurs (free) in a simple type.
isFVType :: TyVar -> Type0 -> Bool
isFVType a1 (TVar a2)  = a1 == a2
isFVType a (TAp t1 t2) = isFVType a t1 || isFVType a t2

-- Checks if a type variable occurs (free) in an equation set.
isFVTypeE :: TyVar -> EqSet -> Bool
isFVTypeE _ []           = False
isFVTypeE a ((t1, t2):ts) = isFVType a t1 || isFVType a t2 || isFVTypeE a ts

-- Checks if a type variable occurs (free) in a type substitution.
isFVTypeS :: TyVar -> Subst -> Bool
isFVTypeS _ []           = False
isFVTypeS a ((t1, t2):ts) = isFVType a (TVar t1) || isFVType a t2 || isFVTypeS a ts

-- Unification algorithm.
unify :: (EqSet, Unifier) -> (EqSet, Unifier)
unify ([], u)                          = ([], u)
unify ((t1, t2):ts, u) | t1 == t2      = unify (ts, u)
unify ((TAp t1 t2, TAp t1' t2'):ts, u) = unify ((t1, t1'):(t2, t2'):ts, u)
unify ((TAp t1 t2, TVar a):ts, u)      = unify ((TVar a, TAp t1 t2):ts, u)
unify ((TVar a, t):ts, Uni s)
     | isFVType a t                    = error ("UNIFICATION FAIL: " ++ show ((TVar a, t):ts, Uni s)) --
         ↪ ([], FAIL)
     | isFVTypeE a ts || isFVTypeS a s = let ts' = substE (a, t) ts
```

```
                                        s'  = substS (a, t) s
                                        in unify (ts', Uni ((a, t):s'))
    | otherwise                     = unify (ts, Uni ((a, t):s))

--------Milner's Type-Inference Algorithm--------

substB :: Sub -> Basis -> Basis
substB _ []         = []
substB s ((x,t):ds) = (x, subst s t):substB s ds

substBasis :: Subst -> Basis -> Basis
substBasis [] b     = b
substBasis (s:ts) b = substBasis ts (substB s b)

isInBasis :: TeVar -> Basis -> Bool
isInBasis _ []            = False
isInBasis x1 ((x2, _):ds) = x1 == x2 || isInBasis x1 ds

findInBasis :: TeVar -> Basis -> Type0
findInBasis x1 ((x2, t):ds) | x1 == x2  = t
                            | otherwise = findInBasis x1 ds

rmFromBasis :: TeVar -> Basis -> Basis
rmFromBasis _ []                        = []
rmFromBasis x1 ((x2, t):ds) | x1 == x2  = rmFromBasis x1 ds
                            | otherwise = (x2, t):rmFromBasis x1 ds

mergeBasis :: Basis -> Basis -> EqSet
mergeBasis [] _                           = []
mergeBasis ((x,t):ds) b2 | isInBasis x b2 = (t, findInBasis x b2):mb
                         | otherwise      = mb
                        where mb = mergeBasis ds b2

substTyVar :: Subst -> TyVar -> Type0
substTyVar [] a                         = TVar a
substTyVar ((a1, t):ts) a2 | a1 == a2   = t
                           | otherwise = substTyVar ts a2

typeInf :: Term -> Int -> (Basis, Type0, Int)
typeInf (Var x) n0     = ([(x, TVar (TyVar ('a':(show n0))))], TVar (TyVar ('a':(show n0))), n0+1)
typeInf (App m1 m2) n0 = (substBasis s b, substTyVar s (TyVar ('a':(show n2))), n2+1)
                       where (b1, t1, n1) = typeInf m1 n0
                             (b2, t2, n2) = typeInf m2 n1
                             b            = b1 ++ b2
                             u            = mergeBasis b1 b2
                             ([], Uni s)  = unify (((t1, TAp t2 (TVar (TyVar ('a':(show n2))))):u), Uni [])
typeInf (Abs x m) n0
      | isInBasis x b = (rmFromBasis x b, TAp (findInBasis x b) t, n1)
      | otherwise     = (b, TAp (TVar (TyVar ('a':(show n1)))) t, n1+1)
     where (b, t, n1) = typeInf m n0
```

C Rank 2 Intersection Types in Haskell

```
module Rank2IntersectionTypes where

import LambdaCalculus
import SimpleTypes

data Type1 = T1_0 Type0 | Inters Type1 Type1
           deriving Eq

data Type2 = T2_0 Type0 | T2Ap Type1 Type2
           deriving Eq

type Ineq = (Type2, Type1)

type SatProblem = [Ineq]

type Env = [(TeVar, Type1)]

instance Show Type1 where
    show (T1_0 t)       = show t
    show (Inters t1 t2) = ('(':show t1) ++ ('/':'\\':show t2) ++ [')']

instance Show Type2 where
    show (T2_0 t)     = show t
    show (T2Ap t1 t2) = ('(':show t1) ++ ('-':'>':show t2) ++ [')']

-------Transformation of <=2,1-satisfaction problems into unification problems (based on Corollary 33
    ↪ (Cap.3.1), Rank 2 type systems and recursive definitions)-------

transformSat :: (SatProblem, EqSet, Int) -> (SatProblem, EqSet, Int)
transformSat ([], eqs, n0)                                          = ([], eqs, n0)
transformSat ((T2Ap t1 t2, T1_0 (TVar a)):ts, eqs, n0)              = transformSat ((T2_0 t0, t1):(t2, T1_0
    ↪ t0'):ts, (TVar a, TAp t0 t0'):eqs, n0+2)
                                                                    where t0  = TVar (TyVar ('a':(show n0)))
                                                                          t0' = TVar (TyVar ('a':(show (n0+1))))
transformSat ((T2_0 (TAp t1 t2), T1_0 (TVar a)):ts, eqs, n0)        = transformSat ((T2_0 t0, T1_0 t1):(T2_0
    ↪ t2, T1_0 t0'):ts, (TVar a, TAp t0 t0'):eqs, n0+2)
                                                                    where t0  = TVar (TyVar ('a':(show n0)))
                                                                          t0' = TVar (TyVar ('a':(show (n0+1))))
transformSat ((T2Ap t1 t2, T1_0 (TAp t0 t0')):ts, eqs, n0)          = transformSat ((T2_0 t0, t1):(t2, T1_0
    ↪ t0'):ts, eqs, n0)
transformSat ((T2_0 (TAp t1 t2), T1_0 (TAp t0 t0')):ts, eqs, n0) = transformSat ((T2_0 t0, T1_0 t1):(T2_0
    ↪ t2, T1_0 t0'):ts, eqs, n0)
transformSat ((t2, Inters t1 t1'):ts, eqs, n0)                      = transformSat ((t2, t1):(t2, t1'):ts, eqs,
    ↪ n0)
transformSat ((T2_0 (TVar a), T1_0 t0):ts, eqs, n0)                 = transformSat (ts, (TVar a, t0):eqs, n0)

-------Type-Inference Algorithm (based on Def.36 (Cap.3.2), Rank 2 type systems and recursive
    ↪ definitions)-------

listToInters :: [Type1] -> Type1
listToInters [t1]    = t1
listToInters (t1:ts) = Inters (listToInters ts) t1

subst1 :: Sub -> Type1 -> Type1
subst1 s (T1_0 t0)       = T1_0 (subst s t0)
subst1 s (Inters t1 t1') = Inters (subst1 s t1) (subst1 s t1')

subst2 :: Sub -> Type2 -> Type2
subst2 s (T2_0 t0)    = T2_0 (subst s t0)
subst2 s (T2Ap t1 t2) = T2Ap (subst1 s t1) (subst2 s t2)

substty2 :: Subst -> Type2 -> Type2
substty2 [] t2     = t2
substty2 (s:ts) t2 = substty2 ts (subst2 s t2)

substEn :: Sub -> Env -> Env
substEn _ []         = []
substEn s ((x,t):es) = (x, subst1 s t):substEn s es

substEnv :: Subst -> Env -> Env
substEnv [] e     = e
substEnv (s:ts) e = substEnv ts (substEn s e)

isInEnv :: TeVar -> Env -> Bool
isInEnv _ []             = False
isInEnv x1 ((x2, _):es) = x1 == x2 || isInEnv x1 es

findAllInEnv :: TeVar -> Env -> [Type1]
findAllInEnv _ []                        = []
findAllInEnv x1 ((x2, t):es) | x1 == x2  = t:findAllInEnv x1 es
```

```
                          | otherwise = findAllInEnv x1 es

findInEnv :: TeVar -> Env -> Type1
findInEnv x1 ((x2, t):es) | x1 == x2  = t
                          | otherwise = findInEnv x1 es

rmFromEnv :: TeVar -> Env -> Env
rmFromEnv _ []                      = []
rmFromEnv x1 ((x2, t):es) | x1 == x2  = rmFromEnv x1 es
                          | otherwise = (x2, t):rmFromEnv x1 es

mergeEnv :: Env -> Env
mergeEnv []           = []
mergeEnv ((x, t1):es) = (x, listToInters (findAllInEnv x ((x, t1):es))):mergeEnv (rmFromEnv x es)

genPPSat :: Type1 -> Term -> Int -> (Env, SatProblem, Int)
genPPSat (T1_0 t0) m n0       = (env, [(t, T1_0 t0)], n1)
                              where (env, t, n1) = r2typeInf m n0
genPPSat (Inters t1 t1') m n0 = (mergeEnv (envm1++envm2), sat1++sat2, n2)
                              where (envm1, sat1, n1) = genPPSat t1 m n0
                                    (envm2, sat2, n2) = genPPSat t1' m n1

r2typeInf :: Term -> Int -> (Env, Type2, Int)
r2typeInf (Var x) n0     = let t0 = TVar (TyVar ('a':(show n0))) in ([(x, T1_0 t0)], T2_0 t0, n0+1)
r2typeInf (App m1 m2) n0 = let (env1, t, n1) = r2typeInf m1 n0 in
                           case t of
                               T2_0 (TVar t2_0)   -> (substEnv s env, substty2 s (T2_0 t0'), n3)
                                                     where (env2, t2, n2) = r2typeInf m2 n1
                                                           env            = mergeEnv (env1++env2)
                                                           t0             = TVar (TyVar ('a':(show n2)))
                                                           t0'            = TVar (TyVar ('a':(show (n2+1))))
                                                           ([], eqs, n3)  = transformSat ([(t2, T1_0 t0)],
                                                               ↪ [((TVar t2_0), TAp t0 t0')], n2+2)
                                                           ([], Uni s)    = unify (eqs, Uni [])
                               T2Ap t_1 t_2       -> (substEnv s (mergeEnv (env1++envs)), substty2 s
                                   ↪ t_2, n3)
                                                     where (envs, sat, n2) = genPPSat t_1 m2 n1
                                                           ([], eqs, n3)   = transformSat (sat, [], n2)
                                                           ([], Uni s)     = unify (eqs, Uni [])
                               T2_0 (TAp t_1 t_2) -> (substEnv s (mergeEnv (env1++envs)), substty2 s
                                   ↪ (T2_0 t_2), n3)
                                                     where (envs, sat, n2) = genPPSat (T1_0 t_1) m2 n1
                                                           ([], eqs, n3)   = transformSat (sat, [], n2)
                                                           ([], Uni s)     = unify (eqs, Uni [])
r2typeInf (Abs x m) n0
         | isInEnv x env = (rmFromEnv x env, T2Ap (findInEnv x env) t2, n1)
         | otherwise     = (env, T2Ap (T1_0 (TVar (TyVar ('a':(show n1))))) t2, n1+1)
        where (env, t2, n1) = r2typeInf m n0
```

D Quantitative Rank 2 Intersection Types in Haskell

```
module QuantRank2IntersectionTypes where

import LambdaCalculus
import SimpleTypes

-- Rank 1 intersection types.
data Type1 = T1_0 Type0 | Inters Type1 Type1
             deriving Eq

-- Rank 2 intersection types.
data Type2 = T2_0 Type0 | T2Ap Type1 Type2
             deriving Eq

-- Constraints (inequalities or equalities) of <=2,1-satisfaction problems.
data Constraint = Ineq (Type2, Type1) | Equal (Type2, Type1)

-- <=2,1-satisfaction problems.
type SatProblem = [Constraint]

-- Environments.
type Env = [(TeVar, Type1)]

instance Show Type1 where
    show (T1_0 t)       = show t
    show (Inters t1 t2) = ('(':show t1) ++ ('/':'\\':show t2) ++ [')']

instance Show Type2 where
    show (T2_0 t)      = show t
    show (T2Ap t1 t2) = ('(':show t1) ++ ('-':'>':show t2) ++ [')']

-- Note: every Int appearing in the last position of a returning tuple or
-- as the last argument of a function, is for generating new type variables (a1, a2, a3, ...).


-------Transformation of <=2,1-satisfaction problems into unification problems-------

-- Applies a substitution to a <=2,1-satisfaction problem.
substSat :: Sub -> SatProblem -> SatProblem
substSat _ []                 = []
substSat s (Ineq (t2,t1):sat)  = Ineq (subst2 s t2, subst1 s t1):substSat s sat
substSat s (Equal (t2,t1):sat) = Equal (subst2 s t2, subst1 s t1):substSat s sat

-- Transformation of <=2,1-satisfaction problems into unification problems, with counting of quantitative
    ↪ information.
-- (The third element of the returning tuple is the counter of the quantitative information.)
transformSat :: (SatProblem, EqSet, Int, Int) -> (SatProblem, EqSet, Int, Int)
transformSat ([], eqs, count, n0)                                                 = ([], eqs, count, n0)
transformSat (Ineq (T2Ap t1 t2, T1_0 (TVar a)):ts, eqs, count, n0)                = transformSat (Ineq (T2_0 t0,
    ↪ t1):Ineq (t2, T1_0 t0'):Equal (T2_0 (TVar a), T1_0 (TAp t0 t0')):ts, eqs, count, n0+2)
                                                                                    where t0  = TVar (TyVar
                                                                                        ↪ ('a':(show n0)))
                                                                                          t0' = TVar (TyVar
                                                                                              ↪ ('a':(show (n0+1))))
transformSat (Ineq (T2_0 (TAp t1 t2), T1_0 (TVar a)):ts, eqs, count, n0)          = transformSat (Ineq (T2_0 t0,
    ↪ T1_0 t1):Ineq (T2_0 t2, T1_0 t0'):Equal (T2_0 (TVar a), T1_0 (TAp t0 t0')):ts, eqs, count, n0+2)
                                                                                    where t0  = TVar (TyVar
                                                                                        ↪ ('a':(show n0)))
                                                                                          t0' = TVar (TyVar
                                                                                              ↪ ('a':(show (n0+1))))
transformSat (Ineq (T2Ap t1 t2, T1_0 (TAp t0 t0')):ts, eqs, count, n0)            = transformSat (Ineq (T2_0 t0,
    ↪ t1):Ineq (t2, T1_0 t0'):ts, eqs, count+1, n0)
transformSat (Ineq (T2_0 (TAp t1 t2), T1_0 (TAp t0 t0')):ts, eqs, count, n0)      = transformSat (Ineq (T2_0 t0,
    ↪ T1_0 t1):Ineq (T2_0 t2, T1_0 t0'):ts, eqs, count+1, n0)
transformSat (Ineq (t2, Inters t1 t1'):ts, eqs, count, n0)                        = transformSat (Ineq (t2,
    ↪ t1):Ineq (t2, t1'):ts, eqs, count, n0)
transformSat (Ineq (T2_0 (TVar a), T1_0 t0):ts, eqs, count, n0)                   = transformSat (Equal (T2_0
    ↪ (TVar a), T1_0 t0):ts, eqs, count, n0)
transformSat (Equal (T2_0 (TVar a), T1_0 t0):ts, eqs, count, n0)                  = transformSat (substSat (a,t0)
    ↪ ts, (TVar a, t0):eqs, count, n0)


-------Type-Inference Algorithm-------

-- Given a list of rank 1 intersection types, returns a single type
-- consisting of the intersection of the types in the given list.
listToInters :: [Type1] -> Type1
listToInters [t1]    = t1
listToInters (t1:ts) = Inters (listToInters ts) t1

-- Given a Sub s=(a,t) and a rank 1 intersection type T1, replaces all free
```

```
-- occurrences of the type variable a in the type T1 with type t.
subst1 :: Sub -> Type1 -> Type1
subst1 s (T1_0 t0)       = T1_0 (subst s t0)
subst1 s (Inters t1 t1') = Inters (subst1 s t1) (subst1 s t1')

-- Given a Sub s=(a,t) and a rank 2 intersection type T2, replaces all free
-- occurrences of the type variable a in the type T2 with type t.
subst2 :: Sub -> Type2 -> Type2
subst2 s (T2_0 t0)    = T2_0 (subst s t0)
subst2 s (T2Ap t1 t2) = T2Ap (subst1 s t1) (subst2 s t2)

-- Applies a substitution to a rank 2 intersection type.
substty2 :: Subst -> Type2 -> Type2
substty2 [] t2     = t2
substty2 (s:ts) t2 = substty2 ts (subst2 s t2)

-- Given a Sub s=(a,t) and an environment env, replaces all free
-- occurrences of the type variable a in the types in env with type t.
substEn :: Sub -> Env -> Env
substEn _ []          = []
substEn s ((x,t):es) = (x, subst1 s t):substEn s es

-- Applies a substitution to an environment.
substEnv :: Subst -> Env -> Env
substEnv [] e     = e
substEnv (s:ts) e = substEnv ts (substEn s e)

-- Checks whether or not a term variable is in an environment.
isInEnv :: TeVar -> Env -> Bool
isInEnv _ []                = False
isInEnv x1 ((x2, _):es) = x1 == x2 || isInEnv x1 es

-- Given a term variable x and an environment env,
-- returns a list with all types of x in env.
findAllInEnv :: TeVar -> Env -> [Type1]
findAllInEnv _ []                        = []
findAllInEnv x1 ((x2, t):es) | x1 == x2  = t:findAllInEnv x1 es
                             | otherwise = findAllInEnv x1 es

-- Given a term variable x and an environment env,
-- returns the type of x in env.
-- (It is guaranteed that the function will only be called when
-- there is one and only one occurrence of x in env.)
findInEnv :: TeVar -> Env -> Type1
findInEnv x1 ((x2, t):es) | x1 == x2  = t
                          | otherwise = findInEnv x1 es

-- Removes all occurrences of a term variable from an environment.
rmFromEnv :: TeVar -> Env -> Env
rmFromEnv _ []                        = []
rmFromEnv x1 ((x2, t):es) | x1 == x2  = rmFromEnv x1 es
                          | otherwise = (x2, t):rmFromEnv x1 es

-- Given an environment, replaces all pairs (x,t1), (x,t2)... with a same
-- term variable x with a single pair (x,t) where t=(t1/\t2/\...), ie,
-- the intersection type of t1, t2, ... .
mergeEnv :: Env -> Env
mergeEnv []          = []
mergeEnv ((x, t1):es) = (x, listToInters (findAllInEnv x ((x, t1):es))):mergeEnv (rmFromEnv x es)

-- Auxiliar of the type inference algorithm, performs as many type inferences for the given term
-- as the number of simple types of the given intersection type, and returns the environment
-- and the generated satisfaction problem described in the algorithm.
-- (The third element of the returning tuple is the counter of the quantitative information.)
genPPSat :: Type1 -> Term -> Int -> (Env, SatProblem, Int, Int)
genPPSat (T1_0 t0) m n0       = (env, [Ineq (t, T1_0 t0)], count, n1)
                              where (env, t, count, n1) = quantR2typeInf m n0
genPPSat (Inters t1 t1') m n0 = (mergeEnv (envm1++envm2), sat1++sat2, count1+count2, n2)
                              where (envm1, sat1, count1, n1) = genPPSat t1 m n0
                                    (envm2, sat2, count2, n2) = genPPSat t1' m n1

-- Type inference algorithm for rank 2 intersection types with counting of quantitative information.
-- (The third element of the returning tuple is the counter of the quantitative information.)
quantR2typeInf :: Term -> Int -> (Env, Type2, Int, Int)
quantR2typeInf (Var x) n0     = let t0 = TVar (TyVar ('a':(show n0))) in ([(x, T1_0 t0)], T2_0 t0, 0, n0+1)
quantR2typeInf (App m1 m2) n0 = let (env1, t, count1, n1) = quantR2typeInf m1 n0 in
                                case t of
                                    T2_0 (TVar t2_0)   -> (substEnv s env, substty2 s (T2_0 t0'),
                                        ↪ count1+count2, n3)
                                                          where (env2, t2, count2, n2) = quantR2typeInf m2 n1
                                                                env                    = mergeEnv
                                                                    ↪ (env1++env2)
                                                                t0                     = TVar (TyVar
                                                                    ↪ ('a':(show n2)))
```

```
                                                    t0'                     = TVar (TyVar
                                                        ↪ ('a':(show (n2+1))))
                                                    ([], eqs, _, n3)        = transformSat ([Ineq
                                                        ↪ (t2, T1_0 t0)], [((TVar t2_0), TAp t0
                                                        ↪ t0')], 0, n2+2)
                                                    ([], Uni s)             = unify (eqs, Uni [])
                                        T2Ap t_1 t_2        -> (substEnv s (mergeEnv (env1++envs)), substty2
                                            ↪ s t_2, count1+count2+count3+1, n3)
                                                where (envs, sat, count2, n2) = genPPSat t_1 m2 n1
                                                    ([], eqs, count3, n3)   = transformSat (sat,
                                                        ↪ [], 0, n2)
                                                    ([], Uni s)             = unify (eqs, Uni [])
                                        T2_0 (TAp t_1 t_2) -> (substEnv s (mergeEnv (env1++envs)), substty2
                                            ↪ s (T2_0 t_2), count1+count2+count3+1, n3)
                                                where (envs, sat, count2, n2) = genPPSat (T1_0
                                                    ↪ t_1) m2 n1
                                                    ([], eqs, count3, n3)   = transformSat (sat,
                                                        ↪ [], 0, n2)
                                                    ([], Uni s)             = unify (eqs, Uni [])
quantR2typeInf (Abs x m) n0
              | isInEnv x env = (rmFromEnv x env, T2Ap (findInEnv x env) t2, count, n1)
              | otherwise     = (env, T2Ap (T1_0 (TVar (TyVar ('a':(show n1))))) t2, count, n1+1)
            where (env, t2, count, n1) = quantR2typeInf m n0
```