$\lambda_{\cap}$

Linear Rank Quantitative Types

Fábio Daniel Martins Reis
Mestrado em Ciência de Computadores
Departamento de Ciência de Computadores
2022

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

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

U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

MSc

Declaração de Honra

Eu, Fábio Daniel Martins Reis, inscrito no Mestrado em Ciência de Computadores da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente dissertação reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta dissertação, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente dissertação quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Fábio Reis

Porto, 29/07/2022

Acknowledgments

Throughout the execution of this exciting and challenging work, during both the greatest and the toughest moments, I could always count on my family, who did everything to provide me with the best possible conditions to achieve success, which I share with them. Special thanks to my parents, Daniel and Jacinta, my brother, Diogo, and my aunts, Arminda and Conceição, for their constant support and care.

I would also like to thank all my amazing friends, the ones I met outside the academic context and the university colleagues who quickly became my great friends. The mutual support and companionship within our group were essential, and I could not have asked for better partners to share this journey with.

Last but not least, I want to thank all my Professors, especially my supervisors, Professor Mário Florido and Professor Sandra Alves, with whom I have worked for the last three years. Their clear passion for teaching and researching was very motivating and made this journey a truly exciting and enjoyable experience. They were incredibly supportive, always available to guide me and help improve my work. I have learned immensely from them, and this work would certainly not have the quality I believe it has without all their help and effort. For all of that, I am beyond grateful.

Abstract

Non-idempotent intersection types provide quantitative information about typed programs, and have been used to obtain time and space complexity measures. Intersection type systems characterize termination, so restrictions need to be made in order to make typability decidable. One such restriction consists in using a notion of finite rank for the idempotent intersection types. In this work, we define a new notion of rank for the non-idempotent intersection types. We then define a novel type system and a type inference algorithm for the λ-calculus, using the new notion of rank 2. In the second part of this work, we extend the type system and the type inference algorithm to use the quantitative properties of the non-idempotent intersection types to infer quantitative information related to resource usage. In the last part of this work, as a complement to the theoretical results, we implement (in Haskell) the newly defined type inference algorithms.

Keywords: lambda-calculus,intersection types,quantitative types,tight typings.

Resumo

Tipos com interseções não-idempotentes podem ser usados para fornecer informação quantitativa sobre os programas tipados, e têm sido usados para obter medidas de complexidade. Sistemas de tipos com interseções caracterizam terminação, por isso é necessário fazer restrições de modo a tornar o problema de *typability* decidível. Uma possível restrição consiste em usar uma noção de rank finito para os tipos com interseções idempotentes. Neste trabalho, definimos uma noção nova de rank para os tipos com interseções não-idempotentes. Definimos então um novo sistema de tipos e um algoritmo de inferência de tipos para o λ-*calculus*, usando a nova definição de rank 2. Na segunda parte deste trabalho, estendemos o sistema de tipos e o algoritmo de inferência para usar as propriedades quantitativas dos tipos com interseções não-idempotentes para inferir informação quantitativa relacionada com o uso de recursos. Na ultima parte deste trabalho, como complemento aos resultados teóricos, implementamos (em Haskell) os algoritmos de inferência de tipos que definimos.

Palavras-chave: lambda-calculus,tipos com interseções,tipos quantitativos,tipagens *tight*.

Contents

Chapter 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, besides helping on the detection of errors, can also provide quantitative information related to computational properties.

1.1 Quantitative Types

Intersection types, defined by the grammar $\sigma ::= \alpha \mid \sigma_1 \cap \cdots \cap \sigma_n \to \sigma$ (where α is a type variable and $n \geq 1$), are used in several type systems for the λ-calculus [7, 8, 19, 28] and allow λ-terms to have more than one type. For instance, in an intersection type system, it is possible to assign the type $((\alpha_1 \multimap \alpha_2) \cap \alpha_1) \to \alpha_2$ to the λ-term $\lambda x.xx$ – essentially, the first occurrence of x has the type $\alpha_1 \multimap \alpha_2$ and the second occurrence has the type α_1. Note that this term is not typable in a system like the Curry Type System [9, 10] that uses simple types.

Non-idempotent intersection types [3, 13, 16, 21], also known as *quantitative types*, are a flavour of intersection types in which the type constructor $\cap$ is non-idempotent, and provide more than just qualitative information about programs. They 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.

Type systems based on non-idempotent intersection types, use non-idempotence to count the number of evaluation steps and the size of the result. For instance, in [1], the authors define several quantitative type systems, corresponding to different evaluation strategies, for which they are able to measure the number of steps taken by that strategy to reduce a term to its normal form, and the size of the term's normal form.

1.2 Linear Rank

Typability is undecidable for intersection type systems, because they characterize termination – a λ-term is strongly-normalizable if and only if it is typable in an intersection type system.

One way to get around this is to restrict intersection types to finite ranks, a notion defined by Daniel Leivant in [23] that makes typability decidable [20]. Type systems that use finite-rank intersection types are still very powerful and useful. For instance, rank 2 intersection type systems [12, 19, 27] are more powerful, in the sense that they can type strictly more terms, than popular systems like the ML type system [11].

In Chapter 3, we present a new definition of rank for the quantitative types, which we call *linear rank* and differs from the classical one in the base case – instead of simple types, linear rank 0 intersection types are the linear types. In a non-idempotent intersection type system, every linear term is typable with a simple type (in fact, in many of those systems, only the linear terms are), which is the motivation to use linear types for the base case. The relation between non-idempotent intersection types and linearity has already been studied by Kfoury [21], de Carvalho [13], Philippa Gardner [16] and Florido and Damas [15].

Our motivation to redefine rank in the first place, has to do with our interest in using non-idempotent intersection types to estimate the number of evaluation steps of a λ-term to normal form while inferring its type, and the realization that there is a way to define rank which is more suitable for the quantitative types.

Further in Chapter 3, we define a new intersection type system for the λ-calculus, restricted to linear rank 2 non-idempotent intersection types, and a new type inference algorithm (based on Trevor Jim's [19]), which we prove to be sound and complete with respect to the type system.

1.3 Counting Reductions

One of the main goals in this work is to have a type system and a type inference algorithm capable of giving quantitative information related to resource usage. So in Chapter 4, we extend the type system and inference algorithm presented in Chapter 3, to use the quantitative properties of the linear rank 2 non-idempotent intersection types to infer not only the type of a λ-term, but also the number of evaluation steps of the term to its normal form.

The new type system is the result of a merge between our Linear Rank 2 Intersection Type System from Chapter 3 and the system for the leftmost-outermost evaluation strategy presented in [1]. We prove that the system gives the correct number of evaluation steps for a kind of derivation.

As for the new type inference algorithm, we show that it is sound and complete with respect to the type system for the inferred types, and conjecture that the inferred measures correspond to the ones given by the type system (i.e., correspond to the number of evaluation steps of the term to its normal form, when using the leftmost-outermost evaluation strategy).

In order to test the new algorithm, we also implement it in Haskell, as well as other type inference algorithms and procedures to evaluate terms to normal form.

1.4 Contributions

The main contributions of this work are the following:

- A new definition of rank for non-idempotent intersection types, which we call *linear rank* (Chapter 3);
- A Linear Rank 2 Intersection Type System for the λ-calculus (Chapter 3);
- A type inference algorithm that is sound and complete with respect to the Linear Rank 2 Intersection Type System (Chapter 3);
- A Linear Rank 2 Quantitative Type System for the λ-calculus that derives a measure related to the number of evaluation steps for the leftmost-outermost strategy (Chapter 4);
- A type inference algorithm that is sound and complete with respect to the Linear Rank 2 Quantitative Type System, for the inferred types, and gives a measure that we conjecture to correspond to the number of evaluation steps of the typed term for the leftmost-outermost strategy (Chapter 4);
- Implementation of the newly defined type inference algorithms (Chapter 5).

Part of the work from Chapter 3 and Chapter 4 was presented before by us at the TYPES 2022 conference [25].

Chapter 2

Background

In this chapter we present the basic concepts and existing work that underlie our thesis, including definitions and notations that will be used in subsequent chapters.

2.1 λ-Calculus

The λ-calculus was introduced by Alonzo Church [4] in the 1930s as part of a system intended as a foundation for mathematics. That system was shown to be logically inconsistent in 1935 by Kleene and Rosser [22]. So in 1936, Church separately published the consistent part of the system [5], which we now call the type-free λ-calculus. This, along with its typed versions, has been playing, since then, an instrumental role in computer science, in the theory of programming languages, as well as in many areas of mathematics, philosophy, linguistics and category theory.

For a more complete view into the λ-calculus, please refer to [14]. Some of the definitions in this section can be found in [2, 18].

Notation 2.1.1. We use x, y to range over a countable infinite set $\mathcal{V}$ of variables and M, N to range over the set Λ of λ-terms. In both cases, we may use or not single quotes and/or number subscripts.

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

$$M ::= x \mid (MM) \mid (\lambda x M)$$

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 2.1.1. Some examples of λ-terms are:

$$x;$$
$$(x_1x_2);$$
$$(\lambda x_1(x_1x_2));$$
$$((\lambda x_1(x_1x_2))x_3);$$
$$((\lambda x_3((\lambda x_1(x_1x_2))x_3))x_4).$$

An application (M_1M_2) can be seen as a function M_1 being applied to an argument M_2, and an abstraction (λxM) is a function definition and can be interpreted as 'the function that assigns to x the value M'.

Notation 2.1.2. We use the following convention that lets us omit parentheses:

– outermost parentheses are not written;

– applications are left-associative: $M_1M_2\dots M_n$ stands for $(\dots((M_1M_2)M_3)\dots M_n)$;

– $\lambda x_1x_2\dots x_n.M$ stands for $(\lambda x_1(\lambda x_2(\dots(\lambda x_n(M))\dots)))$.

Example 2.1.2. Using this convention, the terms in Example 2.1.1 may be written as follows:

$$x;$$
$$x_1x_2;$$
$$\lambda x_1.x_1x_2;$$
$$(\lambda x_1.x_1x_2)x_3;$$
$$(\lambda x_3.(\lambda x_1.x_1x_2)x_3)x_4.$$

Definition 2.1.2 (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 $\mathsf{FV}(M)$ of free variables of M is defined inductively as follows:

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

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

Example 2.1.3. 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 $\mathsf{FV}((\lambda x_1x_2.x_1x_2x_3)x_4) = \{x_3, x_4\}$.

In the λ-term $\lambda x_1x_2.x_1x_2x_2$, x_1 and x_2 occur both as bound variables and $\mathsf{FV}(\lambda x_1x_2.x_1x_2x_2) = \emptyset$, so this term is closed.

In the λ-term $x_1(\lambda x_1 x_2.x_1 x_2 x_3)$, x_1 and x_3 occur as free variables, x_1 and x_2 occur as bound variables and $\mathsf{FV}(x_1(\lambda x_1 x_2.x_1 x_2 x_3)) = \{x_1, x_3\}$. 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 convention below, a case like this one will never happen.

Convention 2.1.1 (Barendregt's Variable Convention)**.** If $M, M_1, M_2, \dots, N, N_1, N_2, \dots$ 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 2.1.3 (Substitution)**.** We call *substitution* to

$$\mathcal{S} = [N/x].$$

$\mathcal{S}(M) = M[N/x]$ is the result of substituting the term N for each free occurrence of x in the term M and can be inductively defined as follows:

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

Example 2.1.4. If we apply the substitution $[x_5/x_3]$ to the first term in Example 2.1.3, we have:

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

Notation 2.1.3. We write $M[M_1/x_1, M_2/x_2, \dots, M_n/x_n]$ for $(\dots((M[M_1/x_1])[M_2/x_2])\dots)[M_n/x_n]$.

Composing two substitutions $\mathcal{S}_1$ and $\mathcal{S}_2$ results in a substitution $\mathcal{S}_2 \circ \mathcal{S}_1$ that when applied, has the same effect as applying $\mathcal{S}_1$ followed by $\mathcal{S}_2$.

Definition 2.1.4 (Composition)**.** The composition of two substitutions $\mathcal{S}_1 = [N_1/x_1]$ and $\mathcal{S}_2 = [N_2/x_2]$, denoted by $\mathcal{S}_2 \circ \mathcal{S}_1$, is defined as:

$$\mathcal{S}_2 \circ \mathcal{S}_1(M) = M[N_1/x_1, N_2/x_2].$$

Also, we assume that the operation is right-associative:

$$\mathcal{S}_1 \circ \mathcal{S}_2 \circ \cdots \circ \mathcal{S}_{n-1} \circ \mathcal{S}_n = \mathcal{S}_1 \circ (\mathcal{S}_2 \circ \cdots \circ (\mathcal{S}_{n-1} \circ \mathcal{S}_n) \ldots).$$

Definition 2.1.5 (β-reduction)**.** β-reduction captures the notion of function application and the rule states that a term of the form $(\lambda x.M)N$ (called a *β-redex*) β-reduces to $M[N/x]$ (its *contractum*), notation:

$$(\lambda x.M)N \longrightarrow_\beta M[N/x].$$

Definition 2.1.6 (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 2.1.5. 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

$$(\lambda x_2.x_2x_3)x_4 \longrightarrow_\beta (x_2x_3)[x_4/x_2] = x_4x_3,$$

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.

2.2 Simple Types

The simply typed λ-calculus is a typed interpretation of the λ-calculus, introduced by Alonzo Church in [6] and by Haskell Curry and Robert Feys in [10].

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 [9] for the theory of combinators, and then modified for the λ-calculus in [10].

Notation 2.2.1. We use α to range over a countable infinite set $\mathbb{V}$ of type variables and τ to range over the set $\mathbb{T}_0$ of simple types. In both cases, we may use or not single quotes and/or number subscripts.

Definition 2.2.1 (Simple types)**.** Simple types $\tau, \tau_1, \tau_2, \ldots \in \mathbb{T}_0$ are defined by the following grammar:

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

where a type of the form:

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

Notation 2.2.2. 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 2.2.1. Some examples of simple types are:

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

Definition 2.2.2.

- A *statement* is an expression 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.
- An *environment* Γ is a set of declarations where all subjects are distinct.

Definition 2.2.3. If $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is an environment, then

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

Definition 2.2.4 (Curry Type System)**.** In the Curry Type System, we say that M has type τ given the environment Γ, and write

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

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

$$\Gamma \cup \{x : \tau\} \vdash_{\mathcal{C}} x : \tau \tag{Axiom}$$

$$\frac{\Gamma \cup \{x : \tau_1\} \vdash_{\mathcal{C}} M : \tau_2}{\Gamma \vdash_{\mathcal{C}} \lambda x.M : \tau_1 \to \tau_2} \tag{$\to$ Intro}$$

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

Example 2.2.2. For the λ-term $\lambda x_1 x_2.x_1$ the following derivation is obtained:

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

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

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

Definition 2.2.5 (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 $\mathbb{T}_0$, $\mathbb{S}(\tau) = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ is the type obtained by simultaneously substituting α_i by τ_i in τ, with $1 \leq i \leq n$.

The type $\mathbb{S}(\tau)$ is called an *instance* of the type τ.

The notion of type-substitution can be extended to environments 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 environment $\mathbb{S}(\Gamma)$ is called an *instance* of the environment Γ.

Example 2.2.3. 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 2.2.6 (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 an environment 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 called a *type inference algorithm* and for the Curry Type System there is the Milner's Type Inference Algorithm, presented in [24].

2.3 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 [7], which was one of the first to use intersection types, and a basis for subsequent systems.

Definition 2.3.1 (Intersection types)**.** Intersection types $\sigma, \sigma_1, \sigma_2, \ldots \in \mathbb{T}$ 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* of types.

Note that intersections arise in different systems in different scopes. Here we follow several previous presentations where intersections are only allowed directly on the left-hand side of arrow types and sequences are non-empty [7, 8, 19, 28].

Notation 2.3.1. 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 2.3.1. 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 2.3.2 (Coppo-Dezani Type System)**.** In the Coppo-Dezani Type System, we say that M has type σ given the environment 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 $1 \leq i \leq n$:

$$\Gamma \cup \{x : \sigma_1 \cap \cdots \cap \sigma_n\} \vdash_{\mathcal{CD}} x : \sigma_i \tag{Axiom}$$

$$\frac{\Gamma \cup \{x : \sigma_1 \cap \cdots \cap \sigma_n\} \vdash_{\mathcal{CD}} M : \sigma}{\Gamma \vdash_{\mathcal{CD}} \lambda x.M : \sigma_1 \cap \cdots \cap \sigma_n \to \sigma} \tag{$\to$ Intro}$$

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

Example 2.3.2. For the λ-term $\lambda x.xx$ the following derivation is obtained:

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

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.

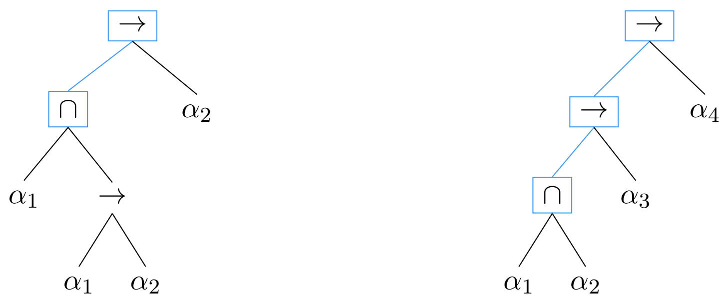
2.3.1 Finite Rank

Intersection type systems, like the Coppo-Dezani Type System, characterize termination, in the sense that a λ-term is strongly-normalizable if and only if it is typable in an intersection type system. Thus, typability is undecidable for these systems.

To get around this, some current intersection type systems are restricted to types of finite rank [12, 19, 20, 27] using a notion of rank first defined by Daniel Leivant in [23]. This restriction makes typability decidable [20]. Despite using finite-rank intersection types, these systems are still very powerful and useful. For instance, rank 2 intersection type systems [12, 19, 27] are more powerful, in the sense that they can type strictly more terms, than popular systems like the ML type system [11].

The *rank* of an intersection type is related to the depth of the nested intersections and it can be easily determined by examining the type 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.

Example 2.3.3. The intersection type $\alpha_1 \cap (\alpha_1 \to \alpha_2) \to \alpha_2$ (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:



Definition 2.3.3 (Rank of intersection types)**.** Let $\mathbb{T}_0$ be the set of simple types and $\mathbb{T}_1 = \{\tau_1 \cap \cdots \cap \tau_m \mid \tau_1, \ldots, \tau_m \in \mathbb{T}_0, m \geq 1\}$ the set of sequences of simple types (written as $\vec{\tau}, \vec{\tau}_1, \vec{\tau}_2, \ldots$). The set $\mathbb{T}_k$, of rank k intersection types (for $k \geq 2$), can be defined recursively in the following way ($n \geq 3$, $m \geq 1$):

$$\mathbb{T}_2 = \mathbb{T}_0 \cup \{\vec{\tau} \to \sigma \mid \vec{\tau} \in \mathbb{T}_1, \sigma \in \mathbb{T}_2\}$$
$$\mathbb{T}_n = \mathbb{T}_{n-1} \cup \{\vec{\tau}_1 \cap \cdots \cap \vec{\tau}_m \to \sigma \mid \vec{\tau}_1, \ldots, \vec{\tau}_m \in \mathbb{T}_{n-1}, \sigma \in \mathbb{T}_n\}$$

Notation 2.3.2. We consider the intersection type constructor $\cap$ to be associative, commutative and non-idempotent (meaning that $\alpha \cap \alpha$ is not equivalent to α).

We are particularly interested in non-idempotent intersection types, also known as quantitative types, because they provide more quantitative information than the idempotent ones.

2.4 Quantitative Types

Quantitative types [3, 13, 16, 21] 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. These systems are based on non-idempotent intersection types, where non-idempotence has been used to count the number of evaluation steps and the size of the result.

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 $\lambda fx.f(fx)$ is typed by $((\alpha \to \alpha) \cap (\alpha \to \alpha)) \to \alpha \to \alpha$, in an idempotent system that type corresponds to $(\alpha \to \alpha) \to \alpha \to \alpha$, 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 twice, while in the idempotent one, that information is lost.

There is previous work that makes use of the non-idempotent intersection types with unlimited rank, to obtain quantitative information through type derivations. Namely, in [1], 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 notion related to minimal typings named *tightness*, where rank 0 types include *tight constants*.

We now present the type system for the leftmost-outermost evaluation strategy in [1], as we will define a new type system in Chapter 4, based on that system, with the ultimate goal of creating a new type inference algorithm capable of inferring the number of evaluations steps of a term to its normal form.

The type system makes use of the predicates normal, neutral and abs. The predicates normal and neutral defining, respectively, the leftmost-outermost normal terms and neutral terms, are in Definition 2.4.1. The predicate $\mathsf{abs}(M)$ is true if and only if M is an abstraction; $\mathsf{normal}(\mathsf{M})$ means that M is in normal form; and $\mathsf{neutral}(M)$ means that M is in normal form and can never behave as an abstraction, i.e., it does not create a redex when applied to an argument.

Definition 2.4.1 (Leftmost-outermost normal forms).

$$\frac{}{\mathsf{neutral}(x)} \qquad \frac{\mathsf{neutral}(M) \qquad \mathsf{normal}(N)}{\mathsf{neutral}(MN)} \qquad \frac{\mathsf{neutral}(M)}{\mathsf{normal}(M)} \qquad \frac{\mathsf{normal}(M)}{\mathsf{normal}(\lambda x.M)}$$

Definition 2.4.2 (Leftmost-outermost evaluation strategy).

$$\frac{}{(\lambda x.M)N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'} \qquad \frac{M \longrightarrow M' \qquad \neg\mathsf{abs}(M)}{MN \longrightarrow M'N}$$

$$\frac{\mathsf{neutral}(N) \qquad M \longrightarrow M'}{NM \longrightarrow NM'}$$

Definition 2.4.3 (Leftmost-outermost size of terms)**.** The leftmost-outermost size $|M|$ of a term M is defined as follows:

$$\begin{aligned}
|x| &= 0 \\
|\lambda x.M| &= |M| + 1 \\
|M_1 M_2| &= |M_1| + |M_2| + 1
\end{aligned}$$

Definition 2.4.4 (Multi-types)**.** The types $\sigma, \sigma_1, \sigma_2, \ldots$ of the system (called *multi-types*) are defined by the following grammar:

$$\begin{aligned}
\mathsf{tight} &::= \mathsf{Neutral} \mid \mathsf{Abs} && \text{(Tight constants)} \\
\sigma &::= \mathsf{tight} \mid \alpha \mid \mu \to \sigma && \text{(Multi-types)} \\
\mu &::= [\sigma_1, \ldots, \sigma_n] \quad (n \geq 0) && \text{(Multisets)}
\end{aligned}$$

Note that this definition is similar to the classical definition of intersection types (Definition 2.3.1). The only differences are that here, a sequence is represented by a (possibly empty) multiset, and a type can also be a tight constant ($\mathsf{Neutral}$ or Abs).

Definition 2.4.5.

- Here, an environment Γ is a map from variables to finite multisets μ of types such that only finitely many variables are not mapped to the empty multiset $[\,]$;
- $\mathsf{dom}(\Gamma) = \{x \mid \Gamma(x) \neq [\,]\}$;
- Γ_x is defined by $\Gamma_x(x) = [\,]$ and $\Gamma_x(y) = \Gamma(y)$ if $y \neq x$;
- The environment $\Gamma_1 + \Gamma_2$ is defined as $(\Gamma_1 + \Gamma_2)(x) = \Gamma_1(x) \uplus \Gamma_2(x)$, where $\uplus$ is the multiset sum.
- We use the notation Tight for multisets with only types of the form tight. Moreover, we write $\mathsf{tight}(\sigma)$ if σ is of the form tight, $\mathsf{tight}(\mu)$ if μ is of the form Tight, and $\mathsf{tight}(\Gamma)$ if $\mathsf{tight}(\Gamma(x))$ for all x, in which case we also say that Γ is tight.

Definition 2.4.6. In the type system for the leftmost-outermost evaluation presented in [1], we say that M has type σ given the environment Γ, with indices (b, r), and write

$$\Gamma \vdash^{(b,r)} M : \sigma$$

if it can be obtained from the following *derivation rules*:

$$\{x : [\sigma]\} \vdash^{(0,0)} x : \sigma \tag{ax}$$

$$\frac{\Gamma \vdash^{(b,r)} M : \sigma}{\Gamma_x \vdash^{(b+1,r)} \lambda x.M : \Gamma(x) \to \sigma} \tag{fun$_b$}$$

$$\frac{\Gamma \vdash^{(b,r)} M : \mathsf{tight} \qquad \mathsf{tight}(\Gamma(x))}{\Gamma_x \vdash^{(b,r+1)} \lambda x.M : \mathsf{Abs}} \qquad (\mathrm{fun}_r)$$

$$\frac{\Gamma_1 \vdash^{(b_1,r_1)} M_1 : \mu \to \sigma \qquad \Gamma_2 \vdash^{(b_2,r_2)} M_2 : \mu}{\Gamma_1 + \Gamma_2 \vdash^{(b_1+b_2+1,r_1+r_2)} M_1 M_2 : \sigma} \qquad (\mathrm{app}_b)$$

$$\frac{\Gamma_1 \vdash^{(b_1,r_1)} M_1 : \mathsf{Neutral} \qquad \Gamma_2 \vdash^{(b_2,r_2)} M_2 : \mathsf{tight}}{\Gamma_1 + \Gamma_2 \vdash^{(b_1+b_2,r_1+r_2+1)} M_1 M_2 : \mathsf{Neutral}} \qquad (\mathrm{app}_r)$$

$$\frac{\Gamma_1 \vdash^{(b_1,r_1)} M : \sigma_1 \quad \cdots \quad \Gamma_n \vdash^{(b_n,r_n)} M : \sigma_n}{\sum_{i=1}^{n} \Gamma_i \vdash^{(b_1+\cdots+b_n,r_1+\cdots+r_n)} M : [\sigma_1, \ldots, \sigma_n]} \qquad (\mathrm{many})$$

Definition 2.4.7 (Tight derivations)**.** A derivation ending with $\Gamma \vdash^{(b,r)} M : \sigma$ is tight if $\mathsf{tight}(\sigma)$ and $\mathsf{tight}(\Gamma)$.

In [1], it has been proved that whenever a term is tightly typable with indices (b, r), then b is exactly the double of the number of evaluations steps to leftmost-outermost normal form and r is exactly the size of the leftmost-outermost normal form. Moreover, every leftmost-outermost normalising term has a tight derivation in the system. These two properties are formalized in Theorem 2.4.1 and Theorem 2.4.2.

The following example of derivation is adapted from [1].

Example 2.4.1. Let $M = (\lambda x_1.(\lambda x_2.x_2 x_1)x_1)I$, where I is the identity function $\lambda y.y$.

Let us first consider the leftmost-outermost evaluation of M to normal form:

$$(\lambda x_1.(\lambda x_2.x_2 x_1)x_1)I \longrightarrow (\lambda x_2.x_2 I)I \longrightarrow II \longrightarrow I$$

So the evaluation sequence has length 3 and the leftmost-outermost normal form has size 1.

Let us write $\overrightarrow{\mathsf{Abs}}$ for the type $[\mathsf{Abs}] \to \mathsf{Abs}$. Then for the λ-term M, the following tight derivation is obtained:

$$
\begin{prooftree}
\AxiomC{$\{x_2 : [\overrightarrow{\mathsf{Abs}}]\} \vdash^{(0,0)} x_2 : \overrightarrow{\mathsf{Abs}}$}
\AxiomC{$\{x_1 : [\mathsf{Abs}]\} \vdash^{(0,0)} x_1 : \mathsf{Abs}$}
\UnaryInfC{$\{x_1 : [\mathsf{Abs}]\} \vdash^{(0,0)} x_1 : [\mathsf{Abs}]$}
\BinaryInfC{$\{x_2 : [\overrightarrow{\mathsf{Abs}}], x_1 : [\mathsf{Abs}]\} \vdash^{(1,0)} x_2 x_1 : \mathsf{Abs}$}
\UnaryInfC{$\{x_1 : [\mathsf{Abs}]\} \vdash^{(2,0)} \lambda x_2.x_2 x_1 : [\overrightarrow{\mathsf{Abs}}] \to \mathsf{Abs}$}
\AxiomC{$\{x_1 : [\overrightarrow{\mathsf{Abs}}]\} \vdash^{(0,0)} x_1 : \overrightarrow{\mathsf{Abs}}$}
\UnaryInfC{$\{x_1 : [\overrightarrow{\mathsf{Abs}}]\} \vdash^{(0,0)} x_1 : [\overrightarrow{\mathsf{Abs}}]$}
\BinaryInfC{$\{x_1 : [\mathsf{Abs}, \overrightarrow{\mathsf{Abs}}]\} \vdash^{(3,0)} (\lambda x_2.x_2 x_1)x_1 : \mathsf{Abs}$}
\UnaryInfC{$\{\,\} \vdash^{(4,0)} \lambda x_1.(\lambda x_2.x_2 x_1)x_1 : [\mathsf{Abs}, \overrightarrow{\mathsf{Abs}}] \to \mathsf{Abs}$}
\AxiomC{$\vdots$}
\UnaryInfC{$\{\,\} \vdash^{(1,1)} I : [\mathsf{Abs}, \overrightarrow{\mathsf{Abs}}]$}
\BinaryInfC{$\{\,\} \vdash^{(6,1)} (\lambda x_1.(\lambda x_2.x_2 x_1)x_1)I : \mathsf{Abs}$}
\end{prooftree}
$$

So indeed, the indices $(6, 1)$ represent $6/2 = 3$ evaluation steps to leftmost-outermost normal form and a leftmost-outermost normal form of size 1.

Theorem 2.4.1 (Tight correctness)**.** If there is a tight derivation ending with $\Gamma \vdash^{(b,r)} M : \sigma$, then there exists N such that $M \longrightarrow^{b/2} N$, $\mathsf{normal}(N)$ and $|N| = r$. Moreover, if $\sigma = \mathsf{Neutral}$, then $\mathsf{neutral}(N)$.

Theorem 2.4.2 (Tight completeness)**.** Let $M \longrightarrow^k N$, with $\mathsf{normal}(N)$.

Then there exists a tight derivation ending with $\Gamma \vdash^{(2k,|N|)} M : \sigma$. Moreover, if $\mathsf{neutral}(N)$ then $\sigma = \mathsf{Neutral}$, and if $\mathsf{abs}(N)$ then $\sigma = \mathsf{Abs}$.

For the proofs of these and other properties of this system (and other type systems for different evaluation strategies), please refer to [1].

Chapter 3

Linear Rank Intersection Types

In the previous chapter, we mentioned several intersection type systems in which intersection is idempotent and types are rank-restricted. We followed by presenting quantitative type systems that, on the other hand, make use of non-idempotent intersection types, for which there is no specific definition of rank.

The generalization of ranking for non-idempotent intersection types is not trivial and raises interesting questions that we will address in this chapter, along with a definition of a new non-idempotent intersection type system and a type inference algorithm.

This and the following chapters cover original work that we presented at the TYPES 2022 conference [25].

3.1 Linear Rank

We noticed that the set of terms typed using idempotent rank 2 intersection types and non-idempotent rank 2 intersection types is not the same. For instance, the term $(\lambda x.xx)(\lambda fx.f(fx))$ is typable with a simple type when using idempotent intersection types, but not when using non-idempotent intersection types. This comes from the two different occurrences of f in $\lambda fx.f(fx)$, which even if typed with the same type, are not contractible because intersection is non-idempotent. Note that this is strongly related to the linearity features of terms. A λ-term M is called a *linear term* if and only if, for each subterm of the form $\lambda x.N$ in M, x occurs free in N exactly once, and if each free variable of M has just one occurrence free in M. So the term $(\lambda x.xx)(\lambda fx.f(fx))$ is not typable with a non-idempotent rank 2 intersection type precisely because the term $\lambda fx.f(fx)$ is not linear.

Note that in a non-idempotent intersection type system, every linear term is typable with a simple type (in fact, in many of those systems, only the linear terms are). This motivated us to come up with a new notion of rank for non-idempotent intersection types, based on linear types (the ones derived in a linear type system – a substructural type system in which each assumption must be used exactly once, corresponding to the implicational fragment of linear logic [17]).

The relation between non-idempotent intersection types and linearity was first introduced by Kfoury [21] and further explored by de Carvalho [13], who established its relation with linear logic.

Here we propose a new definition of rank for intersection types, which we call *linear rank* and differs from the classical one in the base case – instead of simple types, linear rank 0 intersection types are the linear types – and in the introduction of the functional type constructor 'linear arrow' $\multimap$.

Definition 3.1.1 (Linear rank of intersection types)**.** Let $\mathbb{T}_{\mathbb{L}0} = \mathbb{V} \cup \{\tau_1 \multimap \tau_2 \mid \tau_1, \tau_2 \in \mathbb{T}_{\mathbb{L}0}\}$ be the set of linear types and $\mathbb{T}_{\mathbb{L}1} = \{\tau_1 \cap \cdots \cap \tau_m \mid \tau_1, \ldots, \tau_m \in \mathbb{T}_{\mathbb{L}0}, m \geq 1\}$ the set of sequences of linear types. The set $\mathbb{T}_{\mathbb{L}k}$, of *linear rank* k intersection types (for $k \geq 2$), can be defined recursively in the following way ($n \geq 3$, $m \geq 2$):

$$\begin{aligned}
\mathbb{T}_{\mathbb{L}2} &= \mathbb{T}_{\mathbb{L}0} \cup \{\tau \multimap \sigma \mid \tau \in \mathbb{T}_{\mathbb{L}0}, \sigma \in \mathbb{T}_{\mathbb{L}2}\} \\
&\quad \cup \{\tau_1 \cap \cdots \cap \tau_m \to \sigma \mid \tau_1, \ldots, \tau_m \in \mathbb{T}_{\mathbb{L}0}, \sigma \in \mathbb{T}_{\mathbb{L}2}\} \\
\mathbb{T}_{\mathbb{L}n} &= \mathbb{T}_{\mathbb{L}n-1} \cup \{\vec{\tau} \multimap \sigma \mid \vec{\tau} \in \mathbb{T}_{\mathbb{L}n-1}, \sigma \in \mathbb{T}_{\mathbb{L}n}\} \\
&\quad \cup \{\vec{\tau}_1 \cap \cdots \cap \vec{\tau}_m \to \sigma \mid \vec{\tau}_1, \ldots, \vec{\tau}_m \in \mathbb{T}_{\mathbb{L}n-1}, \sigma \in \mathbb{T}_{\mathbb{L}n}\}
\end{aligned}$$

Initially, the idea for the change arose from our interest in using rank-restricted intersection types to estimate the number of evaluation steps of a λ-term while inferring its type. While defining the intersection type system to obtain quantitative information, we realized that the ranks could be potentially more useful for that purpose if the base case was changed to types that give more quantitative information in comparison to simple types, which is the case for linear types – for instance, if a term is typed with a linear rank 2 intersection type, one knows that its arguments are linear, meaning that they will be used exactly once.

It is not clear, and most likely non-trivial, the relation between the standard definition of rank and our definition of linear rank. Note that the set of terms typed using standard rank 2 intersection types [19, 27] and linear rank 2 intersection types is not the same. For instance, again, the term $(\lambda x.xx)(\lambda fx.f(fx))$, typable with a simple type in the standard Rank 2 Intersection Type System, is not typable in the Linear Rank 2 Intersection Type System, because, as the term $(\lambda fx.f(fx))$ is not linear and intersection is not idempotent, by Definition 3.1.1, the type of $(\lambda x.xx)(\lambda fx.f(fx))$ is now (linear) rank 3. This relation between rank and linear rank is an interesting question that will not be covered here, but one that we would like to explore in the future.

3.2 Type System

We now define a new type system for the λ-calculus with linear rank 2 non-idempotent intersection types.

Note that some of the definitions presented in this section and the next, were already introduced in Chapter 2, but will now be recalled and adapted.

Notation 3.2.1. From now on, we will use α to range over a countable infinite set $\mathbb{V}$ of type variables, τ to range over the set $\mathbb{T}_{\mathbb{L}0}$ of linear types, $\vec{\tau}$ to range over the set $\mathbb{T}_{\mathbb{L}1}$ of linear type sequences and σ to range over the set $\mathbb{T}_{\mathbb{L}2}$ of linear rank 2 intersection types. In all cases, we may use or not single quotes and/or number subscripts.

Convention 3.2.1. We consider types equal up to renaming of variables.

Definition 3.2.1.

- A *statement* is an expression of the form $M : \vec{\tau}$, where $\vec{\tau}$ 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.
- The comma operator (,) appends a declaration to the end of a list (of declarations). The list (Γ_1, Γ_2) is the list that results from appending the list Γ_2 to the end of the list Γ_1.
- A finite list of declarations is *consistent* if and only if the term variables are all distinct.
- We call *environment* to a consistent finite list of declarations which predicates are sequences of linear types (i.e., elements of $\mathbb{T}_{\mathbb{L}1}$) and we use Γ (possibly with single quotes and/or number subscripts) to range over environments.
- If $\Gamma = [x_1 : \vec{\tau}_1, \dots, x_n : \vec{\tau}_n]$ is an environment, then Γ is a partial function, with domain $\mathsf{dom}(\Gamma) = \{x_1, \dots, x_n\}$, and $\Gamma(x_i) = \vec{\tau}_i$.
- We write Γ_x for the resulting environment of eliminating the declaration of x from Γ (if there is no declaration of x in Γ, then $\Gamma_x = \Gamma$).
- We write $\Gamma_1 \equiv \Gamma_2$ if the environments Γ_1 and Γ_2 are equal up to the order of the declarations.
- If Γ_1 and Γ_2 are environments, the environment $\Gamma_1 + \Gamma_2$ is defined as follows:

  for each $x \in \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$,

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

  with the declarations of the variables in $\mathsf{dom}(\Gamma_1)$ in the beginning of the list, by the same order they appear in Γ_1, followed by the declarations of the variables in $\mathsf{dom}(\Gamma_2) \setminus \mathsf{dom}(\Gamma_1)$, by the order they appear in Γ_2.

Definition 3.2.2 (Linear Rank 2 Intersection Type System)**.** In the Linear Rank 2 Intersection Type System, we say that M has type σ given the environment Γ, and write

$$\Gamma \vdash_2 M : \sigma$$

if it can be obtained from the following *derivation rules*:

$$[x : \tau] \vdash_2 x : \tau \tag{Axiom}$$

$$\frac{\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2 \vdash_2 M : \sigma}{\Gamma_1, y : \vec{\tau}_2, x : \vec{\tau}_1, \Gamma_2 \vdash_2 M : \sigma} \tag{Exchange}$$

$$\frac{\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M : \sigma}{\Gamma_1, x : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2 \vdash_2 M[x/x_1, x/x_2] : \sigma} \tag{Contraction}$$

$$\frac{\Gamma, x : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M : \sigma \qquad n \geq 2}{\Gamma \vdash_2 \lambda x.M : \tau_1 \cap \cdots \cap \tau_n \to \sigma} \tag{$\to$ Intro}$$

$$\frac{\Gamma \vdash_2 M_1 : \tau_1 \cap \cdots \cap \tau_n \to \sigma \qquad \Gamma_1 \vdash_2 M_2 : \tau_1 \ \cdots \ \Gamma_n \vdash_2 M_2 : \tau_n \qquad n \geq 2}{\Gamma, \sum_{i=1}^{n} \Gamma_i \vdash_2 M_1 M_2 : \sigma} \tag{$\to$ Elim}$$

$$\frac{\Gamma, x : \tau \vdash_2 M : \sigma}{\Gamma \vdash_2 \lambda x.M : \tau \multimap \sigma} \tag{$\multimap$ Intro}$$

$$\frac{\Gamma_1 \vdash_2 M_1 : \tau \multimap \sigma \qquad \Gamma_2 \vdash_2 M_2 : \tau}{\Gamma_1, \Gamma_2 \vdash_2 M_1 M_2 : \sigma} \tag{$\multimap$ Elim}$$

Example 3.2.1. Let us write $\overset{\circ}{\alpha}$ for the type $(\alpha \multimap \alpha)$. For the λ-term $(\lambda x.xx)(\lambda y.y)$, the following derivation is obtained:

$$\dfrac{\dfrac{\dfrac{\dfrac{[x_1 : \overset{\circ}{\alpha} \multimap \overset{\circ}{\alpha}] \vdash_2 x_1 : \overset{\circ}{\alpha} \multimap \overset{\circ}{\alpha} \qquad [x_2 : \overset{\circ}{\alpha}] \vdash_2 x_2 : \overset{\circ}{\alpha}}{[x_1 : \overset{\circ}{\alpha} \multimap \overset{\circ}{\alpha}, x_2 : \overset{\circ}{\alpha}] \vdash_2 x_1 x_2 : \overset{\circ}{\alpha}}}{[x : (\overset{\circ}{\alpha} \multimap \overset{\circ}{\alpha}) \cap \overset{\circ}{\alpha}] \vdash_2 xx : \overset{\circ}{\alpha}}}{[\,] \vdash_2 \lambda x.xx : (\overset{\circ}{\alpha} \multimap \overset{\circ}{\alpha}) \cap \overset{\circ}{\alpha} \to \overset{\circ}{\alpha}} \qquad \dfrac{[y : \overset{\circ}{\alpha}] \vdash_2 y : \overset{\circ}{\alpha}}{[\,] \vdash_2 \lambda y.y : \overset{\circ}{\alpha} \multimap \overset{\circ}{\alpha}} \qquad \dfrac{[y : \alpha] \vdash_2 y : \alpha}{[\,] \vdash_2 \lambda y.y : \overset{\circ}{\alpha}}}{[\,] \vdash_2 (\lambda x.xx)(\lambda y.y) : \overset{\circ}{\alpha}}$$

3.3 Type Inference Algorithm

In this section we define a new type inference algorithm for the λ-calculus (Definition 3.3.7), which is sound (Theorem 3.3.5) and complete (Theorem 3.3.8) with respect to the Linear Rank 2 Intersection Type System.

Our algorithm is based on Trevor Jim's type inference algorithm [19] for a Rank 2 Intersection Type System that was introduced by Daniel Leivant in [23], where the algorithm was briefly covered. Different versions of the algorithm were later defined by Steffen van Bakel in [27] and by Trevor Jim in [19].

Part of the definitions, properties and proofs here presented are also adapted from [19].

Definition 3.3.1 (Type-substitution)**.** We call *type-substitution* to

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

where $\alpha_1, \dots, \alpha_n$ are distinct type variables in $\mathbb{V}$ and $\tau_1, \dots, \tau_n$ are types in $\mathbb{T}_{\mathbb{L}0}$. For any τ in $\mathbb{T}_{\mathbb{L}0}$, $\mathbb{S}(\tau) = \tau[\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$ is the type obtained by simultaneously substituting α_i by τ_i in τ, with $1 \leq i \leq n$.

The type $\mathbb{S}(\tau)$ is called an *instance* of the type τ.

The notion of type-substitution can be extended to environments in the following way:

$$\mathbb{S}(\Gamma) = [x_1 : \mathbb{S}(\vec{\tau}_1), \dots, x_n : \mathbb{S}(\vec{\tau}_n)] \qquad \text{if } \Gamma = [x_1 : \vec{\tau}_1, \dots, x_n : \vec{\tau}_n]$$

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

If $\mathbb{S}_1 = [\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$ and $\mathbb{S}_2 = [\tau'_1/\alpha'_1, \dots, \tau'_n/\alpha'_n]$ are type-substitutions such that the variables $\alpha_1, \dots, \alpha_n, \alpha'_1, \dots, \alpha'_n$ are all distinct, then the type-substitution $\mathbb{S}_1 \cup \mathbb{S}_2$ is defined as $\mathbb{S}_1 \cup \mathbb{S}_2 = [\tau_1/\alpha_1, \dots, \tau_n/\alpha_n, \tau'_1/\alpha'_1, \dots, \tau'_n/\alpha'_n]$.

3.3.1 Unification

Definition 3.3.2 (Unification problem)**.** A *unification problem* is a finite set of equations $P = \{\tau_1 = \tau'_1, \dots, \tau_n = \tau'_n\}$. A *unifier* (or *solution*) is a substitution $\mathbb{S}$, such that $\mathbb{S}(\tau_i) = \mathbb{S}(\tau'_i)$, for $1 \leq i \leq n$. We call $\mathbb{S}(\tau_i)$ (or $\mathbb{S}(\tau'_i)$) a *common instance* of τ_i and τ'_i. P is *unifiable* if it has at least one unifier. $\mathcal{U}(P)$ is the set of unifiers of P.

Example 3.3.1. The types $\alpha_1 \multimap \alpha_2 \multimap \alpha_1$ and $(\alpha_3 \multimap \alpha_3) \multimap \alpha_4$ are *unifiable*. For the type-substitution $\mathbb{S} = [(\alpha_3 \multimap \alpha_3)/\alpha_1, (\alpha_2 \multimap (\alpha_3 \multimap \alpha_3))/\alpha_4]$, the *common instance* is $(\alpha_3 \multimap \alpha_3) \multimap \alpha_2 \multimap (\alpha_3 \multimap \alpha_3)$.

Definition 3.3.3 (Most general unifier)**.** A substitution $\mathbb{S}$ is a *most general unifier* (MGU) of P if $\mathbb{S}$ is a least element of $\mathcal{U}(P)$. That is,

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

Example 3.3.2. Consider the types $\tau_1 = (\alpha_1 \multimap \alpha_1)$ and $\tau_2 = (\alpha_2 \multimap \alpha_3)$.

The type-substitution $\mathbb{S}' = [(\alpha_4 \multimap \alpha_5)/\alpha_1, (\alpha_4 \multimap \alpha_5)/\alpha_2, (\alpha_4 \multimap \alpha_5)/\alpha_3]$ is a unifier of τ_1 and τ_2, but it is not the MGU.

The MGU of τ_1 and τ_2 is $\mathbb{S} = [\alpha_3/\alpha_1, \alpha_3/\alpha_2]$. The common instance of τ_1 and τ_2 by $\mathbb{S}'$, $(\alpha_4 \multimap \alpha_5) \multimap (\alpha_4 \multimap \alpha_5)$, is an instance of $(\alpha_3 \multimap \alpha_3)$, the common instance by $\mathbb{S}$.

Definition 3.3.4 (Solved form)**.** A unification problem $P = \{\alpha_1 = \tau_1, \ldots, \alpha_n = \tau_n\}$ is in *solved form* if $\alpha_1, \ldots, \alpha_n$ are all pairwise distinct variables that do not occur in any of the τ_i. In this case, we define $\mathbb{S}_P = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$.

Definition 3.3.5 (Type unification)**.** We define the following relation $\Rightarrow$ on type unification problems (for types in $\mathbb{T}_{\mathbb{L}0}$):

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

where $P[\tau/\alpha]$ corresponds to the notion of type-substitution extended to type unification problems. If $P = \{\tau_1 = \tau'_1, \ldots, \tau_n = \tau'_n\}$, then $P[\tau/\alpha] = \{\tau_1[\tau/\alpha] = \tau'_1[\tau/\alpha], \ldots, \tau_n[\tau/\alpha] = \tau'_n[\tau/\alpha]\}$. And $\mathsf{fv}(P)$ and $\mathsf{fv}(\tau)$ are the sets of free type variables in P and τ, respectively. Since in our system all occurrences of type variables are free, $\mathsf{fv}(P)$ and $\mathsf{fv}(\tau)$ are the sets of type variables in P and τ, respectively.

Definition 3.3.6 (Unification algorithm)**.** Let P be a unification problem (with types in $\mathbb{T}_{\mathbb{L}0}$). The unification function $\mathsf{UNIFY}(P)$ that decides whether P has a solution and, if so, returns the MGU of P (see [26]), is defined as:

```
function UNIFY(P)
    while P ⇒ P' do
        P := P';
    if P is in solved form then
        return S_P;
    else
        FAIL;
```

Example 3.3.3. Consider again the types $\alpha_1 \multimap \alpha_1$ and $\alpha_2 \multimap \alpha_3$ in Example 3.3.2. For the unification problem $P = \{\alpha_1 \multimap \alpha_1 = \alpha_2 \multimap \alpha_3\}$, $\mathsf{UNIFY}(P)$ performs the following transformations over P:

$$\begin{array}{rlcl}
\{\alpha_1 \multimap \alpha_1 = \alpha_2 \multimap \alpha_3\} & \Rightarrow \{\alpha_1 = \alpha_2, \alpha_1 = \alpha_3\} \cup \{\,\} & = & \{\alpha_1 = \alpha_2, \alpha_1 = \alpha_3\} \\
& \Rightarrow \{\alpha_1 = \alpha_2\} \cup \{\alpha_1 = \alpha_3\}[\alpha_2/\alpha_1] & = & \{\alpha_1 = \alpha_2, \alpha_2 = \alpha_3\} \\
& \Rightarrow \{\alpha_2 = \alpha_3\} \cup \{\alpha_1 = \alpha_2\}[\alpha_3/\alpha_2] & = & \{\alpha_1 = \alpha_3, \alpha_2 = \alpha_3\}
\end{array}$$

and, since $\{\alpha_1 = \alpha_3, \alpha_2 = \alpha_3\}$ is in solved form, it returns the type-substitution $[\alpha_3/\alpha_1, \alpha_3/\alpha_2]$.

3.3.2 Type Inference

Definition 3.3.7 (Type inference algorithm)**.** Let Γ be an environment, M a λ-term, σ a linear rank 2 intersection type and UNIFY the function in Definition 3.3.6. The function $\mathsf{T}(M) = (\Gamma, \sigma)$ defines a type inference algorithm for the λ-calculus in the Linear Rank 2 Intersection Type System, in the following way:

1. If $M = x$, then $\Gamma = [x : \alpha]$ and $\sigma = \alpha$, where α is a new variable;
2. If $M = \lambda x.M_1$ and $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ then:
   (a) if $x \notin \mathsf{dom}(\Gamma_1)$, then FAIL;
   (b) if $(x : \tau) \in \Gamma_1$, then $\mathsf{T}(M) = (\Gamma_{1x}, \tau \multimap \sigma_1)$;
   (c) if $(x : \tau_1 \cap \dots \cap \tau_n) \in \Gamma_1$ (with $n \geq 2$), then $\mathsf{T}(M) = (\Gamma_{1x}, \tau_1 \cap \dots \cap \tau_n \to \sigma_1)$.
3. If $M = M_1 M_2$, then:
   (a) if $\mathsf{T}(M_1) = (\Gamma_1, \alpha_1)$ and $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$,
   then $\mathsf{T}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3))$,
   where $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\})$ and α_2, α_3 are new variables;
   (b) if $\mathsf{T}(M_1) = (\Gamma'_1, \tau'_1 \cap \dots \cap \tau'_n \to \sigma'_1)$ (with $n \geq 2$) and, for each $1 \leq i \leq n$, $\mathsf{T}(M_2) = (\Gamma_i, \tau_i)$,
   then $\mathsf{T}(M) = (\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i), \mathbb{S}(\sigma'_1))$,
   where $\mathbb{S} = \mathsf{UNIFY}(\{\tau_i = \tau'_i \mid 1 \leq i \leq n\})$;
   (c) if $\mathsf{T}(M_1) = (\Gamma_1, \tau \multimap \sigma_1)$ and $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$,
   then $\mathsf{T}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\sigma_1))$,
   where $\mathbb{S} = \mathsf{UNIFY}(\{\tau_2 = \tau\})$;
   (d) otherwise FAIL.

Example 3.3.4. Let us show the type inference process for the λ-term $\lambda x.xx$.

- By rule 1., $\mathsf{T}(x) = ([x : \alpha_1], \alpha_1)$.
- By rule 1., again, $\mathsf{T}(x) = ([x : \alpha_2], \alpha_2)$.
- Then by rule 3.(a), $\mathsf{T}(xx) = (\mathbb{S}([x : \alpha_1] + [x : \alpha_2]), \mathbb{S}(\alpha_4)) = (\mathbb{S}([x : \alpha_1 \cap \alpha_2]), \mathbb{S}(\alpha_4))$,
  where $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 = \alpha_3\}) = [\alpha_3 \multimap \alpha_4/\alpha_1, \alpha_3/\alpha_2]$.
  So $\mathsf{T}(xx) = ([x : (\alpha_3 \multimap \alpha_4) \cap \alpha_3], \alpha_4)$.
- Finally, by rule 2.(c), $\mathsf{T}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4)$.

Example 3.3.5. Let us now show the type inference process for the λ-term $(\lambda x.xx)(\lambda y.y)$.

- From the previous example, we have $\mathsf{T}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4)$.
- By rules 1. and 2.(b), for the identity, the algorithm gives $\mathsf{T}(\lambda y.y) = ([\,], \alpha_1 \multimap \alpha_1)$.
- By rules 1. and 2.(b), again, for the identity, $\mathsf{T}(\lambda y.y) = ([\,], \alpha_2 \multimap \alpha_2)$.
- Then by rule 3.(b), $\mathsf{T}((\lambda x.xx)(\lambda y.y)) = (\mathbb{S}([\,] + [\,] + [\,]), \mathbb{S}(\alpha_4)) = ([\,], \mathbb{S}(\alpha_4))$,

  where $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\})$, calculated by performing the following transformations:

$$\begin{aligned}
\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\} &\Rightarrow \{\alpha_1 = \alpha_3, \alpha_1 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\
&\Rightarrow \{\alpha_1 = \alpha_3, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\
&\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_4\}\\
&\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_4 = \alpha_2 \multimap \alpha_2\}\\
&\Rightarrow \{\alpha_1 = \alpha_2 \multimap \alpha_2, \alpha_3 = \alpha_2 \multimap \alpha_2, \alpha_4 = \alpha_2 \multimap \alpha_2\}
\end{aligned}$$

  So $\mathbb{S} = [(\alpha_2 \multimap \alpha_2)/\alpha_1, (\alpha_2 \multimap \alpha_2)/\alpha_3, (\alpha_2 \multimap \alpha_2)/\alpha_4]$
  and $\mathsf{T}((\lambda x.xx)(\lambda y.y)) = ([\,], \alpha_2 \multimap \alpha_2)$.

Now we show several properties of our type system and type inference algorithm, in order to prove the soundness and completeness of the algorithm with respect to the system.

Notation 3.3.1. We write $\Phi \rhd \Gamma \vdash_2 M : \sigma$ if Φ is a derivation tree ending with $\Gamma \vdash_2 M : \sigma$. In this case, $|\Phi|$ is the length of the derivation tree Φ.

Lemma 3.3.1 (Substitution)**.** If $\Phi \rhd \Gamma \vdash_2 M : \sigma$, then $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$ for any substitution $\mathbb{S}$.

Proof. By induction on $|\Phi|$.

1. <u>(Axiom)</u>: Then $\Gamma = [x : \tau]$, $M = x$ and $\sigma = \tau$.

   So $\mathbb{S}(\Gamma) = [x : \mathbb{S}(\tau)]$ and $\mathbb{S}(\sigma) = \mathbb{S}(\tau)$,

   and by rule (Axiom) we have $\mathbb{S}(\Gamma) \vdash_2 x : \mathbb{S}(\sigma)$.

2. <u>(Exchange)</u>: Then $\Gamma = (\Gamma_1, y : \vec{\tau}_2, x : \vec{\tau}_1, \Gamma_2)$, $M = M_1$, $\sigma = \sigma_1$, and assuming that the premise $\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$ holds.

   By the induction hypothesis, for any substitution $\mathbb{S}$, $\mathbb{S}(\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$, which is the same as $\mathbb{S}(\Gamma_1), x : \mathbb{S}(\vec{\tau}_1), y : \mathbb{S}(\vec{\tau}_2), \mathbb{S}(\Gamma_2) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$.

By rule (Exchange) we get $\mathbb{S}(\Gamma_1), y : \mathbb{S}(\vec{\tau}_2), x : \mathbb{S}(\vec{\tau}_1), \mathbb{S}(\Gamma_2) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$,

which is the same as $\mathbb{S}(\Gamma_1, y : \vec{\tau}_2, x : \vec{\tau}_1, \Gamma_2) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$, i.e., $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$.

3. (Contraction): Then $\Gamma = (\Gamma_1, x : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)$, $M = M_1[x/x_1, x/x_2]$, $\sigma = \sigma_1$, and assuming that the premise $\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$ holds.

   By the induction hypothesis, for any substitution $\mathbb{S}$, $\mathbb{S}(\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$,

   which is the same as $\mathbb{S}(\Gamma_1), x_1 : \mathbb{S}(\vec{\tau}_1), x_2 : \mathbb{S}(\vec{\tau}_2), \mathbb{S}(\Gamma_2) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$.

   By rule (Contraction) we get $\mathbb{S}(\Gamma_1), x : \mathbb{S}(\vec{\tau}_1) \cap \mathbb{S}(\vec{\tau}_2), \mathbb{S}(\Gamma_2) \vdash_2 M_1[x/x_1, x/x_2] : \mathbb{S}(\sigma_1)$,

   which is the same as $\mathbb{S}(\Gamma_1, x : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2) \vdash_2 M_1[x/x_1, x/x_2] : \mathbb{S}(\sigma_1)$, i.e., $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$.

4. ($\rightarrow$ Intro): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $\sigma = \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$, and assuming that the premise $\Gamma_1, x : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M_1 : \sigma_1$ (with $n \geq 2$) holds.

   By the induction hypothesis, for any substitution $\mathbb{S}$, $\mathbb{S}(\Gamma_1, x : \tau_1 \cap \cdots \cap \tau_n) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$,

   which is the same as $\mathbb{S}(\Gamma_1), x : \mathbb{S}(\tau_1) \cap \cdots \cap \mathbb{S}(\tau_n) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$.

   By rule ($\rightarrow$ Intro) we get $\mathbb{S}(\Gamma_1) \vdash_2 \lambda x.M_1 : \mathbb{S}(\tau_1) \cap \cdots \cap \mathbb{S}(\tau_n) \rightarrow \mathbb{S}(\sigma_1)$,

   which is the same as $\mathbb{S}(\Gamma_1) \vdash_2 \lambda x.M_1 : \mathbb{S}(\tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1)$, i.e., $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$.

5. ($\rightarrow$ Elim): Then $\Gamma = (\Gamma_0, \sum_{i=1}^{n} \Gamma_i)$, $M = M_1 M_2$, $\sigma = \sigma_1$, and assuming that the premises $\Gamma_0 \vdash_2 M_1 : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$ and $\Gamma_i \vdash_2 M_2 : \tau_i$, for $1 \leq i \leq n$ (with $n \geq 2$), hold.

   By the induction hypothesis, for any substitution $\mathbb{S}$:

   - $\mathbb{S}(\Gamma_0) \vdash_2 M_1 : \mathbb{S}(\tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1)$,
     which is the same as $\mathbb{S}(\Gamma_0) \vdash_2 M_1 : \mathbb{S}(\tau_1) \cap \cdots \cap \mathbb{S}(\tau_n) \rightarrow \mathbb{S}(\sigma_1)$;
   - $\mathbb{S}(\Gamma_i) \vdash_2 M_2 : \mathbb{S}(\tau_i)$, for $1 \leq i \leq n$.

   By rule ($\rightarrow$ Elim) we get $\mathbb{S}(\Gamma_0), \sum_{i=1}^{n} \mathbb{S}(\Gamma_i) \vdash_2 M_1 M_2 : \mathbb{S}(\sigma_1)$,

   which is the same as $\mathbb{S}(\Gamma_0, \sum_{i=1}^{n} \Gamma_i) \vdash_2 M_1 M_2 : \mathbb{S}(\sigma_1)$, i.e., $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$.

6. ($\multimap$ Intro): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $\sigma = \tau \multimap \sigma_1$, and assuming that the premise $\Gamma_1, x : \tau \vdash_2 M_1 : \sigma_1$ holds.

   By the induction hypothesis, for any substitution $\mathbb{S}$, $\mathbb{S}(\Gamma_1, x : \tau) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$, which is the same as $\mathbb{S}(\Gamma_1), x : \mathbb{S}(\tau) \vdash_2 M_1 : \mathbb{S}(\sigma_1)$.

By rule ($\multimap$ Intro) we get $\mathbb{S}(\Gamma_1) \vdash_2 \lambda x.M_1 : \mathbb{S}(\tau) \multimap \mathbb{S}(\sigma_1)$,

which is the same as $\mathbb{S}(\Gamma_1) \vdash_2 \lambda x.M_1 : \mathbb{S}(\tau \multimap \sigma_1)$, i.e., $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$.

7. ($\multimap$ Elim): Then $\Gamma = (\Gamma_1, \Gamma_2)$, $M = M_1M_2$, $\sigma = \sigma_1$, and assuming that the premises $\Gamma_1 \vdash_2 M_1 : \tau \multimap \sigma_1$ and $\Gamma_2 \vdash_2 M_2 : \tau$ hold.

   By the induction hypothesis, for any substitution $\mathbb{S}$:

   - $\mathbb{S}(\Gamma_1) \vdash_2 M_1 : \mathbb{S}(\tau \multimap \sigma_1)$, which is the same as $\mathbb{S}(\Gamma_1) \vdash_2 M_1 : \mathbb{S}(\tau) \multimap \mathbb{S}(\sigma_1)$;
   - $\mathbb{S}(\Gamma_2) \vdash_2 M_2 : \mathbb{S}(\tau)$.

   By rule ($\multimap$ Elim) we get $\mathbb{S}(\Gamma_1), \mathbb{S}(\Gamma_2) \vdash_2 M_1M_2 : \mathbb{S}(\sigma_1)$,

   which is the same as $\mathbb{S}(\Gamma_1, \Gamma_2) \vdash_2 M_1M_2 : \mathbb{S}(\sigma_1)$, i.e., $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$.

□

Lemma 3.3.2 (Relevance)**.** If $\Phi \rhd \Gamma \vdash_2 M : \sigma$, then $x \in \mathsf{dom}(\Gamma)$ if and only if $x \in \mathsf{FV}(M)$.

Proof. Easy induction on $|\Phi|$. □

Lemma 3.3.3. If $\mathsf{T}(M) = (\Gamma, \sigma)$, then $x \in \mathsf{dom}(\Gamma)$ if and only if $x \in \mathsf{FV}(M)$.

Proof. Easy induction on the definition of $\mathsf{T}(M)$. □

Corollary 3.3.3.1. From Lemma 3.3.2 and Lemma 3.3.3, it follows that if $\mathsf{T}(M) = (\Gamma, \sigma)$ and $\Gamma' \vdash_2 M : \sigma'$, then $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$.

Lemma 3.3.4. If $\Phi_1 \rhd \Gamma \vdash_2 M : \sigma$, $x \in \mathsf{FV}(M)$ and y does not occur in M, then $\Phi_2 \rhd \Gamma[y/x] \vdash_2 M[y/x] : \sigma$ and $|\Phi_1| = |\Phi_2|$.

Proof. By induction on $|\Phi_1|$.

(We will only prove the first part of the lemma, since the second ($|\Phi_1| = |\Phi_2|$) can be shown with a trivial induction proof.)

Let x be a variable that occurs free in M and y a new variable not occurring in M.

1. (Axiom): Then $\Gamma = [x_1 : \tau]$, $M = x_1$, $\sigma = \tau$ and $x = x_1$.

   By rule (Axiom) we have $[y : \tau] \vdash_2 y : \tau$, which is the same as $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

2. (Exchange): Then $\Gamma = (\Gamma_1, y_1 : \vec{\tau}_2, x_1 : \vec{\tau}_1, \Gamma_2)$, $M = M_1$, $\sigma = \sigma_1$, and assuming that the premise $\Gamma_1, x_1 : \vec{\tau}_1, y_1 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$ holds.

   Since $x \in \mathsf{FV}(M_1)$ and y does not occur in M_1,

   by induction, $(\Gamma_1, x_1 : \vec{\tau}_1, y_1 : \vec{\tau}_2, \Gamma_2)[y/x] \vdash_2 M_1[y/x] : \sigma_1$,

   which is the same as $(\Gamma_1[y/x]), x_1[y/x] : \vec{\tau}_1, y_1[y/x] : \vec{\tau}_2, (\Gamma_2[y/x]) \vdash_2 M_1[y/x] : \sigma_1$.

   Then by rule (Exchange), $(\Gamma_1[y/x]), y_1[y/x] : \vec{\tau}_2, x_1[y/x] : \vec{\tau}_1, (\Gamma_2[y/x]) \vdash_2 M_1[y/x] : \sigma_1$,

   which is the same as $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

3. (Contraction): Then $\Gamma = (\Gamma_1, x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)$, $M = M_1[x'/x_1, x'/x_2]$, $\sigma = \sigma_1$, and assuming that the premise $\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$ holds.

   There are two possible cases regarding x:

   (a) $x = x'$:

      Since y does not occur in M, $y \notin \mathsf{FV}(M)$, so by *Lemma* 3.3.2, $y \notin \mathsf{dom}(\Gamma)$. So $y \notin \mathsf{dom}(\Gamma_1)$ and $y \notin \mathsf{dom}(\Gamma_2)$.

      Then we can apply the rule (Contraction) to $\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$
      and get $\Gamma_1, y : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2 \vdash_2 M_1[y/x_1, y/x_2] : \sigma_1$,
      which is equivalent to $(\Gamma_1, x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)[y/x'] \vdash_2 (M_1[x'/x_1, x'/x_2])[y/x'] : \sigma_1$
      and the same as $(\Gamma_1, x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)[y/x] \vdash_2 (M_1[x'/x_1, x'/x_2])[y/x] : \sigma_1$.

      So $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

   (b) $x \neq x'$ (and so $x \in \mathsf{FV}(M_1)$):

      There are three possible cases regarding y:

      i. $y \neq x_1$ and $y \neq x_2$:

         Since $x \in \mathsf{FV}(M_1)$ and y does not occur in M_1,
         by induction, $(\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2)[y/x] \vdash_2 M_1[y/x] : \sigma_1$,
         which is the same as $(\Gamma_1[y/x]), x_1[y/x] : \vec{\tau}_1, x_2[y/x] : \vec{\tau}_2, (\Gamma_2[y/x]) \vdash_2 M_1[y/x] : \sigma_1$.

Then by rule (Contraction),
$(\Gamma_1[y/x]), x' : \vec{\tau}_1 \cap \vec{\tau}_2, (\Gamma_2[y/x]) \vdash_2 (M_1[y/x])[x'/(x_1[y/x]), x'/(x_2[y/x])] : \sigma_1$.

Since $x \neq x'$, $x' = x'[y/x]$.
So $(\Gamma_1[y/x]), x' : \vec{\tau}_1 \cap \vec{\tau}_2, (\Gamma_2[y/x]) = (\Gamma_1, x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)[y/x] = \Gamma[y/x]$.

And $x_1[y/x] = x_1$, $x_2[y/x] = x_2$ because $x \neq x_1, x \neq x_2$ (otherwise it would contradict the assumption that $x \in \mathsf{FV}(M)$),
so $(M_1[y/x])[x'/(x_1[y/x]), x'/(x_2[y/x])] = (M_1[y/x])[x'/x_1, x'/x_2]$.

And since $x \neq x_1$, $x \neq x_2$, $y \neq x_1$, $y \neq x_2$ and $x \neq x'$,
then $(M_1[y/x])[x'/x_1, x'/x_2] = (M_1[x'/x_1, x'/x_2])[y/x] = M[y/x]$.

So $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

ii. $y = x_1$:

So the premise can be written as $\Gamma_1, y : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$.
Let y' be a fresh variable not occurring in any of the terms and environments mentioned.

Then by induction, we have $(\Gamma_1, y : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2)[y'/y] \vdash_2 M_1[y'/y] : \sigma_1$,
which is the same as $\Gamma_1, y' : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1[y'/y] : \sigma_1$.

As $x \in \mathsf{FV}(M)$, by Lemma 3.3.2, $x \in \mathsf{dom}(\Gamma)$.
And since $x \neq x'$, then either $x \in \mathsf{dom}(\Gamma_1)$ or $x \in \mathsf{dom}(\Gamma_2)$.
This means that $x \in \mathsf{dom}(\Gamma_1, y' : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2)$, and so by *Lemma* 3.3.2, $x \in \mathsf{FV}(M_1[y'/y])$.

And y does not occur in $M_1[y'/y]$.

So we can then apply the induction hypothesis to the derivation ending with $\Gamma_1, y' : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1[y'/y] : \sigma_1$
and get $(\Gamma_1, y' : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2)[y/x] \vdash_2 (M_1[y'/y])[y/x] : \sigma_1$,
which is equivalent to $\Gamma_1[y/x], y' : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2[y/x] \vdash_2 (M_1[y'/y])[y/x] : \sigma_1$.

Then by rule (Contraction),
$\Gamma_1[y/x], x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2[y/x] \vdash_2 ((M_1[y'/y])[y/x])[x'/y', x'/x_2] : \sigma_1$,
which is the same as
$(\Gamma_1, x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)[y/x] \vdash_2 ((M_1[y'/x_1])[y/x])[x'/y', x'/x_2] : \sigma_1$.

This is equivalent to
$(\Gamma_1, x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)[y/x] \vdash_2 ((M_1[y'/x_1])[x'/y', x'/x_2])[y/x] : \sigma_1$,
which is the equivalent to
$(\Gamma_1, x' : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)[y/x] \vdash_2 ((M_1[x'/x_1, x'/x_2])[y/x] : \sigma_1$.

So $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

iii. $y = x_2$:

Analogous to the case where $y = x_1$.

4. ($\rightarrow$ Intro): Then $\Gamma = \Gamma_1$, $M = \lambda x_1.M_1$, $\sigma = \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$, and assuming that the premise $\Gamma_1, x_1 : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M_1 : \sigma_1$ (with $n \geq 2$) holds.

   Since $x \in \mathsf{FV}(M_1)$ and y does not occur in M_1,
   by induction, $(\Gamma_1, x_1 : \tau_1 \cap \cdots \cap \tau_n)[y/x] \vdash_2 M_1[y/x] : \sigma_1$,
   which is the same as $(\Gamma_1[y/x]), x_1[y/x] : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M_1[y/x] : \sigma_1$.

   Then by rule ($\rightarrow$ Intro), $\Gamma_1[y/x] \vdash_2 \lambda(x_1[y/x]).M_1[y/x] : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$.

   Since $x \neq x_1$ (otherwise it would contradict the assumption that $x \in \mathsf{FV}(M)$), $x_1[y/x] = x_1$ and $\lambda x_1.M_1[y/x] = (\lambda x_1.M_1)[y/x]$.

   So we have $\Gamma_1[y/x] \vdash_2 (\lambda x_1.M_1)[y/x] : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$,
   which is the same as $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

5. ($\rightarrow$ Elim): Then $\Gamma = (\Gamma', \sum_{i=1}^{n} \Gamma_i)$, $M = M_1 M_2$, $\sigma = \sigma_1$, and assuming that the premises $\Gamma' \vdash_2 M_1 : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$ and $\Gamma_i \vdash_2 M_2 : \tau_i$, for $1 \leq i \leq n$ (with $n \geq 2$), hold.

   Since $x \in \mathsf{FV}(M)$ and y does not occur in M, then y does not occur in M_1 nor in M_2 and there are three possible cases regarding x:

   (a) $x \in \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

   Then by induction, $\Gamma'[y/x] \vdash_2 M_1[y/x] : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$ and, for $1 \leq i \leq n$, $\Gamma_i[y/x] \vdash_2 M_2[y/x] : \tau_i$.

So by rule ($\rightarrow$ Elim), $\Gamma'[y/x], \sum_{i=1}^{n} \Gamma_i[y/x] \vdash_2 (M_1[y/x])(M_2[y/x]) : \sigma_1$, which is equivalent to $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

(b) $x \in \mathsf{FV}(M_1)$ and $x \notin \mathsf{FV}(M_2)$:

Then $M_2[y/x] = M_2$ and $\Gamma_i[y/x] = \Gamma_i$, for $1 \leq i \leq n$.
So $\Gamma_i[y/x] \vdash_2 M_2[y/x] : \tau_i$ is equivalent to $\Gamma_i \vdash_2 M_2 : \tau_i$, for $1 \leq i \leq n$.

By induction, $\Gamma'[y/x] \vdash_2 M_1[y/x] : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$.

So by rule ($\rightarrow$ Elim), $\Gamma'[y/x], \sum_{i=1}^{n} \Gamma_i[y/x] \vdash_2 (M_1[y/x])(M_2[y/x]) : \sigma_1$, which is equivalent to $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

(c) $x \notin \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

Then $M_1[y/x] = M_1$ and $\Gamma'[y/x] = \Gamma'$.
So $\Gamma'[y/x] \vdash_2 M_1[y/x] : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$ is equivalent to $\Gamma' \vdash_2 M_1 : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$.

By induction, $\Gamma_i[y/x] \vdash_2 M_2[y/x] : \tau_i$, for $1 \leq i \leq n$.

So by rule ($\rightarrow$ Elim), $\Gamma'[y/x], \sum_{i=1}^{n} \Gamma_i[y/x] \vdash_2 (M_1[y/x])(M_2[y/x]) : \sigma_1$, which is equivalent to $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

6. <u>($\multimap$ Intro)</u>: Then $\Gamma = \Gamma_1$, $M = \lambda x_1.M_1$, $\sigma = \tau \multimap \sigma_1$, and assuming that the premise $\Gamma_1, x_1 : \tau \vdash_2 M_1 : \sigma_1$ holds.

Since $x \in \mathsf{FV}(M_1)$ and y does not occur in M_1,
by induction, $(\Gamma_1, x_1 : \tau)[y/x] \vdash_2 M_1[y/x] : \sigma_1$,
which is the same as $(\Gamma_1[y/x]), x_1[y/x] : \tau \vdash_2 M_1[y/x] : \sigma_1$.

Then by rule ($\multimap$ Intro), $\Gamma_1[y/x] \vdash_2 \lambda(x_1[y/x]).M_1[y/x] : \tau \multimap \sigma_1$.

Since $x \neq x_1$ (otherwise it would contradict the assumption that $x \in \mathsf{FV}(M)$), $x_1[y/x] = x_1$ and $\lambda x_1.M_1[y/x] = (\lambda x_1.M_1)[y/x]$.

So we have $\Gamma_1[y/x] \vdash_2 (\lambda x_1.M_1)[y/x] : \tau \multimap \sigma_1$, which is the same as $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

7. (⊸ Elim): Then $\Gamma = (\Gamma_1, \Gamma_2)$, $M = M_1M_2$, $\sigma = \sigma_1$, and assuming that the premises $\Gamma_1 \vdash_2 M_1 : \tau \multimap \sigma_1$ and $\Gamma_2 \vdash_2 M_2 : \tau$ hold.

   Since $x \in \mathsf{FV}(M)$ and y does not occur in M, then y does not occur in M_1 nor in M_2 and there are three possible cases regarding x:

   (a) $x \in \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

   Then by induction, $\Gamma_1[y/x] \vdash_2 M_1[y/x] : \tau \multimap \sigma_1$ and $\Gamma_2[y/x] \vdash_2 M_2[y/x] : \tau$.

   So by rule (⊸ Elim), $\Gamma_1[y/x], \Gamma_2[y/x] \vdash_2 (M_1[y/x])(M_2[y/x]) : \sigma_1$, which is equivalent to $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

   (b) $x \in \mathsf{FV}(M_1)$ and $x \notin \mathsf{FV}(M_2)$:

   Then $M_2[y/x] = M_2$ and $\Gamma_2[y/x] = \Gamma_2$.
   So $\Gamma_2[y/x] \vdash_2 M_2[y/x] : \tau$ is equivalent to $\Gamma_2 \vdash_2 M_2 : \tau$.

   By induction, $\Gamma_1[y/x] \vdash_2 M_1[y/x] : \tau \multimap \sigma_1$.

   So by rule (⊸ Elim), $\Gamma_1[y/x], \Gamma_2[y/x] \vdash_2 (M_1[y/x])(M_2[y/x]) : \sigma_1$, which is equivalent to $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

   (c) $x \notin \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

   Then $M_1[y/x] = M_1$ and $\Gamma_1[y/x] = \Gamma_1$.
   So $\Gamma_1[y/x] \vdash_2 M_1[y/x] : \tau \multimap \sigma_1$ is equivalent to $\Gamma_1 \vdash_2 M_1 : \tau \multimap \sigma_1$.

   By induction, $\Gamma_2[y/x] \vdash_2 M_2[y/x] : \tau$.

   So by rule (⊸ Elim), $\Gamma_1[y/x], \Gamma_2[y/x] \vdash_2 (M_1[y/x])(M_2[y/x]) : \sigma_1$, which is equivalent to $\Gamma[y/x] \vdash_2 M[y/x] : \sigma$.

□

Corollary 3.3.4.1. From Lemma 3.3.4, it follows that if $\Gamma \vdash_2 M : \sigma$, $\{x_1, \dots, x_n\} \subseteq \mathsf{FV}(M)$ and $y_1, \dots, y_n$ are all different variables not occurring in M, then $\Gamma[y_1/x_1, \dots, y_n/x_n] \vdash_2 M[y_1/x_1, \dots, y_n/x_n] : \sigma$.

Theorem 3.3.5 (Soundness)**.** If $\mathsf{T}(M) = (\Gamma, \sigma)$, then $\Gamma \vdash_2 M : \sigma$.

Proof. By induction on the definition of $\mathsf{T}(M)$.

1. If $M = x$, then $(\Gamma, \sigma) = ([x : \alpha], \alpha)$, and we have $\Gamma \vdash_2 x : \sigma$ by rule (Axiom).

2. If $M = \lambda x.M_1$, we have the following cases:

   (a) $x \in \mathsf{FV}(M_1)$ and $(\Gamma, \sigma) = (\Gamma_{1x}, \Gamma_1(x) \multimap \sigma_1)$, where $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ and $\Gamma_1(x) = \tau \in \mathbb{T}_{\mathbb{L}0}$.

   By induction, $\Gamma_1 \vdash_2 M_1 : \sigma_1$, and by Lemma 3.3.3, $x \in \mathsf{dom}(\Gamma_1)$.

   So by applying the rule (Exchange) zero or more times successively, we obtain $\Gamma_{1x}, x : \tau \vdash_2 M_1 : \sigma_1$.

   So $\Gamma \vdash_2 \lambda x.M_1 : \sigma$ by rule ($\multimap$ Intro).

   (b) $x \in \mathsf{FV}(M_1)$ and $(\Gamma, \sigma) = (\Gamma_{1x}, \Gamma_1(x) \to \sigma_1)$, where $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ and $\Gamma_1(x) = \tau_1 \cap \cdots \cap \tau_n$, with $n \geq 2$.

   By induction, $\Gamma_1 \vdash_2 M_1 : \sigma_1$, and by Lemma 3.3.3, $x \in \mathsf{dom}(\Gamma_1)$.

   So by applying the rule (Exchange) zero or more times successively, we obtain $\Gamma_{1x}, x : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M_1 : \sigma_1$.

   So $\Gamma \vdash_2 \lambda x.M_1 : \sigma$ by rule ($\to$ Intro).

3. If $M = M_1 M_2$, we have the following cases:

   (a) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3))$, where $\mathsf{T}(M_1) = (\Gamma_1, \alpha_1)$, $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$, $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\})$ and α_2, α_3 do not occur in $\Gamma_1, \Gamma_2, \alpha_1, \tau_2$.

   By induction, $\Gamma_1 \vdash_2 M_1 : \alpha_1$ and $\Gamma_2 \vdash_2 M_2 : \tau_2$.

   Let $\mathcal{S}_1 = [y_1/x_1, \ldots, y_n/x_n]$ and $\mathcal{S}_2 = [z_1/x_1, \ldots, z_n/x_n]$, where $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \{x_1, \ldots, x_n\}$ (which by Lemma 3.3.2, occur free in M_1 and M_2) and $y_1, \ldots, y_n, z_1, \ldots, z_n$ are all distinct fresh term variables, not occurring in M_1 nor in M_2 (and consequently, by Lemma 3.3.2, not occurring in Γ_1 nor in Γ_2).

By Corollary 3.3.4.1, $\mathcal{S}_1(\Gamma_1) \vdash_2 \mathcal{S}_1(M_1) : \alpha_1$ and $\mathcal{S}_2(\Gamma_2) \vdash_2 \mathcal{S}_2(M_2) : \tau_2$.

By Lemma 3.3.1, $\mathbb{S}(\mathcal{S}_1(\Gamma_1)) \vdash_2 \mathcal{S}_1(M_1) : \mathbb{S}(\alpha_1)$ and $\mathbb{S}(\mathcal{S}_2(\Gamma_2)) \vdash_2 \mathcal{S}_2(M_2) : \mathbb{S}(\tau_2)$.

Since $\mathbb{S}(\tau_2) = \mathbb{S}(\alpha_2)$, $\mathbb{S}(\alpha_1) = \mathbb{S}(\alpha_2) \multimap \mathbb{S}(\alpha_3)$ and $(\mathcal{S}_1(\Gamma_1), \mathcal{S}_2(\Gamma_2))$ is consistent, by rule ($\multimap$ Elim) we have $(\mathbb{S}(\mathcal{S}_1(\Gamma_1)), \mathbb{S}(\mathcal{S}_2(\Gamma_2))) \vdash_2 (\mathcal{S}_1(M_1))(\mathcal{S}_2(M_2)) : \mathbb{S}(\alpha_3)$, which is the same as $\mathbb{S}(\mathcal{S}_1(\Gamma_1), \mathcal{S}_2(\Gamma_2)) \vdash_2 (\mathcal{S}_1(M_1))(\mathcal{S}_2(M_2)) : \mathbb{S}(\alpha_3)$.

For each pair $(y_i : \vec{\tau}_i, z_i : \vec{\tau}'_i)$ (for $1 \leq i \leq n$) in the environment $\mathbb{S}(\mathcal{S}_1(\Gamma_1), \mathcal{S}_2(\Gamma_2))$ in the previous derivation, let us apply the rule (Contraction) to obtain the environment with $x_i : \vec{\tau}_i \cap \vec{\tau}'_i$ instead (and applying the rule (Exchange) as necessary).

After these applications of the rules (Contraction) and (Exchange) (and consequent applications of (Exchange), if necessary), and by looking at the definition of (+), we end up with $\mathbb{S}(\Gamma_1 + \Gamma_2) \vdash_2 M_1 M_2 : \mathbb{S}(\alpha_3)$.

(b) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma' + \sum_{i=1}^{n} \Gamma_i), \mathbb{S}(\sigma'_1))$, where $\mathsf{T}(M_1) = (\Gamma', \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1)$, with $n \geq 2$, $\mathsf{T}(M_2) = (\Gamma_i, \tau_i)$ for $1 \leq i \leq n$, and $\mathbb{S} = \mathsf{UNIFY}(\{\tau_i = \tau'_i \mid 1 \leq i \leq n\})$.

By induction, $\Gamma' \vdash_2 M_1 : \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1$
and $\Gamma_i \vdash_2 M_2 : \tau_i$ (for $1 \leq i \leq n$).
Note that $\mathsf{dom}(\Gamma_1) = \mathsf{dom}(\Gamma_2) = \cdots = \mathsf{dom}(\Gamma_{n-1}) = \mathsf{dom}(\Gamma_n)$.

Let $\mathcal{S}_1 = [y_1/x_1, \ldots, y_n/x_n]$ and $\mathcal{S}_2 = [z_1/x_1, \ldots, z_n/x_n]$, where $\mathsf{dom}(\Gamma') \cap \mathsf{dom}(\Gamma_1) = \{x_1, \ldots, x_n\}$ (which by Lemma 3.3.2, occur free in M_1 and in M_2) and $y_1, \ldots, y_n, z_1, \ldots, z_n$ are all distinct fresh term variables, not occurring in M_1 nor in M_2 (and consequently, by Lemma 3.3.2, not occurring in Γ' nor in Γ_i, for all $1 \leq i \leq n$).

By Corollary 3.3.4.1, $\mathcal{S}_1(\Gamma') \vdash_2 \mathcal{S}_1(M_1) : \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1$
and $\mathcal{S}_2(\Gamma_i) \vdash_2 \mathcal{S}_2(M_2) : \tau_i$ (for $1 \leq i \leq n$).

By Lemma 3.3.1, $\mathbb{S}(\mathcal{S}_1(\Gamma')) \vdash_2 \mathcal{S}_1(M_1) : \mathbb{S}(\tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1)$
and $\mathbb{S}(\mathcal{S}_2(\Gamma_i)) \vdash_2 \mathcal{S}_2(M_2) : \mathbb{S}(\tau_i)$ (for $1 \leq i \leq n$).

Since $\mathbb{S}(\tau_i) = \mathbb{S}(\tau'_i)$ for all $1 \leq i \leq n$ and $(\mathcal{S}_1(\Gamma'), (\mathcal{S}_2(\Gamma_1) + \cdots + \mathcal{S}_2(\Gamma_n)))$ is consistent, by rule ($\to$ Elim) we have $(\mathbb{S}(\mathcal{S}_1(\Gamma')), (\mathbb{S}(\mathcal{S}_2(\Gamma_1)) + \cdots + \mathbb{S}(\mathcal{S}_2(\Gamma_n)))) \vdash_2 (\mathcal{S}_1(M_1))(\mathcal{S}_2(M_2)) : \mathbb{S}(\sigma'_1)$,
which is the same as $\mathbb{S}(\mathcal{S}_1(\Gamma'), (\mathcal{S}_2(\Gamma_1) + \cdots + \mathcal{S}_2(\Gamma_n))) \vdash_2 (\mathcal{S}_1(M_1))(\mathcal{S}_2(M_2)) : \mathbb{S}(\sigma'_1)$.

For each pair $(y_i : \vec{\tau}_i, z_i : \vec{\tau}'_i)$ (for $1 \leq i \leq n$) in the environment $\mathbb{S}(\mathcal{S}_1(\Gamma'), (\mathcal{S}_2(\Gamma_1) + \cdots + \mathcal{S}_2(\Gamma_n)))$ in the previous derivation, let us apply the rule (Contraction) to obtain the environment with $x_i : \vec{\tau}_i \cap \vec{\tau}'_i$ instead (and applying the rule (Exchange) as necessary).

After these applications of the rules (Contraction) and (Exchange) (and consequent applications of (Exchange), if necessary), and by looking at the definition of $(+)$, we end up with $\mathbb{S}(\Gamma' + \sum_{i=1}^{n} \Gamma_i) \vdash_2 M_1 M_2 : \mathbb{S}(\sigma'_1)$.

(c) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\sigma_1))$, where $\mathsf{T}(M_1) = (\Gamma_1, \tau \multimap \sigma_1)$, $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$ and $\mathbb{S} = \mathsf{UNIFY}(\{\tau_2 = \tau\})$.

By induction, $\Gamma_1 \vdash_2 M_1 : \tau \multimap \sigma_1$ and $\Gamma_2 \vdash_2 M_2 : \tau_2$.

Let $\mathcal{S}_1 = [y_1/x_1, \ldots, y_n/x_n]$ and $\mathcal{S}_2 = [z_1/x_1, \ldots, z_n/x_n]$, where $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \{x_1, \ldots, x_n\}$ (which by Lemma 3.3.2, occur free in M_1 and M_2) and $y_1, \ldots, y_n, z_1, \ldots, z_n$ are all distinct fresh term variables, not occurring in M_1 nor in M_2 (and consequently, by Lemma 3.3.2, not occurring in Γ_1 nor in Γ_2).

By Corollary 3.3.4.1, $\mathcal{S}_1(\Gamma_1) \vdash_2 \mathcal{S}_1(M_1) : \tau \multimap \sigma_1$ and $\mathcal{S}_2(\Gamma_2) \vdash_2 \mathcal{S}_2(M_2) : \tau_2$.

By Lemma 3.3.1, $\mathbb{S}(\mathcal{S}_1(\Gamma_1)) \vdash_2 \mathcal{S}_1(M_1) : \mathbb{S}(\tau \multimap \sigma_1)$ and $\mathbb{S}(\mathcal{S}_2(\Gamma_2)) \vdash_2 \mathcal{S}_2(M_2) : \mathbb{S}(\tau_2)$.

Since $\mathbb{S}(\tau_2) = \mathbb{S}(\tau)$ and $(\mathcal{S}_1(\Gamma_1), \mathcal{S}_2(\Gamma_2))$ is consistent,
by rule ($\multimap$ Elim) we have $(\mathbb{S}(\mathcal{S}_1(\Gamma_1)), \mathbb{S}(\mathcal{S}_2(\Gamma_2))) \vdash_2 (\mathcal{S}_1(M_1))(\mathcal{S}_2(M_2)) : \mathbb{S}(\sigma_1)$,
which is the same as $\mathbb{S}(\mathcal{S}_1(\Gamma_1), \mathcal{S}_2(\Gamma_2)) \vdash_2 (\mathcal{S}_1(M_1))(\mathcal{S}_2(M_2)) : \mathbb{S}(\sigma_1)$.

For each pair $(y_i : \vec{\tau}_i, z_i : \vec{\tau}'_i)$ (for $1 \leq i \leq n$) in the environment $\mathbb{S}(\mathcal{S}_1(\Gamma_1), \mathcal{S}_2(\Gamma_2))$ in the previous derivation, let us apply the rule (Contraction) to obtain the environment with $x_i : \vec{\tau}_i \cap \vec{\tau}'_i$ instead (and applying the rule (Exchange) as necessary).

After these applications of the rules (Contraction) and (Exchange) (and consequent applications of (Exchange), if necessary), and by looking at the definition of $(+)$, we end up with $\mathbb{S}(\Gamma_1 + \Gamma_2) \vdash_2 M_1 M_2 : \mathbb{S}(\sigma_1)$.

For any other possible case, the algorithm fails (by rules 2.(a) and 3.(d)), thus making the left side of the implication $(\mathsf{T}(M) = (\Gamma, \sigma))$ false, which makes the statement true. $\square$

Lemma 3.3.6. If $\mathsf{T}(M) = (\Gamma, \sigma)$, $x \in \mathsf{FV}(M)$ and y does not occur in M, then $\mathsf{T}(M[y/x]) = (\Gamma[y/x], \sigma)$.

Proof. By induction on the definition of $\mathsf{T}(M)$.

1. If $M = x_1$ and let $x = x_1$ and $y \neq x_1$, then $(\Gamma, \sigma) = ([x_1 : \alpha], \alpha)$ and $\mathsf{T}(M[y/x]) = \mathsf{T}(M[y/x_1]) = \mathsf{T}(y) = ([y : \alpha], \alpha) = (\Gamma[y/x], \sigma)$.

   (Note that we can choose the same type variable α from $\mathsf{T}(M)$ in $\mathsf{T}(y)$ as these are independent, so α is fresh in $\mathsf{T}(y)$.)

2. If $M = \lambda x_1.M_1$ and let x be a variable that occurs free in M and y a new variable not occurring in M, we have the following cases:

   (a) $(\Gamma, \sigma) = (\Gamma_{1x_1}, \Gamma_1(x_1) \multimap \sigma_1)$, where $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ and $\Gamma_1(x_1) = \tau \in \mathbb{T}_{\mathbb{L}0}$.

   Since $x \in \mathsf{FV}(M_1)$ and y does not occur in M_1 (otherwise it would contradict the assumption that $x \in \mathsf{FV}(M)$ and y does not occur in M),
   by induction, $\mathsf{T}(M_1[y/x]) = (\Gamma_1[y/x], \sigma_1)$.

   And $(\Gamma_1[y/x])(x_1) = \Gamma_1(x_1) = \tau \in \mathbb{T}_{\mathbb{L}0}$.

   So by rule 2.(b) of the inference algorithm, $\mathsf{T}(\lambda x_1.(M_1[y/x])) = ((\Gamma_1[y/x])_{x_1}, \tau \multimap \sigma_1)$.

   And $M[y/x] = (\lambda x_1.M_1)[y/x] = \lambda x_1.(M_1[y/x])$, so

$$\begin{aligned}
\mathsf{T}(M[y/x]) &= \mathsf{T}(\lambda x_1.(M_1[y/x])) \\
&= ((\Gamma_1[y/x])_{x_1}, \tau \multimap \sigma_1) \\
&= (\Gamma_{1x_1}[y/x], \Gamma_1(x_1) \multimap \sigma_1) \\
&= (\Gamma[y/x], \sigma).
\end{aligned}$$

   (b) $(\Gamma, \sigma) = (\Gamma_{1x_1}, \Gamma_1(x_1) \to \sigma_1)$, where $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ and $\Gamma_1(x_1) = \tau_1 \cap \cdots \cap \tau_n$, with $n \geq 2$.

   Since $x \in \mathsf{FV}(M_1)$ and y does not occur in M_1, by induction, $\mathsf{T}(M_1[y/x]) = (\Gamma_1[y/x], \sigma_1)$.

   And $(\Gamma_1[y/x])(x_1) = \Gamma_1(x_1) = \tau_1 \cap \cdots \cap \tau_n$.

So by rule 2.(c) of the inference algorithm, $\mathsf{T}(\lambda x_1.(M_1[y/x])) = ((\Gamma_1[y/x])_{x_1}, \tau_1 \cap \cdots \cap \tau_n \to \sigma_1)$.

And $M[y/x] = (\lambda x_1.M_1)[y/x] = \lambda x_1.(M_1[y/x])$, so

$$\begin{aligned}\mathsf{T}(M[y/x]) &= \mathsf{T}(\lambda x_1.(M_1[y/x]))\\ &= ((\Gamma_1[y/x])_{x_1}, \tau_1 \cap \cdots \cap \tau_n \to \sigma_1)\\ &= (\Gamma_{1x_1}[y/x], \Gamma_1(x_1) \to \sigma_1)\\ &= (\Gamma[y/x], \sigma).\end{aligned}$$

3. If $M = M_1M_2$ and let x be a variable that occurs free in M and y a new variable not occurring in M, we have the following cases:

   (a) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3))$, where $\mathsf{T}(M_1) = (\Gamma_1, \alpha_1)$, $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$, $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\})$ and α_2, α_3 do not occur in $\Gamma_1, \Gamma_2, \alpha_1, \tau_2$.

   Since $x \in \mathsf{FV}(M)$ and y does not occur in M, then y does not occur in M_1 nor in M_2 and there are three possible cases regarding x:

   i. $x \in \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

   Then by induction, $\mathsf{T}(M_1[y/x]) = (\Gamma_1[y/x], \alpha_1)$ and $\mathsf{T}(M_2[y/x]) = (\Gamma_2[y/x], \tau_2)$.

   So by rule 3.(a) of the inference algorithm,
   $\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\alpha_3))$.

   (As before, as well as in the following cases, note that we can choose the same type variables α_2, α_3 (and, consequently, the same $\mathbb{S}$) in $\mathsf{T}((M_1[y/x])(M_2[y/x]))$ because they are fresh in this inference and, since they do not occur in Γ_1 and Γ_2, they also do not occur in $\Gamma_1[y/x]$ and $\Gamma_2[y/x]$ (nor in α_1, τ_2).)

   And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x]))\\ &= (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\alpha_3))\\ &= (\mathbb{S}((\Gamma_1 + \Gamma_2)[y/x]), \mathbb{S}(\alpha_3))\\ &= ((\mathbb{S}(\Gamma_1 + \Gamma_2))[y/x], \mathbb{S}(\alpha_3))\\ &= (\Gamma[y/x], \sigma).\end{aligned}$$

   ii. $x \in \mathsf{FV}(M_1)$ and $x \notin \mathsf{FV}(M_2)$:

   Then $M_2[y/x] = M_2$ and $\Gamma_2[y/x] = \Gamma_2$.

So $\mathsf{T}(M_2[y/x]) = \mathsf{T}(M_2) = (\Gamma_2, \tau_2) = (\Gamma_2[y/x], \tau_2)$.

By induction, $\mathsf{T}(M_1[y/x]) = (\Gamma_1[y/x], \alpha_1)$.

So by rule 3.(a) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\alpha_3))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}
\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x])) \\
&= (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\alpha_3)) \\
&= (\mathbb{S}((\Gamma_1 + \Gamma_2)[y/x]), \mathbb{S}(\alpha_3)) \\
&= ((\mathbb{S}(\Gamma_1 + \Gamma_2))[y/x], \mathbb{S}(\alpha_3)) \\
&= (\Gamma[y/x], \sigma).
\end{aligned}$$

iii. $x \notin \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

Then $M_1[y/x] = M_1$ and $\Gamma_1[y/x] = \Gamma_1$.
So $\mathsf{T}(M_1[y/x]) = \mathsf{T}(M_1) = (\Gamma_1, \alpha_1) = (\Gamma_1[y/x], \alpha_1)$.

By induction, $\mathsf{T}(M_2[y/x]) = (\Gamma_2[y/x], \tau_2)$.

So by rule 3.(a) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\alpha_3))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}
\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x])) \\
&= (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\alpha_3)) \\
&= (\mathbb{S}((\Gamma_1 + \Gamma_2)[y/x]), \mathbb{S}(\alpha_3)) \\
&= ((\mathbb{S}(\Gamma_1 + \Gamma_2))[y/x], \mathbb{S}(\alpha_3)) \\
&= (\Gamma[y/x], \sigma).
\end{aligned}$$

(b) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma' + \sum_{i=1}^{n} \Gamma_i), \mathbb{S}(\sigma'_1))$, where $\mathsf{T}(M_1) = (\Gamma', \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1)$, with $n \geq 2$, $\mathsf{T}(M_2) = (\Gamma_i, \tau_i)$ for $1 \leq i \leq n$, and $\mathbb{S} = \mathsf{UNIFY}(\{\tau_i = \tau'_i \mid 1 \leq i \leq n\})$.

Since $x \in \mathsf{FV}(M)$ and y does not occur in M, then y does not occur in M_1 nor in M_2 and there are three possible cases regarding x:

i. $x \in \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

Then by induction, $\mathsf{T}(M_1[y/x]) = (\Gamma'[y/x], \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1)$

and $\mathsf{T}(M_2[y/x]) = (\Gamma_i[y/x], \tau_i)$, for all $1 \leq i \leq n$.

So by rule 3.(b) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma'[y/x]) + \sum_{i=1}^{n}(\Gamma_i[y/x])), \mathbb{S}(\sigma_1'))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}
\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x])) \\
&= (\mathbb{S}((\Gamma'[y/x]) + \sum_{i=1}^{n}(\Gamma_i[y/x])), \mathbb{S}(\sigma_1')) \\
&= (\mathbb{S}((\Gamma' + \sum_{i=1}^{n}\Gamma_i)[y/x]), \mathbb{S}(\sigma_1')) \\
&= ((\mathbb{S}(\Gamma' + \sum_{i=1}^{n}\Gamma_i))[y/x], \mathbb{S}(\sigma_1')) \\
&= (\Gamma[y/x], \sigma).
\end{aligned}$$

ii. $x \in \mathsf{FV}(M_1)$ and $x \notin \mathsf{FV}(M_2)$:

Then $M_2[y/x] = M_2$ and $\Gamma_i[y/x] = \Gamma_i$, for all $1 \leq i \leq n$.
So $\mathsf{T}(M_2[y/x]) = \mathsf{T}(M_2) = (\Gamma_i, \tau_i) = (\Gamma_i[y/x], \tau_i)$, for all $1 \leq i \leq n$.

By induction, $\mathsf{T}(M_1[y/x]) = (\Gamma'[y/x], \tau_1' \cap \cdots \cap \tau_n' \to \sigma_1')$.

So by rule 3.(b) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma'[y/x]) + \sum_{i=1}^{n}(\Gamma_i[y/x])), \mathbb{S}(\sigma_1'))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}
\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x])) \\
&= (\mathbb{S}((\Gamma'[y/x]) + \sum_{i=1}^{n}(\Gamma_i[y/x])), \mathbb{S}(\sigma_1')) \\
&= (\mathbb{S}((\Gamma' + \sum_{i=1}^{n}\Gamma_i)[y/x]), \mathbb{S}(\sigma_1')) \\
&= ((\mathbb{S}(\Gamma' + \sum_{i=1}^{n}\Gamma_i))[y/x], \mathbb{S}(\sigma_1')) \\
&= (\Gamma[y/x], \sigma).
\end{aligned}$$

iii. $x \notin \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

Then $M_1[y/x] = M_1$ and $\Gamma'[y/x] = \Gamma'$.
So $\mathsf{T}(M_1[y/x]) = \mathsf{T}(M_1) = (\Gamma', \tau_1' \cap \cdots \cap \tau_n' \to \sigma_1') = (\Gamma'[y/x], \tau_1' \cap \cdots \cap \tau_n' \to \sigma_1')$.

By induction, $\mathsf{T}(M_2[y/x]) = (\Gamma_i[y/x], \tau_i)$, for all $1 \leq i \leq n$.

So by rule 3.(b) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma'[y/x]) + \sum_{i=1}^{n}(\Gamma_i[y/x])), \mathbb{S}(\sigma'_1))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}
\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x])) \\
&= (\mathbb{S}((\Gamma'[y/x]) + \sum_{i=1}^{n}(\Gamma_i[y/x])), \mathbb{S}(\sigma'_1)) \\
&= (\mathbb{S}((\Gamma' + \sum_{i=1}^{n}\Gamma_i)[y/x]), \mathbb{S}(\sigma'_1)) \\
&= ((\mathbb{S}(\Gamma' + \sum_{i=1}^{n}\Gamma_i))[y/x], \mathbb{S}(\sigma'_1)) \\
&= (\Gamma[y/x], \sigma).
\end{aligned}$$

(c) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\sigma_1))$, where $\mathsf{T}(M_1) = (\Gamma_1, \tau \multimap \sigma_1)$, $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$ and $\mathbb{S} = \mathsf{UNIFY}(\{\tau_2 = \tau\})$.

Since $x \in \mathsf{FV}(M)$ and y does not occur in M, then y does not occur in M_1 nor in M_2 and there are three possible cases regarding x:

i. $x \in \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

Then by induction, $\mathsf{T}(M_1[y/x]) = (\Gamma_1[y/x], \tau \multimap \sigma_1)$ and $\mathsf{T}(M_2[y/x]) = (\Gamma_2[y/x], \tau_2)$.

So by rule 3.(c) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\sigma_1))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}
\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x])) \\
&= (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\sigma_1)) \\
&= (\mathbb{S}((\Gamma_1 + \Gamma_2)[y/x]), \mathbb{S}(\sigma_1)) \\
&= ((\mathbb{S}(\Gamma_1 + \Gamma_2))[y/x], \mathbb{S}(\sigma_1)) \\
&= (\Gamma[y/x], \sigma).
\end{aligned}$$

ii. $x \in \mathsf{FV}(M_1)$ and $x \notin \mathsf{FV}(M_2)$:

Then $M_2[y/x] = M_2$ and $\Gamma_2[y/x] = \Gamma_2$.
So $\mathsf{T}(M_2[y/x]) = \mathsf{T}(M_2) = (\Gamma_2, \tau_2) = (\Gamma_2[y/x], \tau_2)$.

By induction, $\mathsf{T}(M_1[y/x]) = (\Gamma_1[y/x], \tau \multimap \sigma_1)$.

So by rule 3.(c) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\sigma_1))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x]))\\ &= (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\sigma_1))\\ &= (\mathbb{S}((\Gamma_1 + \Gamma_2)[y/x]), \mathbb{S}(\sigma_1))\\ &= ((\mathbb{S}(\Gamma_1 + \Gamma_2))[y/x], \mathbb{S}(\sigma_1))\\ &= (\Gamma[y/x], \sigma).\end{aligned}$$

iii. $x \notin \mathsf{FV}(M_1)$ and $x \in \mathsf{FV}(M_2)$:

Then $M_1[y/x] = M_1$ and $\Gamma_1[y/x] = \Gamma_1$.
So $\mathsf{T}(M_1[y/x]) = \mathsf{T}(M_1) = (\Gamma_1, \tau \multimap \sigma_1) = (\Gamma_1[y/x], \tau \multimap \sigma_1)$.

By induction, $\mathsf{T}(M_2[y/x]) = (\Gamma_2[y/x], \tau_2)$.

So by rule 3.(c) of the inference algorithm,
$\mathsf{T}((M_1[y/x])(M_2[y/x])) = (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\sigma_1))$.

And $M[y/x] = (M_1[y/x])(M_2[y/x])$, so

$$\begin{aligned}\mathsf{T}(M[y/x]) &= \mathsf{T}((M_1[y/x])(M_2[y/x]))\\ &= (\mathbb{S}((\Gamma_1[y/x]) + (\Gamma_2[y/x])), \mathbb{S}(\sigma_1))\\ &= (\mathbb{S}((\Gamma_1 + \Gamma_2)[y/x]), \mathbb{S}(\sigma_1))\\ &= ((\mathbb{S}(\Gamma_1 + \Gamma_2))[y/x], \mathbb{S}(\sigma_1))\\ &= (\Gamma[y/x], \sigma).\end{aligned}$$

Any other possible case makes the left side of the implication ($\mathsf{T}(M) = (\Gamma, \sigma)$, $x \in \mathsf{FV}(M)$ and y does not occur in M) false, which makes the statement true. □

Lemma 3.3.7. If $\mathsf{T}(M) = (\Gamma, \sigma)$, with $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, and y does not occur in M, then $\mathsf{T}(M[y/y_1, y/y_2]) = (\Gamma'', \sigma)$, with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Proof. By induction on the definition of $\mathsf{T}(M)$.

1. If $M = \lambda x_1.M_1$ and let y be a new variable not occurring in M, we have the following cases:

   (a) $(\Gamma, \sigma) = (\Gamma_{1x_1}, \Gamma_1(x_1) \multimap \sigma_1)$, with $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, where $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ and $\Gamma_1(x_1) = \tau \in \mathbb{T}_{\mathbb{L}0}$.

Since $\Gamma_{1_{x_1}} \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, then $\Gamma_1 \equiv (\Gamma', x_1 : \tau, y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$.

And since y does not occur in M_1 (otherwise it would contradict the assumption that y does not occur in M),
by induction, $\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma'_1, \sigma_1)$,
with $\Gamma'_1 \equiv (\Gamma', x_1 : \tau, y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

So by rule 2.(b) of the inference algorithm,
$\mathsf{T}(\lambda x_1.(M_1[y/y_1, y/y_2])) = (\Gamma'_{1_{x_1}}, \tau \multimap \sigma_1)$.
And $\Gamma'_{1_{x_1}} \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \Gamma'_{1_{x_1}}$.

Also, $M[y/y_1, y/y_2] = \lambda x_1.(M_1[y/y_1, y/y_2])$, so

$$\begin{aligned} \mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(\lambda x_1.(M_1[y/y_1, y/y_2])) \\ &= (\Gamma'', \tau \multimap \sigma_1) \\ &= (\Gamma'', \sigma), \end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

(b) $(\Gamma, \sigma) = (\Gamma_{1_{x_1}}, \Gamma_1(x_1) \to \sigma_1)$, with $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, where $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ and $\Gamma_1(x_1) = \tau_1 \cap \cdots \cap \tau_n$, with $n \geq 2$.

Since $\Gamma_{1_{x_1}} \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, then $\Gamma_1 \equiv (\Gamma', x_1 : \tau_1 \cap \cdots \cap \tau_n, y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$.

And since y does not occur in M_1 (otherwise it would contradict the assumption that y does not occur in M),
by induction, $\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma'_1, \sigma_1)$,
with $\Gamma'_1 \equiv (\Gamma', x_1 : \tau_1 \cap \cdots \cap \tau_n, y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

So by rule 2.(c) of the inference algorithm,
$\mathsf{T}(\lambda x_1.(M_1[y/y_1, y/y_2])) = (\Gamma'_{1_{x_1}}, \tau_1 \cap \cdots \cap \tau_n \to \sigma_1)$.
And $\Gamma'_{1_{x_1}} \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \Gamma'_{1_{x_1}}$.

Also, $M[y/y_1, y/y_2] = \lambda x_1.(M_1[y/y_1, y/y_2])$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(\lambda x_1.(M_1[y/y_1, y/y_2])) \\ &= (\Gamma'', \tau_1 \cap \cdots \cap \tau_n \to \sigma_1) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

2. If $M = M_1 M_2$ and let y be a new variable not occurring in M, we have the following cases:

   (a) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3))$, with $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, where $\mathsf{T}(M_1) = (\Gamma_1, \alpha_1)$, $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$, $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\})$ and α_2, α_3 do not occur in $\Gamma_1, \Gamma_2, \alpha_1, \tau_2$.

   Because y does not occur in M, then y does not occur in M_1 nor in M_2.

   Since $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$ and M_1 is a term variable (otherwise its type given by the algorithm would not be a type variable), then there are five possible cases regarding the presence of y_1 and y_2 in $\mathsf{dom}(\Gamma_1)$ and $\mathsf{dom}(\Gamma_2)$:

   i. $y_1, y_2 \notin \mathsf{dom}(\Gamma_1)$ and $y_1, y_2 \in \mathsf{dom}(\Gamma_2)$:

   So $\Gamma_2 \equiv (\Gamma_2', y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ (for some $\vec{\tau}_3, \vec{\tau}_4$ such that $\mathbb{S}(\vec{\tau}_3) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4) = \vec{\tau}_2$).

   By induction,
   $$\mathsf{T}(M_2[y/y_1, y/y_2]) = (\Gamma_2'', \tau \multimap \tau_2), \tag{1}$$
   with $\Gamma_2'' \equiv (\Gamma_2', y : \vec{\tau}_3 \cap \vec{\tau}_4)$.

   And since $y_1, y_2 \notin \mathsf{dom}(\Gamma_1)$, by Lemma 3.3.3, $y_1, y_2 \notin \mathsf{FV}(M_1)$,
   so $M_1[y/y_1, y/y_2] = M_1$
   and then
   $$\mathsf{T}(M_1[y/y_1, y/y_2]) = \mathsf{T}(M_1) = (\Gamma_1, \alpha_1). \tag{2}$$

   So by rule 3.(a) of the inference algorithm (and (1), (2)),
   $\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma_1 + \Gamma_2''), \mathbb{S}(\alpha_3))$.

   (Note that we can choose the same type variables α_2, α_3 (and, consequently, the same $\mathbb{S}$) in $\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2])$ because they are fresh in this inference and, since they do not occur in Γ_2, they also do not occur in Γ_2'' (nor in Γ_1, α_1, τ_2). For analogous reasons, the same can and will be done in the following cases.)

Since $\Gamma_2'' \equiv (\Gamma_2', y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma_2 \equiv (\Gamma_2', y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$,
we have $\mathbb{S}(\Gamma_1 + \Gamma_2'') \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma_1 + \Gamma_2'')$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned} \mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\alpha_3)) \\ &= (\Gamma'', \sigma), \end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

ii. $y_1, y_2 \in \mathsf{dom}(\Gamma_2)$, $y_1 \in \mathsf{dom}(\Gamma_1)$ and $y_2 \notin \mathsf{dom}(\Gamma_1)$:

So $\Gamma_2 \equiv (\Gamma_2', y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\Gamma_1 \equiv (\Gamma_1', y_1 : \vec{\tau}_3')$ (for some $\vec{\tau}_3, \vec{\tau}_4, \vec{\tau}_3'$ such that $\mathbb{S}(\vec{\tau}_3' \cap \vec{\tau}_3) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4) = \vec{\tau}_2$).

By induction,

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = (\Gamma_2'', \tau_2), \tag{1}$$

with $\Gamma_2'' \equiv (\Gamma_2', y : \vec{\tau}_3 \cap \vec{\tau}_4)$.

Since $y_1 \in \mathsf{dom}(\Gamma_1)$, by Lemma 3.3.3, $y_1 \in \mathsf{FV}(M_1)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_1[y/y_1]) = (\Gamma_1[y/y_1], \alpha_1)$.

And since $y_2 \notin \mathsf{dom}(\Gamma_1)$, by Lemma 3.3.3, $y_2 \notin \mathsf{FV}(M_1)$,
so $M_1[y/y_1, y/y_2] = M_1[y/y_1]$
and then

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = \mathsf{T}(M_1[y/y_1]) = (\Gamma_1[y/y_1], \alpha_1). \tag{2}$$

So by rule 3.(a) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2''), \mathbb{S}(\alpha_3))$.

Since $\Gamma_2'' \equiv (\Gamma_2', y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma_2 \equiv (\Gamma_2', y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$, $\Gamma_1[y/y_1] \equiv (\Gamma_1', y : \vec{\tau}_3')$, $\Gamma_1 \equiv (\Gamma_1', y_1 : \vec{\tau}_3')$, $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$ and $\vec{\tau}_3' \cap (\vec{\tau}_3 \cap \vec{\tau}_4) = (\vec{\tau}_3' \cap \vec{\tau}_3) \cap \vec{\tau}_4$,
we have $\mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2'') \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2'')$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\alpha_3)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

iii. $y_1, y_2 \in \mathsf{dom}(\Gamma_2)$, $y_1 \notin \mathsf{dom}(\Gamma_1)$ and $y_2 \in \mathsf{dom}(\Gamma_1)$:

Analogous to the previous case.

iv. $y_1 \in \mathsf{dom}(\Gamma_1)$, $y_2 \notin \mathsf{dom}(\Gamma_1)$, $y_1 \notin \mathsf{dom}(\Gamma_2)$ and $y_2 \in \mathsf{dom}(\Gamma_2)$:

Since $y_1 \in \mathsf{dom}(\Gamma_1)$, by Lemma 3.3.3, $y_1 \in \mathsf{FV}(M_1)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_1[y/y_1]) = (\Gamma_1[y/y_1], \alpha_1)$.

And since $y_2 \notin \mathsf{dom}(\Gamma_1)$, by Lemma 3.3.3, $y_2 \notin \mathsf{FV}(M_1)$,
so $M_1[y/y_1, y/y_2] = M_1[y/y_1]$
and then

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = \mathsf{T}(M_1[y/y_1]) = (\Gamma_1[y/y_1], \alpha_1). \tag{1}$$

Since $y_2 \in \mathsf{dom}(\Gamma_2)$, by Lemma 3.3.3, $y_2 \in \mathsf{FV}(M_2)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_2[y/y_2]) = (\Gamma_2[y/y_2], \tau_2)$.

And since $y_1 \notin \mathsf{dom}(\Gamma_2)$, by Lemma 3.3.3, $y_1 \notin \mathsf{FV}(M_2)$,
so $M_2[y/y_1, y/y_2] = M_2[y/y_2]$
and then

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = \mathsf{T}(M_2[y/y_2]) = (\Gamma_2[y/y_2], \tau_2). \tag{2}$$

So by rule 3.(a) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2[y/y_2]), \mathbb{S}(\alpha_3))$.

Since $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$,
we have $\mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2[y/y_2]) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2[y/y_2])$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\alpha_3)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

v. $y_1 \notin \mathsf{dom}(\Gamma_1)$, $y_2 \in \mathsf{dom}(\Gamma_1)$, $y_1 \in \mathsf{dom}(\Gamma_2)$ and $y_2 \notin \mathsf{dom}(\Gamma_2)$:

Analogous to the previous case.

(b) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i), \mathbb{S}(\sigma'_1))$, with $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, where $\mathsf{T}(M_1) = (\Gamma'_1, \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1)$, with $n \geq 2$, $\mathsf{T}(M_2) = (\Gamma_i, \tau_i)$ for $1 \leq i \leq n$, and $\mathbb{S} = \mathsf{UNIFY}(\{\tau_i = \tau'_i \mid 1 \leq i \leq n\})$.

Because y does not occur in M, then y does not occur in M_1 nor in M_2.

Since $\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, then there are nine possible cases regarding the presence of y_1 and y_2 in $\mathsf{dom}(\Gamma'_1)$ and $\mathsf{dom}(\Gamma_i)$ (for all $1 \leq i \leq n$):

i. $y_1, y_2 \in \mathsf{dom}(\Gamma'_1)$ and $y_1, y_2 \notin \mathsf{dom}(\Gamma_i)$:

So $\Gamma'_1 \equiv (\Gamma''_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ (for some $\vec{\tau}_3, \vec{\tau}_4$ such that $\mathbb{S}(\vec{\tau}_3) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4) = \vec{\tau}_2$).

By induction,

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma'''_1, \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1), \tag{1}$$

with $\Gamma'''_1 \equiv (\Gamma''_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$.

And since $y_1, y_2 \notin \mathsf{dom}(\Gamma_i)$, by Lemma 3.3.3, $y_1, y_2 \notin \mathsf{FV}(M_2)$,
so $M_2[y/y_1, y/y_2] = M_2$
and then

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = \mathsf{T}(M_2) = (\Gamma_i, \tau_i). \tag{2}$$

So by rule 3.(b) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma_i), \mathbb{S}(\sigma'_1))$.

Since $\Gamma'''_1 \equiv (\Gamma''_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma'_1 \equiv (\Gamma''_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$,
we have $\mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma_i) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma_i)$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\sigma'_1)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

ii. $y_1, y_2 \notin \mathsf{dom}(\Gamma'_1)$ and $y_1, y_2 \in \mathsf{dom}(\Gamma_i)$:

Analogous to the previous case.

iii. $y_1, y_2 \in \mathsf{dom}(\Gamma'_1)$ and $y_1, y_2 \in \mathsf{dom}(\Gamma_i)$:

So $\Gamma'_1 \equiv (\Gamma''_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\Gamma_i \equiv (\Gamma'_i, y_1 : \vec{\tau}_{3_i}, y_2 : \vec{\tau}_{4_i})$ (for some $\vec{\tau}_3, \vec{\tau}_4, \vec{\tau}_{3_i}, \vec{\tau}_{4_i}$ such that $\mathbb{S}(\vec{\tau}_3 \cap \vec{\tau}_{3_1} \cap \cdots \cap \vec{\tau}_{3_n}) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4 \cap \vec{\tau}_{4_1} \cap \cdots \cap \vec{\tau}_{4_n}) = \vec{\tau}_2$).

By induction,

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma'''_1, \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1), \tag{1}$$

with $\Gamma'''_1 \equiv (\Gamma''_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$;

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = (\Gamma''_i, \tau_i), \tag{2}$$

with $\Gamma''_i \equiv (\Gamma'_i, y : \vec{\tau}_{3_i} \cap \vec{\tau}_{4_i})$.

So by rule 3.(b) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma''_i), \mathbb{S}(\sigma'_1))$.

Since $\Gamma'''_1 \equiv (\Gamma''_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma'_1 \equiv (\Gamma''_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$, $\Gamma''_i \equiv (\Gamma'_i, y : \vec{\tau}_{3_i} \cap \vec{\tau}_{4_i})$, $\Gamma_i \equiv (\Gamma'_i, y_1 : \vec{\tau}_{3_i}, y_2 : \vec{\tau}_{4_i})$, $\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$ and $(\vec{\tau}_3 \cap \vec{\tau}_4) \cap (\vec{\tau}_{3_1} \cap \vec{\tau}_{4_1}) \cap \cdots \cap (\vec{\tau}_{3_n} \cap \vec{\tau}_{4_n}) = (\vec{\tau}_3 \cap \vec{\tau}_{3_1} \cap \cdots \cap \vec{\tau}_{3_n}) \cap (\vec{\tau}_4 \cap \vec{\tau}_{4_1} \cap \cdots \cap \vec{\tau}_{4_n})$,
we have $\mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma''_i) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma''_i)$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\sigma'_1)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

iv. $y_1, y_2 \in \mathsf{dom}(\Gamma'_1)$, $y_1 \in \mathsf{dom}(\Gamma_i)$ and $y_2 \notin \mathsf{dom}(\Gamma_i)$:

So $\Gamma'_1 \equiv (\Gamma''_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\Gamma_i \equiv (\Gamma'_i, y_1 : \vec{\tau}_{3_i})$ (for some $\vec{\tau}_3, \vec{\tau}_4, \vec{\tau}_{3_i}$ such that $\mathbb{S}(\vec{\tau}_3 \cap \vec{\tau}_{3_1} \cap \cdots \cap \vec{\tau}_{3_n}) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4) = \vec{\tau}_2$).

By induction,

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma'''_1, \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1), \tag{1}$$

with $\Gamma'''_1 \equiv (\Gamma''_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$.

Since $y_1 \in \mathsf{dom}(\Gamma_i)$, by Lemma 3.3.3, $y_1 \in \mathsf{FV}(M_2)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_2[y/y_1]) = (\Gamma_i[y/y_1], \tau_i)$.

And since $y_2 \notin \mathsf{dom}(\Gamma_i)$, by Lemma 3.3.3, $y_2 \notin \mathsf{FV}(M_2)$,
so $M_2[y/y_1, y/y_2] = M_2[y/y_1]$
and then

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = \mathsf{T}(M_2[y/y_1]) = (\Gamma_i[y/y_1], \tau_i). \tag{2}$$

So by rule 3.(b) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma_i[y/y_1]), \mathbb{S}(\sigma'_1))$.

Since $\Gamma'''_1 \equiv (\Gamma''_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma'_1 \equiv (\Gamma''_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$, $\Gamma_i[y/y_1] \equiv (\Gamma'_i, y : \vec{\tau}_{3_i})$, $\Gamma_i \equiv (\Gamma'_i, y_1 : \vec{\tau}_{3_i})$, $\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$ and $(\vec{\tau}_3 \cap \vec{\tau}_4) \cap \vec{\tau}_{3_1} \cap \cdots \cap \vec{\tau}_{3_n} = (\vec{\tau}_3 \cap \vec{\tau}_{3_1} \cap \cdots \cap \vec{\tau}_{3_n}) \cap \vec{\tau}_4$,
we have $\mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma_i[y/y_1]) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma'''_1 + \sum_{i=1}^{n} \Gamma_i[y/y_1])$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\sigma'_1)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

v. $y_1, y_2 \in \mathsf{dom}(\Gamma'_1)$, $y_1 \notin \mathsf{dom}(\Gamma_i)$ and $y_2 \in \mathsf{dom}(\Gamma_i)$:

Analogous to the previous case.

vi. $y_1, y_2 \in \mathsf{dom}(\Gamma_i)$, $y_1 \in \mathsf{dom}(\Gamma'_1)$ and $y_2 \notin \mathsf{dom}(\Gamma'_1)$:

Analogous to the previous case.

vii. $y_1, y_2 \in \mathsf{dom}(\Gamma_i)$, $y_1 \notin \mathsf{dom}(\Gamma'_1)$ and $y_2 \in \mathsf{dom}(\Gamma'_1)$:

Analogous to the previous case.

viii. $y_1 \in \mathsf{dom}(\Gamma'_1)$, $y_2 \notin \mathsf{dom}(\Gamma'_1)$, $y_1 \notin \mathsf{dom}(\Gamma_i)$ and $y_2 \in \mathsf{dom}(\Gamma_i)$:

Since $y_1 \in \mathsf{dom}(\Gamma'_1)$, by Lemma 3.3.3, $y_1 \in \mathsf{FV}(M_1)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_1[y/y_1]) = (\Gamma'_1[y/y_1], \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1)$.

And since $y_2 \notin \mathsf{dom}(\Gamma'_1)$, by Lemma 3.3.3, $y_2 \notin \mathsf{FV}(M_1)$,
so $M_1[y/y_1, y/y_2] = M_1[y/y_1]$
and then

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = \mathsf{T}(M_1[y/y_1]) = (\Gamma'_1[y/y_1], \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1). \tag{1}$$

Since $y_2 \in \mathsf{dom}(\Gamma_i)$, by Lemma 3.3.3, $y_2 \in \mathsf{FV}(M_2)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_2[y/y_2]) = (\Gamma_i[y/y_2], \tau_i)$.

And since $y_1 \notin \mathsf{dom}(\Gamma_i)$, by Lemma 3.3.3, $y_1 \notin \mathsf{FV}(M_2)$,
so $M_2[y/y_1, y/y_2] = M_2[y/y_2]$
and then

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = \mathsf{T}(M_2[y/y_2]) = (\Gamma_i[y/y_2], \tau_i). \tag{2}$$

So by rule 3.(b) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma'_1[y/y_1] + \sum_{i=1}^n \Gamma_i[y/y_2]), \mathbb{S}(\sigma'_1))$.

Since $\mathbb{S}(\Gamma'_1 + \sum_{i=1}^n \Gamma_i) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$,
we have $\mathbb{S}(\Gamma'_1[y/y_1] + \sum_{i=1}^n \Gamma_i[y/y_2]) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma'_1[y/y_1] + \sum_{i=1}^n \Gamma_i[y/y_2])$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\sigma'_1)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

ix. $y_1 \notin \mathsf{dom}(\Gamma'_1)$, $y_2 \in \mathsf{dom}(\Gamma'_1)$, $y_1 \in \mathsf{dom}(\Gamma_i)$ and $y_2 \notin \mathsf{dom}(\Gamma_i)$:

Analogous to the previous case.

(c) $(\Gamma, \sigma) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\sigma_1))$, with $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, where $\mathsf{T}(M_1) = (\Gamma_1, \tau \multimap \sigma_1)$, $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$ and $\mathbb{S} = \mathsf{UNIFY}(\{\tau_2 = \tau\})$.

Because y does not occur in M, then y does not occur in M_1 nor in M_2.

Since $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, then there are nine possible cases regarding the presence of y_1 and y_2 in $\mathsf{dom}(\Gamma_1)$ and $\mathsf{dom}(\Gamma_2)$:

i. $y_1, y_2 \in \mathsf{dom}(\Gamma_1)$ and $y_1, y_2 \notin \mathsf{dom}(\Gamma_2)$:

So $\Gamma_1 \equiv (\Gamma'_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ (for some $\vec{\tau}_3, \vec{\tau}_4$ such that $\mathbb{S}(\vec{\tau}_3) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4) = \vec{\tau}_2$).

By induction,
$$\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma''_1, \tau \multimap \sigma_1), \tag{1}$$
with $\Gamma''_1 \equiv (\Gamma'_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$.

And since $y_1, y_2 \notin \mathsf{dom}(\Gamma_2)$, by Lemma 3.3.3, $y_1, y_2 \notin \mathsf{FV}(M_2)$,
so $M_2[y/y_1, y/y_2] = M_2$
and then
$$\mathsf{T}(M_2[y/y_1, y/y_2]) = \mathsf{T}(M_2) = (\Gamma_2, \tau_2). \tag{2}$$

So by rule 3.(c) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma''_1 + \Gamma_2), \mathbb{S}(\sigma_1))$.

Since $\Gamma''_1 \equiv (\Gamma'_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma_1 \equiv (\Gamma'_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$,
we have $\mathbb{S}(\Gamma''_1 + \Gamma_2) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma''_1 + \Gamma_2)$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\sigma_1)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

ii. $y_1, y_2 \notin \mathsf{dom}(\Gamma_1)$ and $y_1, y_2 \in \mathsf{dom}(\Gamma_2)$:

Analogous to the previous case.

iii. $y_1, y_2 \in \mathsf{dom}(\Gamma_1)$ and $y_1, y_2 \in \mathsf{dom}(\Gamma_2)$:

So $\Gamma_1 \equiv (\Gamma'_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\Gamma_2 \equiv (\Gamma'_2, y_1 : \vec{\tau}'_3, y_2 : \vec{\tau}'_4)$ (for some $\vec{\tau}_3, \vec{\tau}_4, \vec{\tau}'_3, \vec{\tau}'_4$ such that $\mathbb{S}(\vec{\tau}_3 \cap \vec{\tau}'_3) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4 \cap \vec{\tau}'_4) = \vec{\tau}_2$).

By induction,

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma''_1, \tau \multimap \sigma_1), \tag{1}$$

with $\Gamma''_1 \equiv (\Gamma'_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$;

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = (\Gamma''_2, \tau_2), \tag{2}$$

with $\Gamma''_2 \equiv (\Gamma'_2, y : \vec{\tau}'_3 \cap \vec{\tau}'_4)$.

So by rule 3.(c) of the inference algorithm (and (1), (2)), $\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma''_1 + \Gamma''_2), \mathbb{S}(\sigma_1))$.

Since $\Gamma''_1 \equiv (\Gamma'_1, y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma_1 \equiv (\Gamma'_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$, $\Gamma''_2 \equiv (\Gamma'_2, y : \vec{\tau}'_3 \cap \vec{\tau}'_4)$, $\Gamma_2 \equiv (\Gamma'_2, y_1 : \vec{\tau}'_3, y_2 : \vec{\tau}'_4)$, $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$ and $(\vec{\tau}_3 \cap \vec{\tau}_4) \cap (\vec{\tau}'_3 \cap \vec{\tau}'_4) = (\vec{\tau}_3 \cap \vec{\tau}'_3) \cap (\vec{\tau}_4 \cap \vec{\tau}'_4)$,
we have $\mathbb{S}(\Gamma''_1 + \Gamma''_2) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma''_1 + \Gamma''_2)$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned} \mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\sigma_1)) \\ &= (\Gamma'', \sigma), \end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

iv. $y_1, y_2 \in \mathsf{dom}(\Gamma_1)$, $y_1 \in \mathsf{dom}(\Gamma_2)$ and $y_2 \notin \mathsf{dom}(\Gamma_2)$:

So $\Gamma_1 \equiv (\Gamma'_1, y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$ and $\Gamma_2 \equiv (\Gamma'_2, y_1 : \vec{\tau}'_3)$ (for some $\vec{\tau}_3, \vec{\tau}_4, \vec{\tau}'_3$ such that $\mathbb{S}(\vec{\tau}_3 \cap \vec{\tau}'_3) = \vec{\tau}_1$ and $\mathbb{S}(\vec{\tau}_4) = \vec{\tau}_2$).

By induction,

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = (\Gamma''_1, \tau \multimap \sigma_1), \tag{1}$$

with $\Gamma_1'' \equiv (\Gamma_1', y : \vec{\tau}_3 \cap \vec{\tau}_4)$.

Since $y_1 \in \mathsf{dom}(\Gamma_2)$, by Lemma 3.3.3, $y_1 \in \mathsf{FV}(M_2)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_2[y/y_1]) = (\Gamma_2[y/y_1], \tau_2)$.

And since $y_2 \notin \mathsf{dom}(\Gamma_2)$, by Lemma 3.3.3, $y_2 \notin \mathsf{FV}(M_2)$,
so $M_2[y/y_1, y/y_2] = M_2[y/y_1]$
and then

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = \mathsf{T}(M_2[y/y_1]) = (\Gamma_2[y/y_1], \tau_2). \tag{2}$$

So by rule 3.(c) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma_1'' + \Gamma_2[y/y_1]), \mathbb{S}(\sigma_1))$.

Since $\Gamma_1'' \equiv (\Gamma_1', y : \vec{\tau}_3 \cap \vec{\tau}_4)$, $\Gamma_1 \equiv (\Gamma_1', y_1 : \vec{\tau}_3, y_2 : \vec{\tau}_4)$, $\Gamma_2[y/y_1] \equiv (\Gamma_2', y : \vec{\tau}_3')$, $\Gamma_2 \equiv (\Gamma_2', y_1 : \vec{\tau}_3')$, $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$ and $(\vec{\tau}_3 \cap \vec{\tau}_4) \cap \vec{\tau}_3' = (\vec{\tau}_3 \cap \vec{\tau}_3') \cap \vec{\tau}_4$, we have $\mathbb{S}(\Gamma_1'' + \Gamma_2[y/y_1]) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma_1'' + \Gamma_2[y/y_1])$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\ &= (\Gamma'', \mathbb{S}(\sigma_1)) \\ &= (\Gamma'', \sigma),\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

v. $y_1, y_2 \in \mathsf{dom}(\Gamma_1)$, $y_1 \notin \mathsf{dom}(\Gamma_2)$ and $y_2 \in \mathsf{dom}(\Gamma_2)$:

Analogous to the previous case.

vi. $y_1, y_2 \in \mathsf{dom}(\Gamma_2)$, $y_1 \in \mathsf{dom}(\Gamma_1)$ and $y_2 \notin \mathsf{dom}(\Gamma_1)$:

Analogous to the previous case.

vii. $y_1, y_2 \in \mathsf{dom}(\Gamma_2)$, $y_1 \notin \mathsf{dom}(\Gamma_1)$ and $y_2 \in \mathsf{dom}(\Gamma_1)$:

Analogous to the previous case.

viii. $y_1 \in \mathsf{dom}(\Gamma_1)$, $y_2 \notin \mathsf{dom}(\Gamma_1)$, $y_1 \notin \mathsf{dom}(\Gamma_2)$ and $y_2 \in \mathsf{dom}(\Gamma_2)$:

Since $y_1 \in \mathsf{dom}(\Gamma_1)$, by Lemma 3.3.3, $y_1 \in \mathsf{FV}(M_1)$.

So by Lemma 3.3.6, we have $\mathsf{T}(M_1[y/y_1]) = (\Gamma_1[y/y_1], \tau \multimap \sigma_1)$.

And since $y_2 \notin \mathsf{dom}(\Gamma_1)$, by Lemma 3.3.3, $y_2 \notin \mathsf{FV}(M_1)$,
so $M_1[y/y_1, y/y_2] = M_1[y/y_1]$
and then

$$\mathsf{T}(M_1[y/y_1, y/y_2]) = \mathsf{T}(M_1[y/y_1]) = (\Gamma_1[y/y_1], \tau \multimap \sigma_1). \tag{1}$$

Since $y_2 \in \mathsf{dom}(\Gamma_2)$, by Lemma 3.3.3, $y_2 \in \mathsf{FV}(M_2)$.
So by Lemma 3.3.6, we have $\mathsf{T}(M_2[y/y_2]) = (\Gamma_2[y/y_2], \tau_2)$.

And since $y_1 \notin \mathsf{dom}(\Gamma_2)$, by Lemma 3.3.3, $y_1 \notin \mathsf{FV}(M_2)$,
so $M_2[y/y_1, y/y_2] = M_2[y/y_2]$
and then

$$\mathsf{T}(M_2[y/y_1, y/y_2]) = \mathsf{T}(M_2[y/y_2]) = (\Gamma_2[y/y_2], \tau_2). \tag{2}$$

So by rule 3.(c) of the inference algorithm (and (1), (2)),
$\mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) = (\mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2[y/y_2]), \mathbb{S}(\sigma_1))$.

Since $\mathbb{S}(\Gamma_1 + \Gamma_2) \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$,
we have $\mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2[y/y_2]) \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

Let $\Gamma'' = \mathbb{S}(\Gamma_1[y/y_1] + \Gamma_2[y/y_2])$.

Also, $M[y/y_1, y/y_2] = M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]$, so

$$\begin{aligned}
\mathsf{T}(M[y/y_1, y/y_2]) &= \mathsf{T}(M_1[y/y_1, y/y_2]M_2[y/y_1, y/y_2]) \\
&= (\Gamma'', \mathbb{S}(\sigma_1)) \\
&= (\Gamma'', \sigma),
\end{aligned}$$

with $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

ix. $y_1 \notin \mathsf{dom}(\Gamma_1)$, $y_2 \in \mathsf{dom}(\Gamma_1)$, $y_1 \in \mathsf{dom}(\Gamma_2)$ and $y_2 \notin \mathsf{dom}(\Gamma_2)$:

Analogous to the previous case.

Any other possible case makes the left side of the implication ($\mathsf{T}(M) = (\Gamma, \sigma)$, with $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, and y does not occur in M) false, which makes the statement true. $\square$

Theorem 3.3.8 (Completeness)**.** If $\Phi \triangleright \Gamma \vdash_2 M : \sigma$, then $\mathsf{T}(M) = (\Gamma', \sigma')$ (for some environment Γ' and type σ') and there is a substitution $\mathbb{S}$ such that $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

Proof. By induction on $|\Phi|$.

1. (Axiom): Then $\Gamma = [x : \tau]$, $M = x$ and $\sigma = \tau$.

   $\mathsf{T}(M) = ([x : \alpha], \alpha)$ and let $\mathbb{S} = [\tau/\alpha]$.

   Then $\mathbb{S}(\sigma') = \mathbb{S}(\alpha) = \alpha[\tau/\alpha] = \tau = \sigma$
   and $\mathbb{S}(\Gamma') = \mathbb{S}([x : \alpha]) = [x : \alpha[\tau/\alpha]] = [x : \tau] = \Gamma \equiv \Gamma$.

2. (Exchange): Then $\Gamma = (\Gamma_1, y : \vec{\tau}_2, x : \vec{\tau}_1, \Gamma_2)$, $M = M_1$, $\sigma = \sigma_1$, and assuming that the premise $\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$ holds.

   By the induction hypothesis, $\mathsf{T}(M_1) = (\Gamma'', \sigma'')$
   and there is a substitution $\mathbb{S}'$ such that $\mathbb{S}'(\sigma'') = \sigma_1$ and $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2)$.

   By definition, $(\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2) \equiv (\Gamma_1, y : \vec{\tau}_2, x : \vec{\tau}_1, \Gamma_2)$.
   So $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2) \equiv (\Gamma_1, y : \vec{\tau}_2, x : \vec{\tau}_1, \Gamma_2) = \Gamma$.

   And $\mathsf{T}(M) = \mathsf{T}(M_1)$ and $\mathbb{S}'(\sigma'') = \sigma_1 = \sigma$.

   So for $\Gamma' = \Gamma''$, $\sigma' = \sigma''$ and $\mathbb{S} = \mathbb{S}'$,
   we have $\mathsf{T}(M) = (\Gamma', \sigma')$, $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

3. (Contraction): Then $\Gamma = (\Gamma_1, x : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2)$, $M = M_1[x/x_1, x/x_2]$, $\sigma = \sigma_1$, and assuming that the premise $\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M_1 : \sigma_1$ holds.

   By the induction hypothesis, $\mathsf{T}(M_1) = (\Gamma'', \sigma'')$
   and there is a substitution $\mathbb{S}'$ such that $\mathbb{S}'(\sigma'') = \sigma_1$ and $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2)$.

   Because $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2)$, then $\mathsf{dom}(\Gamma'') = \mathsf{dom}(\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2)$.
   So $\Gamma'' \equiv (\Gamma_3, x_1 : \vec{\tau}_1', x_2 : \vec{\tau}_2')$ for some environment Γ_3 and types $\vec{\tau}_1'$, $\vec{\tau}_2'$ such that $\mathbb{S}'(\vec{\tau}_1') = \vec{\tau}_1$, $\mathbb{S}'(\vec{\tau}_2') = \vec{\tau}_2$ and $\mathbb{S}'(\Gamma_3) \equiv (\Gamma_1, \Gamma_2)$.

Then by Lemma 3.3.7 (x does not occur in M_1),

$\mathsf{T}(M_1[x/x_1, x/x_2]) = (\Gamma_1'', \sigma'')$, with $\Gamma_1'' \equiv (\Gamma_3, x : \vec{\tau}_1' \cap \vec{\tau}_2')$.

And $\mathbb{S}'(\Gamma_3, x : \vec{\tau}_1' \cap \vec{\tau}_2') \equiv (\Gamma_1, \Gamma_2, x : \vec{\tau}_1 \cap \vec{\tau}_2) \equiv (\Gamma_1, x : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2) = \Gamma$.

Also, $M = M_1[x/x_1, x/x_2]$, so $\mathsf{T}(M) = \mathsf{T}(M_1[x/x_1, x/x_2]) = (\Gamma_1'', \sigma'')$.

So for $\Gamma' = \Gamma_1''$, $\sigma' = \sigma''$ and $\mathbb{S} = \mathbb{S}'$, we have $\mathsf{T}(M) = (\Gamma', \sigma')$, $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

4. ($\rightarrow$ Intro): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $\sigma = \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$, and assuming that the premise $\Gamma_1, x : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M_1 : \sigma_1$ (with $n \geq 2$) holds.

   By the induction hypothesis, $\mathsf{T}(M_1) = (\Gamma'', \sigma'')$

   and there is a substitution $\mathbb{S}'$ such that $\mathbb{S}'(\sigma'') = \sigma_1$ and $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \tau_1 \cap \cdots \cap \tau_n)$.

   There is only one possible case for $\mathsf{T}(M)$:

   - $(x : \tau_1' \cap \cdots \cap \tau_m') \in \Gamma''$ (with $m \geq 2$). Then $\mathsf{T}(M) = (\Gamma''_x, \tau_1' \cap \cdots \cap \tau_m' \rightarrow \sigma'')$.

     By $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \tau_1 \cap \cdots \cap \tau_n)$ and the assumption that $(x : \tau_1' \cap \cdots \cap \tau_m') \in \Gamma''$, we have $\mathbb{S}'(\tau_1' \cap \cdots \cap \tau_m') = \tau_1 \cap \cdots \cap \tau_n$.

     Then by that and by $\mathbb{S}'(\sigma'') = \sigma_1$,
     we have $\mathbb{S}'(\tau_1' \cap \cdots \cap \tau_m' \rightarrow \sigma'') = \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$.

     By $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \tau_1 \cap \cdots \cap \tau_n)$ and the definition of environment, $\mathbb{S}'(\Gamma''_x) \equiv \Gamma_1$.

     So for $\Gamma' = \Gamma''_x$, $\sigma' = \tau_1' \cap \cdots \cap \tau_m' \rightarrow \sigma''$ and $\mathbb{S} = \mathbb{S}'$, we have $\mathsf{T}(M) = (\Gamma', \sigma')$, $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

   Note that there is not the case where $x \notin \mathsf{dom}(\Gamma'')$ because by $\Gamma_1, x : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M_1 : \sigma_1$ (with $n \geq 2$) and Corollary 3.3.3.1, $x \in \mathsf{dom}(\Gamma'')$.

   There is also not the case where $(x : \tau) \in \Gamma''$, as the substitution $\mathbb{S}'$ could not exist (because there is no substitution $\mathbb{S}'$ such that $\mathbb{S}'(\tau) = \tau_1 \cap \cdots \cap \tau_n$).

5. ($\rightarrow$ Elim): Then $\Gamma = (\Gamma_0, \sum_{i=1}^{n} \Gamma_i)$, $M = M_1 M_2$, $\sigma = \sigma_1$, and assuming that the premises $\Gamma_0 \vdash_2 M_1 : \tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma_1$ and $\Gamma_i \vdash_2 M_2 : \tau_i$, for $1 \leq i \leq n$ (with $n \geq 2$), hold.

   By the induction hypothesis,

- $\mathsf{T}(M_1) = (\Gamma'_0, \sigma'_0)$ and there is a substitution $\mathbb{S}'_0$ such that $\mathbb{S}'_0(\sigma'_0) = \tau_1 \cap \cdots \cap \tau_n \to \sigma_1$ and $\mathbb{S}'_0(\Gamma'_0) \equiv \Gamma_0$;
- $\mathsf{T}(M_2) = (\Gamma'_i, \sigma'_i)$ and there are substitutions $\mathbb{S}'_i$ such that $\mathbb{S}'_i(\sigma'_i) = \tau_i$ and $\mathbb{S}'_i(\Gamma'_i) \equiv \Gamma_i$, for $1 \leq i \leq n$.

There is only one possible case for $\mathsf{T}(M)$:

- $\sigma'_0 = \tau'_1 \cap \cdots \cap \tau'_n \to \sigma_3$ and, for each $1 \leq i \leq n$, $\sigma'_i = \tau''_i$:

  Let $P = \{\tau'_i = \tau''_i \mid 1 \leq i \leq n\}$.

  Let us assume, without loss of generality, that $\Gamma'_0, \sigma'_0, \mathbb{S}'_0$ and all $\Gamma'_i, \sigma'_i, \mathbb{S}'_i$ do not have type variables in common (if they did, we could simply rename the type variables in each of the $\Gamma'_i, \sigma'_i, \mathbb{S}'_i$ to fresh type variables and we would have the same result, as we consider types equal up to renaming of variables).

  We have $\mathbb{S}'_i(\sigma'_i) = \tau_i$ and $\sigma'_i = \tau''_i$, so $\mathbb{S}'_i(\tau''_i) = \tau_i$, for each $1 \leq i \leq n$.

  And $\mathbb{S}'_0(\sigma'_0) = \tau_1 \cap \cdots \cap \tau_n \to \sigma_1$ and $\sigma'_0 = \tau'_1 \cap \cdots \cap \tau'_n \to \sigma_3$,
  so $\mathbb{S}'_0(\tau'_1 \cap \cdots \cap \tau'_n \to \sigma_3) = \tau_1 \cap \cdots \cap \tau_n \to \sigma_1$.
  Equivalently, $\mathbb{S}'_0(\tau'_1) \cap \cdots \cap \mathbb{S}'_0(\tau'_n) \to \mathbb{S}'_0(\sigma_3) = \tau_1 \cap \cdots \cap \tau_n \to \sigma_1$.
  So $\mathbb{S}'_0(\sigma_3) = \sigma_1$ and $\mathbb{S}'_0(\tau'_i) = \tau_i$, for each $1 \leq i \leq n$.

  Then $\mathbb{S}_3 = \mathbb{S}'_0 \cup \mathbb{S}'_1 \cup \cdots \cup \mathbb{S}'_n$ is a solution to P:
  for all $1 \leq i \leq n$, $\mathbb{S}_3(\tau'_i) = \mathbb{S}'_0(\tau'_i) = \tau_i = \mathbb{S}'_i(\tau''_i) = \mathbb{S}_3(\tau''_i)$.

  Let $\mathbb{S}' = \mathsf{UNIFY}(P)$.
  Then we have $\mathsf{T}(M) = (\mathbb{S}'(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i), \mathbb{S}'(\sigma_3))$, given by the algorithm.

  By Definition 3.3.3 of most general unifier, there exists an $\mathbb{S}$ such that

  $$(\mathbb{S}(\mathbb{S}'(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i)), \mathbb{S}(\mathbb{S}'(\sigma_3))) = (\mathbb{S}_3(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i), \mathbb{S}_3(\sigma_3)). \quad (1)$$

  And $(\mathbb{S}(\mathbb{S}'(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i)), \mathbb{S}(\mathbb{S}'(\sigma_3)))$ is also a solution to $\mathsf{T}(M)$.

  We have $\mathsf{dom}(\Gamma_0) \cap \mathsf{dom}(\Gamma_i) = \emptyset$, for all $1 \leq i \leq n$ (otherwise $\Gamma = \Gamma_0, \sum_{i=1}^{n} \Gamma_i$ would be inconsistent),
  so $\Gamma_0, \sum_{i=1}^{n} \Gamma_i \equiv \Gamma_0 + \sum_{i=1}^{n} \Gamma_i$.

  Because of that and our initial assumption that $\Gamma'_0, \sigma'_0, \mathbb{S}'_0$ and all $\Gamma'_i, \sigma'_i, \mathbb{S}'_i$ do not have type variables in common, we have $\mathbb{S}'_0(\Gamma'_0) + \sum_{i=1}^{n} \mathbb{S}'_i(\Gamma'_i) \equiv \Gamma_0, \sum_{i=1}^{n} \Gamma_i$.

And $\mathbb{S}_3(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i) = \mathbb{S}'_0(\Gamma'_0) + \sum_{i=1}^{n} \mathbb{S}'_i(\Gamma'_i)$,
so $\mathbb{S}_3(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i) \equiv \Gamma_0, \sum_{i=1}^{n} \Gamma_i$,
which, by (1), is equivalent to $\mathbb{S}(\mathbb{S}'(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i)) \equiv \Gamma_0, \sum_{i=1}^{n} \Gamma_i$.

Finally, we have $\mathbb{S}_3(\sigma_3) = \mathbb{S}'_0(\sigma_3)$ and $\mathbb{S}'_0(\sigma_3) = \sigma_1$,
so $\mathbb{S}_3(\sigma_3) = \sigma_1$,
which, by (1), is equivalent to $\mathbb{S}(\mathbb{S}'(\sigma_3)) = \sigma_1$.

So for $\Gamma' = \mathbb{S}'(\Gamma'_0 + \sum_{i=1}^{n} \Gamma'_i)$ and $\sigma' = \mathbb{S}'(\sigma_3)$, we have $\mathsf{T}(M) = (\Gamma', \sigma')$ and there is an $\mathbb{S}$ such that $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

Note that there is not the case where $\sigma'_0 = \tau'_1 \cap \cdots \cap \tau'_m \to \sigma_3$ with $m \neq n$, as the substitution $\mathbb{S}'_0$ could not exist (because there is no substitution $\mathbb{S}'_0$ such that $\mathbb{S}'_0(\tau'_1 \cap \cdots \cap \tau'_m \to \sigma_3) = \tau_1 \cap \cdots \cap \tau_n \to \sigma_1$, for $m \neq n$).

There is also not the case where $\sigma'_0 = \tau' \multimap \sigma_3$ nor the case where $\sigma'_0 = \alpha$ as the substitution $\mathbb{S}'_0$ (such that $\mathbb{S}'_0(\sigma'_0) = \tau_1 \cap \cdots \cap \tau_n \to \sigma_1$) could not exist.

6. ($\multimap$ Intro): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $\sigma = \tau \multimap \sigma_1$, and assuming that the premise $\Gamma_1, x : \tau \vdash_2 M_1 : \sigma_1$ holds.

By the induction hypothesis, $\mathsf{T}(M_1) = (\Gamma'', \sigma'')$

and there is a substitution $\mathbb{S}'$ such that $\mathbb{S}'(\sigma'') = \sigma_1$ and $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \tau)$.

There is only one possible case for $\mathsf{T}(M)$:

- $(x : \tau') \in \Gamma''$. Then $\mathsf{T}(M) = (\Gamma''_x, \tau' \multimap \sigma'')$.

  By $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \tau)$ and the assumption that $(x : \tau') \in \Gamma''$,
  we have $\mathbb{S}'(\tau') = \tau$.

  Then by that and by $\mathbb{S}'(\sigma'') = \sigma_1$,
  we have $\mathbb{S}'(\tau' \multimap \sigma'') = \tau \multimap \sigma_1$.

  By $\mathbb{S}'(\Gamma'') \equiv (\Gamma_1, x : \tau)$ and the definition of environment, $\mathbb{S}'(\Gamma''_x) \equiv \Gamma_1$.

  So for $\Gamma' = \Gamma''_x$, $\sigma' = \tau' \multimap \sigma''$ and $\mathbb{S} = \mathbb{S}'$, we have $\mathsf{T}(M) = (\Gamma', \sigma')$, $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

Note that there is not the case where $x \notin \mathsf{dom}(\Gamma'')$ because by $\Gamma_1, x : \tau \vdash_2 M_1 : \sigma_1$ and Corollary 3.3.3.1, $x \in \mathsf{dom}(\Gamma'')$.

There is also not the case where $(x : \tau_1 \cap \cdots \cap \tau_n) \in \Gamma''$, as the substitution $\mathbb{S}'$ could not exist (because there is no substitution $\mathbb{S}'$ such that $\mathbb{S}'(\tau_1 \cap \cdots \cap \tau_n) = \tau$).

7. ($\multimap$ Elim): Then $\Gamma = (\Gamma_1, \Gamma_2)$, $M = M_1 M_2$, $\sigma = \sigma_1$, and assuming that the premises $\Gamma_1 \vdash_2 M_1 : \tau \multimap \sigma_1$ and $\Gamma_2 \vdash_2 M_2 : \tau$ hold.

   By the induction hypothesis,

   - $\mathsf{T}(M_1) = (\Gamma'_1, \sigma'_1)$ and there is a substitution $\mathbb{S}'_1$ such that $\mathbb{S}'_1(\sigma'_1) = \tau \multimap \sigma_1$ and $\mathbb{S}'_1(\Gamma'_1) \equiv \Gamma_1$;
   - $\mathsf{T}(M_2) = (\Gamma'_2, \sigma'_2)$ and there is a substitution $\mathbb{S}'_2$ such that $\mathbb{S}'_2(\sigma'_2) = \tau$ and $\mathbb{S}'_2(\Gamma'_2) \equiv \Gamma_2$.

   There are two possible cases for $\mathsf{T}(M)$:

   - $\sigma'_1 = \alpha_1$ and $\sigma'_2 = \tau_2$:

     Let $P = \{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\}$, where α_2, α_3 are fresh variables.

     Let us assume, without loss of generality, that $\Gamma'_1, \sigma'_1, \mathbb{S}'_1$ and $\Gamma'_2, \sigma'_2, \mathbb{S}'_2$ do not have type variables in common (if they did, we could simply rename the type variables in $\Gamma'_2, \sigma'_2, \mathbb{S}'_2$ to fresh type variables and we would have the same result, as we consider types equal up to renaming of variables).

     We have $\mathbb{S}'_2(\sigma'_2) = \tau$ and $\sigma'_2 = \tau_2$, so $\mathbb{S}'_2(\tau_2) = \tau$.

     And $\mathbb{S}'_1(\sigma'_1) = \tau \multimap \sigma_1$ and $\sigma'_1 = \alpha_1$, so $\mathbb{S}'_1(\alpha_1) = \tau \multimap \sigma_1$.

     Then $\mathbb{S}_3 = \mathbb{S}'_1 \cup \mathbb{S}'_2 \cup [\tau/\alpha_2, \sigma_1/\alpha_3]$ is a solution to P:
     - $\mathbb{S}_3(\alpha_1) = \mathbb{S}'_1(\alpha_1) = \tau \multimap \sigma_1 = (\alpha_2 \multimap \alpha_3)[\tau/\alpha_2, \sigma_1/\alpha_3] = \mathbb{S}_3(\alpha_2 \multimap \alpha_3)$;
     - $\mathbb{S}_3(\tau_2) = \mathbb{S}'_2(\tau_2) = \tau = \alpha_2[\tau/\alpha_2, \sigma_1/\alpha_3] = \mathbb{S}_3(\alpha_2)$.

     Let $\mathbb{S}' = \mathsf{UNIFY}(P)$.
     Then we have $\mathsf{T}(M) = (\mathbb{S}'(\Gamma'_1 + \Gamma'_2), \mathbb{S}'(\alpha_3))$, given by the algorithm.

     By Definition 3.3.3 of most general unifier, there exists an $\mathbb{S}$ such that

     $$(\mathbb{S}(\mathbb{S}'(\Gamma'_1 + \Gamma'_2)), \mathbb{S}(\mathbb{S}'(\alpha_3))) = (\mathbb{S}_3(\Gamma'_1 + \Gamma'_2), \mathbb{S}_3(\alpha_3)). \tag{1}$$

     And $(\mathbb{S}(\mathbb{S}'(\Gamma'_1 + \Gamma'_2)), \mathbb{S}(\mathbb{S}'(\alpha_3)))$ is also a solution to $\mathsf{T}(M)$.

We have $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$ (otherwise $\Gamma = \Gamma_1, \Gamma_2$ would be inconsistent), so $\Gamma_1, \Gamma_2 \equiv \Gamma_1 + \Gamma_2$.

Because of that and our initial assumption that $\Gamma'_1, \sigma'_1, \mathbb{S}'_1$ and $\Gamma'_2, \sigma'_2, \mathbb{S}'_2$ do not have type variables in common, we have $\mathbb{S}'_1(\Gamma'_1) + \mathbb{S}'_2(\Gamma'_2) \equiv \Gamma_1, \Gamma_2$.

And $\mathbb{S}_3(\Gamma'_1 + \Gamma'_2) = \mathbb{S}'_1(\Gamma'_1) + \mathbb{S}'_2(\Gamma'_2)$,
so $\mathbb{S}_3(\Gamma'_1 + \Gamma'_2) \equiv \Gamma_1, \Gamma_2$,
which, by (1), is equivalent to $\mathbb{S}(\mathbb{S}'(\Gamma'_1 + \Gamma'_2)) \equiv \Gamma_1, \Gamma_2$.

Finally, we have $\mathbb{S}_3(\alpha_3) = \sigma_1$,
which, by (1), is equivalent to $\mathbb{S}(\mathbb{S}'(\alpha_3)) = \sigma_1$.

So for $\Gamma' = \mathbb{S}'(\Gamma'_1 + \Gamma'_2)$ and $\sigma' = \mathbb{S}'(\alpha_3)$, we have $\mathsf{T}(M) = (\Gamma', \sigma')$ and there is an $\mathbb{S}$ such that $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

- $\sigma'_1 = \tau' \multimap \sigma_3$ and $\sigma'_2 = \tau_2$:

  Let $P = \{\tau_2 = \tau'\}$.

  Let us assume, without loss of generality, that $\Gamma'_1, \sigma'_1, \mathbb{S}'_1$ and $\Gamma'_2, \sigma'_2, \mathbb{S}'_2$ do not have type variables in common (if they did, we could simply rename the type variables in $\Gamma'_2, \sigma'_2, \mathbb{S}'_2$ to fresh type variables and we would have the same result, as we consider types equal up to renaming of variables).

  We have $\mathbb{S}'_2(\sigma'_2) = \tau$ and $\sigma'_2 = \tau_2$, so $\mathbb{S}'_2(\tau_2) = \tau$.

  And $\mathbb{S}'_1(\sigma'_1) = \tau \multimap \sigma_1$ and $\sigma'_1 = \tau' \multimap \sigma_3$,
  so $\mathbb{S}'_1(\tau' \multimap \sigma_3) = \tau \multimap \sigma_1$.
  Equivalently, $(\mathbb{S}'_1(\tau')) \multimap (\mathbb{S}'_1(\sigma_3)) = \tau \multimap \sigma_1$.
  So $\mathbb{S}'_1(\tau') = \tau$ and $\mathbb{S}'_1(\sigma_3) = \sigma_1$.

  Then $\mathbb{S}_3 = \mathbb{S}'_1 \cup \mathbb{S}'_2$ is a solution to P:
  $\mathbb{S}_3(\tau_2) = \mathbb{S}'_2(\tau_2) = \tau = \mathbb{S}'_1(\tau') = \mathbb{S}_3(\tau')$.

  Let $\mathbb{S}' = \mathsf{UNIFY}(P)$.
  Then we have $\mathsf{T}(M) = (\mathbb{S}'(\Gamma'_1 + \Gamma'_2), \mathbb{S}'(\sigma_3))$, given by the algorithm.

  By Definition 3.3.3 of most general unifier, there exists an $\mathbb{S}$ such that

  $$(\mathbb{S}(\mathbb{S}'(\Gamma'_1 + \Gamma'_2)), \mathbb{S}(\mathbb{S}'(\sigma_3))) = (\mathbb{S}_3(\Gamma'_1 + \Gamma'_2), \mathbb{S}_3(\sigma_3)). \tag{1}$$

And $(\mathbb{S}(\mathbb{S}'(\Gamma'_1 + \Gamma'_2)), \mathbb{S}(\mathbb{S}'(\sigma_3)))$ is also a solution to $\mathsf{T}(M)$.

We have $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$ (otherwise $\Gamma = \Gamma_1, \Gamma_2$ would be inconsistent),
so $\Gamma_1, \Gamma_2 \equiv \Gamma_1 + \Gamma_2$.

Because of that and our initial assumption that $\Gamma'_1, \sigma'_1, \mathbb{S}'_1$ and $\Gamma'_2, \sigma'_2, \mathbb{S}'_2$ do not have type variables in common, we have $\mathbb{S}'_1(\Gamma'_1) + \mathbb{S}'_2(\Gamma'_2) \equiv \Gamma_1, \Gamma_2$.

And $\mathbb{S}_3(\Gamma'_1 + \Gamma'_2) = \mathbb{S}'_1(\Gamma'_1) + \mathbb{S}'_2(\Gamma'_2)$,
so $\mathbb{S}_3(\Gamma'_1 + \Gamma'_2) \equiv \Gamma_1, \Gamma_2$,
which, by (1), is equivalent to $\mathbb{S}(\mathbb{S}'(\Gamma'_1 + \Gamma'_2)) \equiv \Gamma_1, \Gamma_2$.

Finally, we have $\mathbb{S}_3(\sigma_3) = \mathbb{S}'_1(\sigma_3)$ and $\mathbb{S}'_1(\sigma_3) = \sigma_1$,
so $\mathbb{S}_3(\sigma_3) = \sigma_1$,
which, by (1), is equivalent to $\mathbb{S}(\mathbb{S}'(\sigma_3)) = \sigma_1$.

So for $\Gamma' = \mathbb{S}'(\Gamma'_1 + \Gamma'_2)$ and $\sigma' = \mathbb{S}'(\sigma_3)$, we have $\mathsf{T}(M) = (\Gamma', \sigma')$ and there is an $\mathbb{S}$ such that $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv \Gamma$.

Note that there is not the case where $\sigma'_1 = \tau'_1 \cap \cdots \cap \tau'_n \to \sigma_3$, as the substitution $\mathbb{S}'_1$ could not exist (because there is no substitution $\mathbb{S}'_1$ such that $\mathbb{S}'_1(\tau'_1 \cap \cdots \cap \tau'_n \to \sigma_3) = \tau \multimap \sigma_1$).

□

Hence, we end up with a sound and complete type inference algorithm for the Linear Rank 2 Intersection Type System.

3.4 Final Remarks

A λ-term M is called a λI-term if and only if, for each subterm of the form $\lambda x.N$ in M, x occurs free in N at least once. Note that our type system and type inference algorithm only type λI-terms, but we could have extended them for the *affine terms* – a λ-term M is affine if and only if, for each subterm of the form $\lambda x.N$ in M, x occurs free in N at most once, and if each free variable of M has just one occurrence free in M.

There is no unique and final way of typing affine terms. For instance, in the systems in [1], arguments that do not occur in the body of the function get the empty type [ ]. Since we do not allow the empty sequence in our definition and adding it would make the system more complex, we decided to only work with λI-terms.

Regarding our choice of defining environments as lists and having the rules (Exchange) and (Contraction) in the type system, instead of defining environments as sets and using the $(+)$ operation for concatenation, that decision had to do with the fact that, this way, the system is closer to a linear type system. In the Linear Rank 2 Intersection Type System, a term is linear until we need to contract variables, so using these definitions makes us have more control over linearity and non-linearity. Also, it makes the system more easily extensible for other algebraic properties of intersection. We could also have rewritten the rule $(\rightarrow \text{Elim})$ in order not to use the $(+)$ operation, which is something we might do in the future.

The downside of choosing these definitions is that it makes the proofs (in Chapter 3 and Chapter 4) more complex, as they are not syntax directed because of the rules (Exchange) and (Contraction).

Chapter 4

Resource Inference

Given the quantitative properties of the linear rank 2 intersection types, we now aim to redefine the type system and the type inference algorithm, in order to infer not only the type of a λ-term, but also parameters related to resource usage. In this case, we are interested in obtaining the number of evaluation steps of the λ-term to its normal form, for the leftmost-outermost strategy.

4.1 Type System

The new type system defined in this chapter results from an adaptation and merge between our Linear Rank 2 Intersection Type System (Definition 3.2.2) and the one we presented in Chapter 2 (Definition 2.4.6) from [1], as that system is able to derive a measure related to the number of evaluation steps for the leftmost-outermost strategy. We then begin by recalling and adapting some definitions that were already introduced in Chapter 2 and Chapter 3.

The predicates normal and neutral defining, respectively, the leftmost-outermost normal terms and neutral terms, are recalled in Definition 4.1.1. The predicate $\mathsf{abs}(M)$ is true if and only if M is an abstraction; $\mathsf{normal}(\mathsf{M})$ means that M is in normal form; and $\mathsf{neutral}(M)$ means that M is in normal form and can never behave as an abstraction, i.e., it does not create a redex when applied to an argument.

Definition 4.1.1 (Leftmost-outermost normal forms).

$$\frac{}{\mathsf{neutral}(x)} \qquad \frac{\mathsf{neutral}(M) \qquad \mathsf{normal}(N)}{\mathsf{neutral}(MN)} \qquad \frac{\mathsf{neutral}(M)}{\mathsf{normal}(M)} \qquad \frac{\mathsf{normal}(M)}{\mathsf{normal}(\lambda x.M)}$$

Definition 4.1.2 (Leftmost-outermost evaluation strategy).

$$\frac{}{(\lambda x.M)N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'} \qquad \frac{M \longrightarrow M' \qquad \neg\mathsf{abs}(M)}{MN \longrightarrow M'N}$$

$$\frac{\mathsf{neutral}(N) \qquad M \longrightarrow M'}{NM \longrightarrow NM'}$$

Definition 4.1.3 (Finite rank multi-types)**.** We define the finite rank multi-types by the following grammar:

$$\begin{aligned} \mathsf{tight} &::= \mathsf{Neutral} \mid \mathsf{Abs} && \text{(Tight constants)} \\ t &::= \mathsf{tight} \mid \alpha \mid t \multimap t && \text{(Rank 0 multi-types)} \\ \vec{t} &::= t \mid \vec{t} \cap \vec{t} && \text{(Rank 1 multi-types)} \\ s &::= t \mid \vec{t} \rightarrow s && \text{(Rank 2 multi-types)} \end{aligned}$$

Definition 4.1.4.

- Here, a *statement* is an expression of the form $M : (\vec{\tau}, \vec{t})$, where the pair $(\vec{\tau}, \vec{t})$ 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.
- The comma operator (,) appends a declaration to the end of a list (of declarations). The list (Γ_1, Γ_2) is the list that results from appending the list Γ_2 to the end of the list Γ_1.
- A finite list of declarations is *consistent* if and only if the term variables are all distinct.
- We call *environment* to a consistent finite list of declarations which predicates are pairs with a sequence from $\mathbb{T}_{\mathbb{L}1}$ as the first element and a rank 1 multi-type as the second element of the pair (i.e., the declarations are of the form $x : (\vec{\tau}, \vec{t})$), and we use Γ (possibly with single quotes and/or number subscripts) to range over environments.
- If $\Gamma = [x_1 : (\vec{\tau}_1, \vec{t}_1), \ldots, x_n : (\vec{\tau}_n, \vec{t}_n)]$ is an environment, then Γ is a partial function, with domain $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$, and $\Gamma(x_i) = (\vec{\tau}_i, \vec{t}_i)$.
- We write Γ_x for the resulting environment of eliminating the declaration of x from Γ (if there is no declaration of x in Γ, then $\Gamma_x = \Gamma$).
- We write $\Gamma_1 \equiv \Gamma_2$ if the environments Γ_1 and Γ_2 are equal up to the order of the declarations.
- If Γ_1 and Γ_2 are environments, the environment $\Gamma_1 + \Gamma_2$ is defined as follows:

  for each $x \in \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$,

  $$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \notin \mathsf{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \notin \mathsf{dom}(\Gamma_1) \\ (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2) & \text{if } \Gamma_1(x) = (\vec{\tau}_1, \vec{t}_1) \text{ and } \Gamma_2(x) = (\vec{\tau}_2, \vec{t}_2) \end{cases}$$

  with the declarations of the variables in $\mathsf{dom}(\Gamma_1)$ in the beginning of the list, by the same order they appear in Γ_1, followed by the declarations of the variables in $\mathsf{dom}(\Gamma_2) \setminus \mathsf{dom}(\Gamma_1)$, by the order they appear in Γ_2.
- We write $\mathsf{tight}(s)$ if s is of the form tight and $\mathsf{tight}(t_1 \cap \cdots \cap t_n)$ if $\mathsf{tight}(t_i)$ for all $1 \leq i \leq n$.

  For $\Gamma = [x_1 : (\vec{\tau}_1, \vec{t}_1), \ldots, x_n : (\vec{\tau}_n, \vec{t}_n)]$, we write $\mathsf{tight}(\Gamma)$ if $\mathsf{tight}(\vec{t}_i)$ for all $1 \leq i \leq n$, in which case we also say that Γ is tight.

Definition 4.1.5 (Linear Rank 2 Quantitative Type System)**.** In the Linear Rank 2 Quantitative Type System, we say that M has type σ and multi-type s given the environment Γ, with index b, and write

$$\Gamma \vdash^{b} M : (\sigma, s)$$

if it can be obtained from the following *derivation rules*:

$$[x : (\tau, t)] \vdash^{0} x : (\tau, t) \qquad \text{(Axiom)}$$

$$\frac{\Gamma_1, x : (\vec{\tau}_1, \vec{t}_1), y : (\vec{\tau}_2, \vec{t}_2), \Gamma_2 \vdash^{b} M : (\sigma, s)}{\Gamma_1, y : (\vec{\tau}_2, \vec{t}_2), x : (\vec{\tau}_1, \vec{t}_1), \Gamma_2 \vdash^{b} M : (\sigma, s)} \qquad \text{(Exchange)}$$

$$\frac{\Gamma_1, x_1 : (\vec{\tau}_1, \vec{t}_1), x_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_2 \vdash^{b} M : (\sigma, s)}{\Gamma_1, x : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma_2 \vdash^{b} M[x/x_1, x/x_2] : (\sigma, s)} \qquad \text{(Contraction)}$$

$$\frac{\Gamma, x : (\tau, t) \vdash^{b} M : (\sigma, s)}{\Gamma \vdash^{b+1} \lambda x.M : (\tau \multimap \sigma, t \multimap s)} \qquad (\multimap \text{ Intro})$$

$$\frac{\Gamma, x : (\tau, \mathsf{tight}) \vdash^{b} M : (\sigma, \mathsf{tight})}{\Gamma \vdash^{b} \lambda x.M : (\tau \multimap \sigma, \mathsf{Abs})} \qquad (\multimap \text{ Intro}_{\mathrm{t}})$$

$$\frac{\Gamma, x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \vdash^{b} M : (\sigma, s) \qquad n \geq 2}{\Gamma \vdash^{b+1} \lambda x.M : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, t_1 \cap \cdots \cap t_n \to s)} \qquad (\to \text{ Intro})$$

$$\frac{\Gamma, x : (\tau_1 \cap \cdots \cap \tau_n, \vec{t}) \vdash^{b} M : (\sigma, \mathsf{tight}) \qquad \mathsf{tight}(\vec{t}) \qquad n \geq 2}{\Gamma \vdash^{b} \lambda x.M : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, \mathsf{Abs})} \qquad (\to \text{ Intro}_{\mathrm{t}})$$

$$\frac{\Gamma_1 \vdash^{b_1} M_1 : (\tau \multimap \sigma, t \multimap s) \qquad \Gamma_2 \vdash^{b_2} M_2 : (\tau, t)}{\Gamma_1, \Gamma_2 \vdash^{b_1 + b_2} M_1 M_2 : (\sigma, s)} \qquad (\multimap \text{ Elim})$$

$$\frac{\Gamma_1 \vdash^{b_1} M_1 : (\tau \multimap \sigma, \mathsf{Neutral}) \qquad \Gamma_2 \vdash^{b_2} M_2 : (\tau, \mathsf{tight})}{\Gamma_1, \Gamma_2 \vdash^{b_1 + b_2} M_1 M_2 : (\sigma, \mathsf{Neutral})} \qquad (\multimap \text{ Elim}_{\mathrm{t}})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{b} M_1 : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, t_1 \cap \cdots \cap t_n \to s) \\ \Gamma_1 \vdash^{b_1} M_2 : (\tau_1, t_1) \ \cdots \ \Gamma_n \vdash^{b_n} M_2 : (\tau_n, t_n) \qquad n \geq 2\end{array}}{\Gamma, \sum_{i=1}^{n} \Gamma_i \vdash^{b + b_1 + \cdots + b_n} M_1 M_2 : (\sigma, s)} \qquad (\to \text{ Elim})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{b} M_1 : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, \mathsf{Neutral}) \\ \Gamma_1 \vdash^{b_1} M_2 : (\tau_1, \mathsf{tight}) \ \cdots \ \Gamma_n \vdash^{b_n} M_2 : (\tau_n, \mathsf{tight}) \qquad n \geq 2\end{array}}{\Gamma, \sum_{i=1}^{n} \Gamma_i \vdash^{b + b_1 + \cdots + b_n} M_1 M_2 : (\sigma, \mathsf{Neutral})} \qquad (\to \text{ Elim}_{\mathrm{t}})$$

The tight rules (the t-indexed ones) are used to introduce the tight constants $\mathsf{Neutral}$ and Abs, and they are related to minimal typings. Note that the index is only incremented in rules ($\multimap$ Intro) and ($\rightarrow$ Intro), as these are used to type abstractions that will be applied, contrary to the abstractions typed with the constant Abs.

Notation 4.1.1. We write $\Phi \triangleright \Gamma \vdash^b M : (\sigma, s)$ if Φ is a derivation tree ending with $\Gamma \vdash^b M : (\sigma, s)$. In this case, $|\Phi|$ is the length of the derivation tree Φ.

Definition 4.1.6 (Tight derivations)**.** A derivation $\Phi \triangleright \Gamma \vdash^b M : (\sigma, s)$ is tight if $\mathsf{tight}(s)$ and $\mathsf{tight}(\Gamma)$.

Similarly to what has been done in [1] for the type system we presented in Chapter 2, in this section we prove that, in the Linear Rank 2 Quantitative Type System, whenever a term is tightly typable with index b, then b is exactly the number of evaluations steps to leftmost-outermost normal form.

Example 4.1.1. Let $M = (\lambda x_1.(\lambda x_2.x_2x_1)x_1)I$, where I is the identity function $\lambda y.y$.

Let us first consider the leftmost-outermost evaluation of M to normal form:

$$(\lambda x_1.(\lambda x_2.x_2x_1)x_1)I \longrightarrow (\lambda x_2.x_2I)I \longrightarrow II \longrightarrow I$$

So the evaluation sequence has length 3.

Let us write $\overset{\multimap}{\alpha}$ for the type $(\alpha \multimap \alpha)$ and $\overset{\multimap}{\mathsf{Abs}}$ for the type $\mathsf{Abs} \multimap \mathsf{Abs}$.

To make the derivation tree easier to read, let us first get the following derivation Φ for the term $\lambda x_1.(\lambda x_2.x_2x_1)x_1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{[x_2 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})] \vdash^0 x_2 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}}) \qquad [x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^0 x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}{[x_2 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}}), x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^0 x_2x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}}{[x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^1 \lambda x_2.x_2x_3 : ((\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}) \multimap \overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}} \multimap \mathsf{Abs})} \qquad [x_4 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})] \vdash^0 x_4 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})}{\dfrac{[x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs}), x_4 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})] \vdash^1 (\lambda x_2.x_2x_3)x_4 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}{[x_1 : (\overset{\multimap}{\alpha} \cap (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}), \mathsf{Abs} \cap \overset{\multimap}{\mathsf{Abs}})] \vdash^1 (\lambda x_2.x_2x_1)x_1 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}}}{[\,] \vdash^2 \lambda x_1.(\lambda x_2.x_2x_1)x_1 : ((\overset{\multimap}{\alpha} \cap (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha})) \rightarrow \overset{\multimap}{\alpha}, (\mathsf{Abs} \cap \overset{\multimap}{\mathsf{Abs}}) \rightarrow \mathsf{Abs})}$$

Then for the λ-term M, the following tight derivation is obtained:

$$\dfrac{\Phi \qquad \dfrac{[y : (\alpha, \mathsf{Neutral})] \vdash^0 y : (\alpha, \mathsf{Neutral})}{[\,] \vdash^0 I : (\overset{\multimap}{\alpha}, \mathsf{Abs})} \qquad \dfrac{[y : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^0 y : (\overset{\multimap}{\alpha}, \mathsf{Abs})}{[\,] \vdash^1 I : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})}}{[\,] \vdash^3 (\lambda x_1.(\lambda x_2.x_2x_1)x_1)I : (\overset{\multimap}{\alpha}, \mathsf{Abs})}$$

So indeed, the index 3 represents the number of evaluation steps to leftmost-outermost normal form.

We now show several properties of the type system, adapted from [1], in order to prove the *tight correctness* (Theorem 4.1.7).

Lemma 4.1.1 (Tight spreading on neutral terms)**.** If M is a term such that $\mathsf{neutral}(M)$ and $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ is a typing derivation such that $\mathsf{tight}(\Gamma)$, then $\mathsf{tight}(s)$.

Proof. By induction on $|\Phi|$.

Note that the last rule in Φ cannot be any of the $\multimap$ and $\rightarrow$ intro ones, because $M = \lambda x.M_1$ is not neutral.

1. (Axiom): Then $\Gamma = [x : (\tau, t)]$, $M = x$ and $s = t$.

   Since by hypothesis $\mathsf{tight}(\Gamma)$, then t is tight. So $\mathsf{tight}(s)$.

2. (Exchange): Then $\Gamma = (\Gamma_1, y : (\vec{\tau_2}, \vec{t_2}), x : (\vec{\tau_1}, \vec{t_1}), \Gamma_2)$, $M = M_1$, $s = s_1$, and assuming that the premise $\Gamma_1, x : (\vec{\tau_1}, \vec{t_1}), y : (\vec{\tau_2}, \vec{t_2}), \Gamma_2 \vdash^b M_1 : (\sigma, s_1)$ holds.

   Since by hypothesis $\mathsf{tight}(\Gamma)$, and $(\Gamma_1, x : (\vec{\tau_1}, \vec{t_1}), y : (\vec{\tau_2}, \vec{t_2}), \Gamma_2) \equiv \Gamma$, then $\mathsf{tight}(\Gamma_1, x : (\vec{\tau_1}, \vec{t_1}), y : (\vec{\tau_2}, \vec{t_2}), \Gamma_2)$. And since $\mathsf{neutral}(M)$ and $M = M_1$, then $\mathsf{neutral}(M_1)$.

   So by induction we get $\mathsf{tight}(s_1)$. And because $s = s_1$, we have $\mathsf{tight}(s)$.

3. (Contraction): Then $\Gamma = (\Gamma_1, x : (\vec{\tau_1} \cap \vec{\tau_2}, \vec{t_1} \cap \vec{t_2}), \Gamma_2)$, $M = M_1[x/x_1, x/x_2]$, $s = s_1$, and assuming that the premise $\Gamma_1, x_1 : (\vec{\tau_1}, \vec{t_1}), x_2 : (\vec{\tau_2}, \vec{t_2}), \Gamma_2 \vdash^b M_1 : (\sigma, s_1)$ holds.

   Since by hypothesis $\mathsf{tight}(\Gamma)$, and all types in $(\Gamma_1, x_1 : (\vec{\tau_1}, \vec{t_1}), x_2 : (\vec{\tau_2}, \vec{t_2}), \Gamma_2)$ appear in Γ, then $\mathsf{tight}(\Gamma_1, x_1 : (\vec{\tau_1}, \vec{t_1}), x_2 : (\vec{\tau_2}, \vec{t_2}), \Gamma_2)$. And $\mathsf{neutral}(M_1)$ because $\mathsf{neutral}(M_1[x/x_1, x/x_2])$.

   So by induction we get $\mathsf{tight}(s_1)$. And because $s = s_1$, we have $\mathsf{tight}(s)$.

4. ($\multimap$ Elim): Then $\Gamma = (\Gamma_1, \Gamma_2)$, $M = M_1 M_2$, $s = s_1$, and assuming that the premises $\Gamma_1 \vdash^{b_1} M_1 : (\tau \multimap \sigma, t \multimap s_1)$ and $\Gamma_2 \vdash^{b_2} M_2 : (\tau, t)$ hold.

   Since by hypothesis $\mathsf{neutral}(M_1 M_2)$, then $\mathsf{neutral}(M_1)$ and $\mathsf{normal}(M_2)$

   All types in Γ_1 appear in Γ. Then since by hypothesis $\mathsf{tight}(\Gamma)$, we have $\mathsf{tight}(\Gamma_1)$.

   Then we could apply the induction hypothesis to obtain $\mathsf{tight}(t \multimap s_1)$, which is false.

   So ($\multimap$ Elim) cannot be the last rule in Φ.

5. ($\to$ Elim): Similarly to ($\multimap$ Elim), here we would obtain $\mathsf{tight}(t_1 \cap \cdots \cap t_n \to s_1)$, so ($\to$ Elim) also cannot be the last rule in Φ.

6. ($\multimap$ Elim$_\mathrm{t}$): Then $\Gamma = (\Gamma_1, \Gamma_2)$, $M = M_1 M_2$ and $s = \mathsf{Neutral}$.

   Since $s = \mathsf{Neutral}$, we already have $\mathsf{tight}(s)$.

7. ($\to$ Elim$_\mathrm{t}$): Then $\Gamma = (\Gamma, \sum_{i=1}^{n} \Gamma_i)$, $M = M_1 M_2$ and $s = \mathsf{Neutral}$.

   Since $s = \mathsf{Neutral}$, we already have $\mathsf{tight}(s)$.

□

Lemma 4.1.2 (Properties of tight typings for normal forms)**.** Let M be such that $\mathsf{normal}(M)$ and $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ be a typing derivation.

(i) *Tightness*: if Φ is tight, then $b = 0$.

(ii) *Neutrality*: if $s = \mathsf{Neutral}$ then $\mathsf{neutral}(M)$.

Proof. By induction on $|\Phi|$.

1. (Axiom): Then $\Gamma = [x : (\tau, t)]$, $M = x$, $b = 0$ and $s = t$.

   Clearly, both properties of the statement are verified in this case.

2. (Exchange): Then $\Gamma = (\Gamma_1, y : (\vec{\tau}_2, \vec{t}_2), x : (\vec{\tau}_1, \vec{t}_1), \Gamma_2)$, $M = M_1$, $b = b_1$, $s = s_1$, and assuming $\Phi_1 \rhd \Gamma_1, x : (\vec{\tau}_1, \vec{t}_1), y : (\vec{\tau}_2, \vec{t}_2), \Gamma_2 \vdash^{b_1} M_1 : (\sigma, s_1)$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = M_1$, then $\mathsf{normal}(M_1)$.

   (i) *Tightness*: if Φ is tight, then Φ_1 is tight and by induction, $b = b_1 = 0$.

   (ii) *Neutrality*: if $s = \mathsf{Neutral}$, since $s = s_1$, $s_1 = \mathsf{Neutral}$ and by induction, $\mathsf{neutral}(M_1)$. So since $M = M_1$, $\mathsf{neutral}(M)$.

3. (Contraction): Then $\Gamma = (\Gamma_1, x : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma_2)$, $M = M_1[x/x_1, x/x_2]$, $b = b_1$, $s = s_1$, and assuming $\Phi_1 \rhd \Gamma_1, x_1 : (\vec{\tau}_1, \vec{t}_1), x_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_2 \vdash^{b_1} M_1 : (\sigma, s_1)$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = M_1[x/x_1, x/x_2]$, then $\mathsf{normal}(M_1)$.

(i) *Tightness*: if Φ is tight, then Φ_1 is tight and by induction, $b = b_1 = 0$.

(ii) *Neutrality*: if $s = \mathsf{Neutral}$, since $s = s_1$, $s_1 = \mathsf{Neutral}$ and by induction, $\mathsf{neutral}(M_1)$. So since $M = M_1[x/x_1, x/x_2]$, $\mathsf{neutral}(M)$.

4. ($\multimap$ Intro): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $b = b_1 + 1$, $s = t \multimap s_1$, and assuming $\Phi_1 \rhd \Gamma_1, x : (\tau, t) \vdash^{b_1} M_1 : (\sigma, s_1)$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = \lambda x.M_1$, then $\mathsf{normal}(M_1)$.

   (i) *Tightness*: Φ is not tight, so the statement trivially holds.

   (ii) *Neutrality*: $s \neq \mathsf{Neutral}$, so the statement trivially holds.

5. ($\to$ Intro): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $b = b_1 + 1$, $s = t_1 \cap \cdots \cap t_n \to s_1$, and assuming $\Phi_1 \rhd \Gamma_1, x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \vdash^{b_1} M_1 : (\sigma, s_1)$, with $n \geq 2$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = \lambda x.M_1$, then $\mathsf{normal}(M_1)$.

   (i) *Tightness*: Φ is not tight, so the statement trivially holds.

   (ii) *Neutrality*: $s \neq \mathsf{Neutral}$, so the statement trivially holds.

6. ($\multimap$ $\mathrm{Intro}_{\mathrm{t}}$): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $b = b_1$, $s = \mathsf{Abs}$, and assuming $\Phi_1 \rhd \Gamma_1, x : (\tau, \mathsf{tight}) \vdash^{b_1} M_1 : (\sigma, \mathsf{tight})$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = \lambda x.M_1$, then $\mathsf{normal}(M_1)$.

   (i) *Tightness*: if Φ is tight, then Φ_1 is tight and by induction, $b = b_1 = 0$.

   (ii) *Neutrality*: $s \neq \mathsf{Neutral}$, so the statement trivially holds.

7. ($\to$ $\mathrm{Intro}_{\mathrm{t}}$): Then $\Gamma = \Gamma_1$, $M = \lambda x.M_1$, $b = b_1$, $s = \mathsf{Abs}$, and assuming $\Phi_1 \rhd \Gamma_1, x : (\tau_1 \cap \cdots \cap \tau_n, \vec{t}) \vdash^{b_1} M_1 : (\sigma, \mathsf{tight})$, with $\mathsf{tight}(\vec{t})$ and $n \geq 2$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = \lambda x.M_1$, then $\mathsf{normal}(M_1)$.

   (i) *Tightness*: if Φ is tight, then Φ_1 is tight and by induction, $b = b_1 = 0$.

   (ii) *Neutrality*: $s \neq \mathsf{Neutral}$, so the statement trivially holds.

8. <u>($\multimap$ Elim)</u>: Then $\Gamma = (\Gamma_1, \Gamma_2)$, $M = M_1 M_2$, $b = b_1 + b_2$, $s = s_1$, and assuming $\Phi_1 \rhd \Gamma_1 \vdash^{b_1} M_1 : (\tau \multimap \sigma, t \multimap s_1)$ and $\Phi_2 \rhd \Gamma_2 \vdash^{b_2} M_2 : (\tau, t)$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = M_1 M_2$, then $\mathsf{neutral}(M_1 M_2)$. So $\mathsf{neutral}(M_1)$ (and then $\mathsf{normal}(M_1)$) and $\mathsf{normal}(M_2)$.

   (i) *Tightness*: this case is impossible. If Φ is tight, then $\Gamma = (\Gamma_1, \Gamma_2)$ is tight, and so is Γ_1. And since $\mathsf{neutral}(M_1)$, Lemma 4.1.1 implies that the type of M_1 in Φ_1 has to be tight, which is absurd.

   (ii) *Neutrality*: $\mathsf{neutral}(M)$ holds by hypothesis.

9. <u>($\rightarrow$ Elim)</u>: Then $\Gamma = (\Gamma', \sum_{i=1}^{n} \Gamma_i)$, $M = M_1 M_2$, $b = b' + b_1 + \cdots + b_n$, $s = s_1$, and assuming $\Phi' \rhd \Gamma' \vdash^{b'} M_1 : (\tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma, t_1 \cap \cdots \cap t_n \rightarrow s_1)$ and $\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$, for $1 \leq i \leq n$, with $n \geq 2$.

   Since by hypothesis $\mathsf{normal}(M)$ and $M = M_1 M_2$, then $\mathsf{neutral}(M_1 M_2)$. So $\mathsf{neutral}(M_1)$ (and then $\mathsf{normal}(M_1)$) and $\mathsf{normal}(M_2)$.

   (i) *Tightness*: this case is impossible. If Φ is tight, then $\Gamma = (\Gamma', \sum_{i=1}^{n} \Gamma_i)$ is tight, and so is Γ'. And since $\mathsf{neutral}(M_1)$, Lemma 4.1.1 implies that the type of M_1 in Φ' has to be tight, which is absurd.

   (ii) *Neutrality*: $\mathsf{neutral}(M)$ holds by hypothesis.

10. <u>($\multimap$ Elim$_\mathrm{t}$)</u>: Then $\Gamma = (\Gamma_1, \Gamma_2)$, $M = M_1 M_2$, $b = b_1 + b_2$, $s = \mathsf{Neutral}$, and assuming $\Phi_1 \rhd \Gamma_1 \vdash^{b_1} M_1 : (\tau \multimap \sigma, \mathsf{Neutral})$ and $\Phi_2 \rhd \Gamma_2 \vdash^{b_2} M_2 : (\tau, \mathsf{tight})$.

    Since by hypothesis $\mathsf{normal}(M)$ and $M = M_1 M_2$, then $\mathsf{neutral}(M_1 M_2)$. So $\mathsf{neutral}(M_1)$ (and then $\mathsf{normal}(M_1)$) and $\mathsf{normal}(M_2)$.

    (i) *Tightness*: if Φ is tight, then Φ_1 and Φ_2 are tight and by induction, $b_1 = 0$ and $b_2 = 0$. So $b = b_1 + b_2 = 0$.

    (ii) *Neutrality*: $\mathsf{neutral}(M)$ holds by hypothesis.

11. <u>($\rightarrow$ Elim$_\mathrm{t}$)</u>: Then $\Gamma = (\Gamma', \sum_{i=1}^{n} \Gamma_i)$, $M = M_1 M_2$, $b = b' + b_1 + \cdots + b_n$, $s = \mathsf{Neutral}$, and assuming $\Phi' \rhd \Gamma' \vdash^{b'} M_1 : (\tau_1 \cap \cdots \cap \tau_n \rightarrow \sigma, \mathsf{Neutral})$ and $\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, \mathsf{tight})$, for $1 \leq i \leq n$, with $n \geq 2$.

    Since by hypothesis $\mathsf{normal}(M)$ and $M = M_1 M_2$, then $\mathsf{neutral}(M_1 M_2)$. So $\mathsf{neutral}(M_1)$ (and then $\mathsf{normal}(M_1)$) and $\mathsf{normal}(M_2)$.

(i) *Tightness*: if Φ is tight, then Φ' and Φ_i (for all $1 \leq i \leq n$) are tight and by induction, $b' = 0$ and $b_i = 0$ (for all $1 \leq i \leq n$). So $b = b' + b_1 + \cdots + b_n = 0$.

(ii) *Neutrality*: $\mathsf{neutral}(M)$ holds by hypothesis.

$\square$

Lemma 4.1.3 (Relevance)**.** If $\Phi \triangleright \Gamma \vdash^b M : (\sigma, s)$, then $x \in \mathsf{dom}(\Gamma)$ if and only if $x \in \mathsf{FV}(M)$.

Proof. Easy induction on $|\Phi|$. $\square$

Lemma 4.1.4 (Substitution and typings)**.** Let $\Phi \triangleright \Gamma \vdash^b M_1 : (\sigma, s)$ be a derivation with $x \in \mathsf{dom}(\Gamma)$ and $\Gamma(x) = (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n)$, for $n \geq 1$. And, for each $1 \leq i \leq n$, let $\Phi_i \triangleright \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$.

Then there exists a derivation $\Phi' \triangleright \Gamma_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$. Moreover, if the derivations $\Phi, \Phi_1, \ldots, \Phi_n$ are tight, then so is the derivation Φ'.

Proof. The proof follows by induction on $|\Phi|$.

Without loss of generality, we assume that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$, so that $\Gamma_x, \sum_{i=1}^{n} \Gamma_i$ is consistent. Otherwise, we could simply rename the free variables in M_1 to get M_1' (and the same derivation Φ, with the variables renamed) such that $\mathsf{FV}(M_1') \cap \mathsf{FV}(M_2) = \emptyset$. Then, by a weaker form of the lemma (for M_1, M_2 such that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$), we would get the derivation Φ' (with the renamed variables) and finally we could apply the rule (Contraction) (and (Exchange), when necessary) to the variables that were renamed in M_1, in order to end up with the proper derivation Φ'.

1. <u>(Axiom)</u>:

   Then we have
   $$\Phi \triangleright [x : (\tau_1, t_1)] \vdash^0 x : (\tau_1, t_1).$$
   So $\Gamma = [x : (\tau_1, t_1)]$, $\Gamma_x = [\,]$, $M_1 = x$, $b = 0$, $\sigma = \tau_1$ and $s = t_1$.

   By hypothesis we also have:
   $$\Phi_1 \triangleright \Gamma_1 \vdash^{b_1} M_2 : (\tau_1, t_1)$$
   Given that $(\Gamma_x, \Gamma_1) = ([\,], \Gamma_1) = \Gamma_1$, $M_1[M_2/x] = x[M_2/x] = M_2$, $b + b_1 = 0 + b_1 = b_1$, $\sigma = \tau_1$ and $s = t_1$, then we already have the derivation $\Phi' = \Phi_1$, as we wanted.

2. (Exchange):

   Then we have
   $$\Phi \rhd \Gamma'_1, y_2 : (\vec{\tau}_2, \vec{t}_2), y_1 : (\vec{\tau}_1, \vec{t}_1), \Gamma'_2 \vdash^b M_1 : (\sigma, s),$$
   which follows from $\Phi'_1 \rhd \Gamma'_1, y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma'_2 \vdash^b M_1 : (\sigma, s)$.

   And there are three different cases depending on x:

   (a) $x \neq y_1$ and $x \neq y_2$:

   So $\Gamma = (\Gamma'_1, y_2 : (\vec{\tau}_2, \vec{t}_2), y_1 : (\vec{\tau}_1, \vec{t}_1), \Gamma'_2)$ and $\Gamma_x = (\Gamma'_{1x}, y_2 : (\vec{\tau}_2, \vec{t}_2), y_1 : (\vec{\tau}_1, \vec{t}_1), \Gamma'_{2x})$.

   Since $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma$, then either that declaration is in Γ'_1 or in Γ'_2, and $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in (\Gamma'_1, y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma'_2)$.

   By hypothesis we also have:
   $$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$
   for $1 \leq i \leq n$.

   Given Φ'_1 and Φ_i, by the induction hypothesis, there is a derivation ending with
   $$\Gamma'_{1x}, y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s).$$
   By rule (Exchange), we get the final judgment we wanted:
   $$\frac{\Gamma'_{1x}, y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)}{\Gamma'_{1x}, y_2 : (\vec{\tau}_2, \vec{t}_2), y_1 : (\vec{\tau}_1, \vec{t}_1), \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)}$$
   So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

   (b) $x = y_1$:

   So $\Gamma = (\Gamma'_1, y_2 : (\vec{\tau}_2, \vec{t}_2), y_1 : (\vec{\tau}_1, \vec{t}_1), \Gamma'_2) = (\Gamma'_1, y_2 : (\vec{\tau}_2, \vec{t}_2), x : (\vec{\tau}_1, \vec{t}_1), \Gamma'_2)$ and $\Gamma_x = (\Gamma'_1, y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma'_2)$.

   By hypothesis we also have:
   $$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$
   for $1 \leq i \leq n$.

Given $\Phi_1' \rhd \Gamma_1', x : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_2' \vdash^b M_1 : (\sigma, s)$ and Φ_i, by the induction hypothesis, there is a derivation

$$\Phi' \rhd \Gamma_1', y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_2', \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s),$$

which is the derivation we wanted.

(c) $x = y_2$:

Analogous to the case where $x = y_1$.

3. <u>(Contraction)</u>:

Then we have

$$\Phi \rhd \Gamma_1', y : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma_2' \vdash^b M_1'[y/y_1, y/y_2] : (\sigma, s),$$

which follows from $\Phi_1' \rhd \Gamma_1', y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_2' \vdash^b M_1' : (\sigma, s)$.

And there are two different cases depending on x:

(a) $x \neq y$:

So $\Gamma = (\Gamma_1', y : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma_2')$, $\Gamma_x = (\Gamma_{1x}', y : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma_{2x}')$, $M_1 = M_1'[y/y_1, y/y_2]$, $x \neq y_1$ and $x \neq y_2$.

Since $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma$, then either that declaration is in Γ_1' or in Γ_2', and $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in (\Gamma_1', y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_2')$.

By hypothesis we also have:

$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$

for $1 \leq i \leq n$.

Given Φ_1' and Φ_i, by the induction hypothesis, there is a derivation ending with

$$\Gamma_{1x}', y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_{2x}', \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1'[M_2/x] : (\sigma, s).$$

Note that this implies that y_1 and y_2 do not occur free in M_2, otherwise, by Lemma 4.1.3, $y_1, y_2 \in \mathsf{dom}(\Gamma_i)$ and so $\Gamma_{1x}', y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_{2x}', \sum_{i=1}^{n} \Gamma_i$ would not be consistent.

By rule (Contraction), we get the final judgment we wanted:

$$\frac{\Gamma'_{1x}, y_1 : (\vec{\tau}_1, \vec{t}_1), y_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M'_1[M_2/x] : (\sigma, s)}{\Gamma'_{1x}, y : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} (M'_1[M_2/x])[y/y_1, y/y_2] : (\sigma, s)}$$

Since $x \neq y_1$, $x \neq y_2$, $x \neq y$ and y_1, y_2 do not occur free in M_2, then $(M'_1[M_2/x])[y/y_1, y/y_2] = (M'_1[y/y_1, y/y_2])[M_2/x]$.
And as $M_1 = M'_1[y/y_1, y/y_2]$, we have $(M'_1[y/y_1, y/y_2])[M_2/x] = M_1[M_2/x]$.

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

(b) $x = y$:

So $\Gamma = (\Gamma'_1, y : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma'_2) = (\Gamma'_1, x : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma'_2)$, $\Gamma_x = (\Gamma'_1, \Gamma'_2)$, $M_1 = M'_1[y/y_1, y/y_2] = M'_1[x/y_1, x/y_2]$ and $x \neq y_1$ and $x \neq y_2$ (assuming that $y \neq y_1$ and $y \neq y_2$, without loss of generality). Let us also assume, without loss of generality, that y_1, y_2 do not occur in M_2.

By hypothesis we have $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma$ and $y : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2) \in \Gamma$.
So since $x = y$, we have $\tau_1 \cap \cdots \cap \tau_n = \vec{\tau}_1 \cap \vec{\tau}_2$ and $t_1 \cap \cdots \cap t_n = \vec{t}_1 \cap \vec{t}_2$.
So for some $1 \leq k < n$, we have $\vec{\tau}_1 = \tau_1 \cap \cdots \cap \tau_k$, $\vec{\tau}_2 = \tau_{k+1} \cap \cdots \cap \tau_n$, $\vec{t}_1 = t_1 \cap \cdots \cap t_k$ and $\vec{t}_2 = t_{k+1} \cap \cdots \cap t_n$.

By hypothesis we also have:

$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$

for $1 \leq i \leq n$.

Given $\Phi'_1 \rhd \Gamma'_1, y_1 : (\tau_1 \cap \cdots \cap \tau_k, t_1 \cap \cdots \cap t_k), y_2 : (\tau_{k+1} \cap \cdots \cap \tau_n, t_{k+1} \cap \cdots \cap t_n), \Gamma'_2 \vdash^{b} M'_1 : (\sigma, s)$ and Φ_j for $1 \leq j \leq k$, by the induction hypothesis, there is a derivation ending with

$$\Gamma'_1, y_2 : (\tau_{k+1} \cap \cdots \cap \tau_n, t_{k+1} \cap \cdots \cap t_n), \Gamma'_2, \sum_{j=1}^{k} \Gamma_j \vdash^{b+b_1+\cdots+b_k} M'_1[M_2/y_1] : (\sigma, s).$$

Now given that derivation and Φ_j for $k+1 \leq j \leq n$, by the induction hypothesis, there is a derivation

$$\Phi' \rhd \Gamma'_1, \Gamma'_2, \sum_{j=k+1}^{n} \Gamma_j \vdash^{b+b_1+\cdots+b_k+b_{k+1}+\cdots+b_n} (M'_1[M_2/y_1])[M_2/y_2] : (\sigma, s),$$

which is the derivation we wanted.

Since $x \neq y_1$, $x \neq y_2$, $y_1 \neq y_2$ and y_1, y_2, x do not occur in M_2 and x does not occur free in M_1', then:

$$\begin{aligned}(M_1'[M_2/y_1])[M_2/y_2] &= ((M_1'[x/y_1])[M_2/x])[M_2/y_2] \\ &= (((M_1'[x/y_1])[M_2/x])[x/y_2])[M_2/x] \\ &= ((M_1'[x/y_1])[x/y_2])[M_2/x] \\ &= (M_1'[x/y_1, x/y_2])[M_2/x]\end{aligned}$$

And as $M_1 = M_1'[x/y_1, x/y_2]$, we have $(M_1'[M_2/y_1])[M_2/y_2] = M_1[M_2/x]$.

Also $(\Gamma_1', \Gamma_2') = \Gamma_x$ and $b + b_1 + \cdots + b_k + b_{k+1} + \cdots + b_n = b + b_1 + \cdots + b_n$.

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^n \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

4. <u>($\multimap$ Intro)</u>:

Then we have

$$\Phi \rhd \Gamma \vdash^{b'+1} \lambda y.M : (\tau \multimap \sigma', t \multimap s'),$$

which follows from $\Phi_1' \rhd \Gamma, y : (\tau, t) \vdash^{b'} M : (\sigma', s')$.
So $M_1 = \lambda y.M$, $b = b' + 1$, $\sigma = \tau \multimap \sigma'$, $s = t \multimap s'$ and $x \neq y$.

Since $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma$, then $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in (\Gamma, y : (\tau, t))$.

By hypothesis we also have:

$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$

for $1 \leq i \leq n$.

Given Φ_1' and Φ_i, by the induction hypothesis, there is a derivation ending with

$$\Gamma_x, y : (\tau, t), \sum_{i=1}^{n} \Gamma_i \vdash^{b'+b_1+\cdots+b_n} M[M_2/x] : (\sigma', s').$$

We can now perform consecutive applications of (Exchange) in order to get

$$\Gamma_x, \sum_{i=1}^{n} \Gamma_i, y : (\tau, t) \vdash^{b'+b_1+\cdots+b_n} M[M_2/x] : (\sigma', s').$$

Finally, by rule ($\multimap$ Intro), we get the final judgment we wanted:

$$\frac{\Gamma_x, \sum_{i=1}^n \Gamma_i, y : (\tau, t) \vdash^{b'+b_1+\cdots+b_n} M[M_2/x] : (\sigma', s')}{\Gamma_x, \sum_{i=1}^n \Gamma_i \vdash^{b'+b_1+\cdots+b_n+1} \lambda y.(M[M_2/x]) : (\tau \multimap \sigma', t \multimap s')}$$

Since $M_1 = \lambda y.M$ and $x \neq y$, then $\lambda y.(M[M_2/x]) = (\lambda y.M)[M_2/x] = M_1[M_2/x]$.
Also $b = b' + 1$, so $b' + b_1 + \cdots + b_n + 1 = b + b_1 + \cdots + b_n$.

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^n \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

5. ($\to$ Intro):

   Similar to the previous case.

6. ($\multimap$ Intro$_\mathrm{t}$):

   Then we have

$$\Phi \rhd \Gamma \vdash^{b'} \lambda y.M : (\tau \multimap \sigma', \mathsf{Abs}),$$

which follows from $\Phi_1' \rhd \Gamma, y : (\tau, \mathsf{tight}) \vdash^{b'} M : (\sigma', \mathsf{tight})$.
So $M_1 = \lambda y.M$, $b = b'$, $\sigma = \tau \multimap \sigma'$, $s = \mathsf{Abs}$ and $x \neq y$.

Since $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma$, then $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in (\Gamma, y : (\tau, \mathsf{tight}))$.

By hypothesis we also have:

$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$

for $1 \leq i \leq n$.

Given Φ_1' and Φ_i, by the induction hypothesis, there is a derivation ending with

$$\Gamma_x, y : (\tau, \mathsf{tight}), \sum_{i=1}^n \Gamma_i \vdash^{b'+b_1+\cdots+b_n} M[M_2/x] : (\sigma', \mathsf{tight}).$$

We can now perform consecutive applications of (Exchange) in order to get

$$\Gamma_x, \sum_{i=1}^n \Gamma_i, y : (\tau, \mathsf{tight}) \vdash^{b'+b_1+\cdots+b_n} M[M_2/x] : (\sigma', \mathsf{tight}).$$

Finally, by rule ($\multimap$ Intro$_\mathrm{t}$), we get the final judgment we wanted:

$$\frac{\Gamma_x, \sum_{i=1}^n \Gamma_i, y : (\tau, \mathsf{tight}) \vdash^{b'+b_1+\cdots+b_n} M[M_2/x] : (\sigma', \mathsf{tight})}{\Gamma_x, \sum_{i=1}^n \Gamma_i \vdash^{b'+b_1+\cdots+b_n} \lambda y.(M[M_2/x]) : (\tau \multimap \sigma', \mathsf{Abs})}$$

Since $M_1 = \lambda y.M$ and $x \neq y$, then $\lambda y.(M[M_2/x]) = (\lambda y.M)[M_2/x] = M_1[M_2/x]$.
Also $b = b'$, so $b' + b_1 + \cdots + b_n = b + b_1 + \cdots + b_n$.

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^n \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

7. $(\to \text{Intro}_t)$:

   Similar to the previous case.

8. $(\multimap \text{Elim})$:

   Then we have
$$\Phi \rhd \Gamma'_1, \Gamma'_2 \vdash^{b'_1+b'_2} N_1N_2 : (\sigma, s),$$
   which follows from $\Phi'_1 \rhd \Gamma'_1 \vdash^{b'_1} N_1 : (\tau \multimap \sigma, t \multimap s)$ and $\Phi'_2 \rhd \Gamma'_2 \vdash^{b'_2} N_2 : (\tau, t)$.

   Since $x \in \mathsf{dom}(\Gamma)$ and $\Gamma = (\Gamma'_1, \Gamma'_2)$, either $x \in \mathsf{dom}(\Gamma'_1)$ (and $x \in \mathsf{FV}(N_1)$, by Lemma 4.1.3) or $x \in \mathsf{dom}(\Gamma'_2)$ (and $x \in \mathsf{FV}(N_2)$, by Lemma 4.1.3). Note that there is not the case where $x \in \mathsf{dom}(\Gamma'_1)$ and $x \in \mathsf{dom}(\Gamma'_2)$ simultaneously, otherwise (Γ'_1, Γ'_2) would be inconsistent.

   So there are two different cases depending on x:

   (a) $x \in \mathsf{dom}(\Gamma'_1)$ and $x \notin \mathsf{dom}(\Gamma'_2)$:

   So $\Gamma = (\Gamma'_1, \Gamma'_2)$, $\Gamma_x = (\Gamma'_{1x}, \Gamma'_2)$, $M_1 = N_1N_2$ and $b = b'_1 + b'_2$.

   And $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma'_1$.

   By hypothesis we also have:
$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$
   for $1 \leq i \leq n$.

   Given Φ'_1 and Φ_i, by the induction hypothesis, there is a derivation ending with
$$\Gamma'_{1x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b'_1+b_1+\cdots+b_n} N_1[M_2/x] : (\tau \multimap \sigma, t \multimap s).$$
   By rule $(\multimap \text{Elim})$, with $\Gamma'_2 \vdash^{b'_2} N_2 : (\tau, t)$ from Φ'_2, we get:
$$\frac{\Gamma'_{1x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b'_1+b_1+\cdots+b_n} N_1[M_2/x] : (\tau \multimap \sigma, t \multimap s) \qquad \Gamma'_2 \vdash^{b'_2} N_2 : (\tau, t)}{\Gamma'_{1x}, \sum_{i=1}^{n} \Gamma_i, \Gamma'_2 \vdash^{b'_1+b_1+\cdots+b_n+b'_2} (N_1[M_2/x])N_2 : (\sigma, s)}$$
   Note that $(\Gamma'_{1x}, \sum_{i=1}^{n} \Gamma_i, \Gamma'_2)$ is consistent because of our initial assumption that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$.

   We can now perform consecutive applications of (Exchange) in order to get the final judgment we wanted:
$$\Gamma'_{1x}, \Gamma'_2, \sum_{i=1}^{n} \Gamma_i \vdash^{b'_1+b_1+\cdots+b_n+b'_2} (N_1[M_2/x])N_2 : (\sigma, s)$$

Since $M_1 = N_1N_2$ and $x \notin \mathsf{FV}(N_2)$ (by Lemma 4.1.3, since $x \notin \mathsf{dom}(\Gamma'_2)$), then $(N_1[M_2/x])N_2 = (N_1[M_2/x])(N_2[M_2/x]) = (N_1N_2)[M_2/x] = M_1[M_2/x]$.
Also $b = b'_1 + b'_2$, so $b'_1 + b_1 + \cdots + b_n + b'_2 = b + b_1 + \cdots + b_n$.

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

(b) $x \in \mathsf{dom}(\Gamma'_2)$ and $x \notin \mathsf{dom}(\Gamma'_1)$:

So $\Gamma = (\Gamma'_1, \Gamma'_2)$, $\Gamma_x = (\Gamma'_1, \Gamma'_{2x})$, $M_1 = N_1N_2$ and $b = b'_1 + b'_2$.

And $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma'_2$.

By hypothesis we also have:

$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$

for $1 \leq i \leq n$.

Given Φ'_2 and Φ_i, by the induction hypothesis, there is a derivation ending with

$$\Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b'_2+b_1+\cdots+b_n} N_2[M_2/x] : (\tau, t).$$

By rule ($\multimap$ Elim), with $\Gamma'_1 \vdash^{b'_1} N_1 : (\tau \multimap \sigma, t \multimap s)$ from Φ'_1, we get the final judgment we wanted:

$$\frac{\Gamma'_1 \vdash^{b'_1} N_1 : (\tau \multimap \sigma, t \multimap s) \qquad \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b'_2+b_1+\cdots+b_n} N_2[M_2/x] : (\tau, t)}{\Gamma'_1, \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i \vdash^{b'_1+b'_2+b_1+\cdots+b_n} N_1(N_2[M_2/x]) : (\sigma, s)}$$

Note that $(\Gamma'_1, \Gamma'_{2x}, \sum_{i=1}^{n} \Gamma_i)$ is consistent because of our initial assumption that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$.

Since $M_1 = N_1N_2$ and $x \notin \mathsf{FV}(N_1)$ (by Lemma 4.1.3, since $x \notin \mathsf{dom}(\Gamma'_1)$), then $N_1(N_2[M_2/x]) = (N_1[M_2/x])(N_2[M_2/x]) = (N_1N_2)[M_2/x] = M_1[M_2/x]$.
Also $b = b'_1 + b'_2$, so $b'_1 + b'_2 + b_1 + \cdots + b_n = b + b_1 + \cdots + b_n$.

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

9. ($\multimap$ $\mathrm{Elim_t}$):

Similar to the previous case.

10. (→ Elim):

Then we have
$$\Phi \rhd \Gamma', \sum_{j=1}^{m} \Gamma'_j \vdash^{b'+b'_1+\cdots+b'_m} N_1 N_2 : (\sigma, s)$$
which follows from $\Phi''_1 \rhd \Gamma' \vdash^{b'} N_1 : (\tau'_1 \cap \cdots \cap \tau'_m \to \sigma, t'_1 \cap \cdots \cap t'_m \to s)$,
$\Phi'_1 \rhd \Gamma'_1 \vdash^{b'_1} N_2 : (\tau'_1, t'_1), \ldots, \Phi'_m \rhd \Gamma'_m \vdash^{b'_m} N_2 : (\tau'_m, t'_m)$ and $m \geq 2$.
Since $x \in \mathsf{dom}(\Gamma)$ and $\Gamma = (\Gamma', \sum_{j=1}^{m} \Gamma'_j)$, either $x \in \mathsf{dom}(\Gamma')$ (and $x \in \mathsf{FV}(N_1)$, by Lemma 4.1.3) or $x \in \mathsf{dom}(\Gamma'_j)$, for $1 \leq j \leq m$ (and $x \in \mathsf{FV}(N_2)$, by Lemma 4.1.3). Note that there is not the case where $x \in \mathsf{dom}(\Gamma')$ and $x \in \mathsf{dom}(\Gamma'_j)$ (for $1 \leq j \leq m$) simultaneously, otherwise $(\Gamma', \sum_{j=1}^{m} \Gamma'_j)$ would be inconsistent.

So there are two different cases depending on x:

(a) $x \in \mathsf{dom}(\Gamma')$ and $x \notin \mathsf{dom}(\Gamma'_j)$, for $1 \leq j \leq m$:

So $\Gamma = (\Gamma', \sum_{j=1}^{m} \Gamma'_j)$, $\Gamma_x = ({\Gamma'}_x, \sum_{j=1}^{m} \Gamma'_j)$, $M_1 = N_1 N_2$ and $b = b' + b'_1 + \cdots + b'_m$.

And $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \Gamma'$.

By hypothesis we also have:
$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$
for $1 \leq i \leq n$.

Given Φ''_1 and Φ_i, by the induction hypothesis, there is a derivation ending with
$${\Gamma'}_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b'+b_1+\cdots+b_n} N_1[M_2/x] : (\tau'_1 \cap \cdots \cap \tau'_m \to \sigma, t'_1 \cap \cdots \cap t'_m \to s).$$
By rule (→ Elim), with $\Gamma'_1 \vdash^{b'_1} N_2 : (\tau'_1, t'_1), \ldots, \Gamma'_m \vdash^{b'_m} N_2 : (\tau'_m, t'_m)$ from $\Phi'_1, \ldots, \Phi'_m$, respectively, we get:
$$\frac{\begin{array}{c}{\Gamma'}_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b'+b_1+\cdots+b_n} N_1[M_2/x] : (\tau'_1 \cap \cdots \cap \tau'_m \to \sigma, t'_1 \cap \cdots \cap t'_m \to s) \\ \Gamma'_1 \vdash^{b'_1} N_2 : (\tau'_1, t'_1) \quad \cdots \quad \Gamma'_m \vdash^{b'_m} N_2 : (\tau'_m, t'_m)\end{array}}{{\Gamma'}_x, \sum_{i=1}^{n} \Gamma_i, \sum_{j=1}^{m} \Gamma'_j \vdash^{b'+b_1+\cdots+b_n+b'_1+\cdots+b'_m} (N_1[M_2/x])N_2 : (\sigma, s)}$$

Note that $({\Gamma'}_x, \sum_{i=1}^{n} \Gamma_i, \sum_{j=1}^{m} \Gamma'_j)$ is consistent because of our initial assumption that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$.

We can now perform consecutive applications of (Exchange) in order to get the final judgment we wanted:
$${\Gamma'}_x, \sum_{j=1}^{m} \Gamma'_j, \sum_{i=1}^{n} \Gamma_i \vdash^{b'+b_1+\cdots+b_n+b'_1+\cdots+b'_m} (N_1[M_2/x])N_2 : (\sigma, s)$$

Since $M_1 = N_1 N_2$ and $x \notin \mathsf{FV}(N_2)$ (by Lemma 4.1.3, since $x \notin \mathsf{dom}(\Gamma'_j)$ for $1 \leq j \leq m$), then $(N_1[M_2/x])N_2 = (N_1[M_2/x])(N_2[M_2/x]) = (N_1 N_2)[M_2/x] = M_1[M_2/x]$.
Also $b = b' + b'_1 + \cdots + b'_m$, so $b' + b_1 + \cdots + b_n + b'_1 + \cdots + b'_m = b + b_1 + \cdots + b_n$.

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

(b) $x \in \mathsf{dom}(\Gamma'_j)$, for $1 \leq j \leq m$, and $x \notin \mathsf{dom}(\Gamma')$:

So $\Gamma = (\Gamma', \sum_{j=1}^{m} \Gamma'_j)$, $\Gamma_x = (\Gamma', \sum_{j=1}^{m} {\Gamma'_j}_x)$, $M_1 = N_1 N_2$ and $b = b' + b'_1 + \cdots + b'_m$.

And $x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \in \sum_{j=1}^{m} \Gamma'_j$, which means that the sequences $(\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n)$ can be split between the environments $\Gamma'_1, \ldots, \Gamma'_m$. Also, note that this implies $n \geq m$ (because by Lemma 4.1.3, $x \in \mathsf{dom}(\Gamma'_j)$ for all $1 \leq j \leq m$).

Then for $1 = k_1 < \cdots < k_m < k_{m+1} = n + 1$ and $1 \leq j \leq m$,
let $x : (\tau_{k_j} \cap \cdots \cap \tau_{k_{(j+1)}-1}, t_{k_j} \cap \cdots \cap t_{k_{(j+1)}-1}) \in \Gamma'_j$.

By hypothesis we also have:

$$\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$$

for $1 \leq i \leq n$.

So for each $1 \leq j \leq m$, given Φ'_j and $\Phi_{k_j}, \ldots, \Phi_{k_{(j+1)}-1}$, by the induction hypothesis, there is a derivation ending with

$${\Gamma'_j}_x, \sum_{i=k_j}^{k_{(j+1)}-1} \Gamma_i \vdash^{b'_j + b_{k_j} + \cdots + b_{k_{(j+1)}-1}} N_2[M_2/x] : (\tau'_j, t'_j).$$

By rule ($\rightarrow$ Elim), with $\Gamma' \vdash^{b'} N_1 : (\tau'_1 \cap \cdots \cap \tau'_m \rightarrow \sigma, t'_1 \cap \cdots \cap t'_m \rightarrow s)$ from Φ''_1, we get the final judgment we wanted:

$$\frac{\Gamma' \vdash^{b'} N_1 : (\tau'_1 \cap \cdots \cap \tau'_m \rightarrow \sigma, t'_1 \cap \cdots \cap t'_m \rightarrow s) \quad {\Gamma'_1}_x, \sum_{i=1}^{k_2-1} \Gamma_i \vdash^{b'_1+b_1+\cdots+b_{k_2-1}} N_2[M_2/x] : (\tau'_1, t'_1) \ \cdots \ {\Gamma'_m}_x, \sum_{i=k_m}^{n} \Gamma_i \vdash^{b'_m+b_{k_m}+\cdots+b_n} N_2[M_2/x] : (\tau'_m, t'_m)}{\Gamma', (({\Gamma'_1}_x, \sum_{i=1}^{k_2-1} \Gamma_i) + \cdots + ({\Gamma'_m}_x, \sum_{i=k_m}^{n} \Gamma_i)) \vdash^{b''} N_1(N_2[M_2/x]) : (\sigma, s)}$$

where $b'' = b' + (b'_1 + b_1 + \cdots + b_{k_2-1}) + \cdots + (b'_m + b_{k_m} + \cdots + b_n)$.

By our initial assumption that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$, by Lemma 4.1.3, and by looking at the definition of (+), we have:

$$
\begin{aligned}
&\Gamma', ((\Gamma'_{1x}, \sum_{i=1}^{k_2-1} \Gamma_i) + \cdots + (\Gamma'_{mx}, \sum_{i=k_m}^{n} \Gamma_i)) \\
&= \Gamma', ((\Gamma'_{1x} + \cdots + \Gamma'_{mx}), (\sum_{i=1}^{k_2-1} \Gamma_i + \cdots + \sum_{i=k_m}^{n} \Gamma_i)) \\
&= \Gamma', (\sum_{j=1}^{m} \Gamma'_{jx}, \sum_{i=1}^{n} \Gamma_i) \\
&= (\Gamma', \sum_{j=1}^{m} \Gamma'_{jx}), \sum_{i=1}^{n} \Gamma_i \\
&= \Gamma_x, \sum_{i=1}^{n} \Gamma_i.
\end{aligned}
$$

Since $M_1 = N_1 N_2$ and $x \notin \mathsf{FV}(N_1)$ (by Lemma 4.1.3, since $x \notin \mathsf{dom}(\Gamma')$), then $N_1(N_2[M_2/x]) = (N_1[M_2/x])(N_2[M_2/x]) = (N_1 N_2)[M_2/x] = M_1[M_2/x]$.

Also since $b = b' + b'_1 + \cdots + b'_m$, we have:

$$
\begin{aligned}
b'' &= b' + (b'_1 + b_1 + \cdots + b_{k_2-1}) + \cdots + (b'_m + b_{k_m} + \cdots + b_n) \\
&= b' + (b'_1 + \cdots + b'_m) + (b_1 + \cdots + b_{k_2-1} + \cdots + b_{k_m} + \cdots + b_n) \\
&= b' + (b'_1 + \cdots + b'_m) + (b_1 + \cdots + b_n) \\
&= b + b_1 + \cdots + b_n.
\end{aligned}
$$

So there is indeed $\Phi' \rhd \Gamma_x, \sum_{i=1}^{n} \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$.

11. $\underline{(\to \mathrm{Elim_t})}$:

    Similar to the previous case.

□

We now show an important property that relates contracted terms with their linear counterpart. Basically, it says that the following diagram commutes (under the described conditions):

$$
\begin{tikzcd}
M' \arrow[r, "\beta"] \arrow[d, "\mathcal{S}(M')"'] & N' \arrow[d, "\mathcal{S}(N')"] \\
M \arrow[r, "\beta"'] & N
\end{tikzcd}
$$

Lemma 4.1.5. Let $M \longrightarrow N$ and $M = \mathcal{S}(M')$ for some substitution $\mathcal{S} = [x/x_1, x/x_2]$ where x_1, x_2 occur free in M' and x does not occur in M'. Then there exists a term N' such that $N = \mathcal{S}(N')$ and $M' \longrightarrow N'$.

Proof. This trivially holds because $\mathcal{S}$ simply renames free variables – suppose that in M we annotate each occurrence of x with the variable it substituted (so that each occurrence is either x^{x_1} or x^{x_2}). Then M is equal to M', up to renaming of variables. $\square$

Convention 4.1.1. Without loss of generality, we assume that, in a derivation tree, all contracted variables (i.e., variables that, at some point in the derivation tree, disappear from the term and environment by an application of the (Contraction) rule) are different from any other variable in the derivation tree.

We also assume that when applying (Contraction), the new variables that substitute the contracted ones are also different from any other variable in the derivation tree.

Lemma 4.1.6 (Quantitative subject reduction)**.** If $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ is tight and $M \longrightarrow N$, then $b \geq 1$ and there exists a tight derivation Φ' such that $\Phi' \rhd \Gamma \vdash^{b-1} N : (\sigma, s)$.

Proof. We prove the following stronger statement:

Assume $M \longrightarrow N$, $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$, $\mathsf{tight}(\Gamma)$, and either $\mathsf{tight}(s)$ or $\neg\mathsf{abs}(M)$.

Then there exists a derivation $\Phi' \rhd \Gamma \vdash^{b-1} N : (\sigma, s)$.

We prove this statement by induction on $M \longrightarrow N$.

1. Rule $\dfrac{}{(\lambda x.M_1)N_1 \longrightarrow M_1[N_1/x]}$:

   Assume $\Phi \rhd \Gamma \vdash^b (\lambda x.M_1)N_1 : (\sigma, s)$ and $\mathsf{tight}(\Gamma)$.

   So at some point in the derivation Φ, either the rule ($\multimap$ Elim) or ($\rightarrow$ Elim) is applied (not ($\multimap$ $\mathrm{Elim_t}$) nor ($\rightarrow$ $\mathrm{Elim_t}$) because it is not possible to derive the type $\mathsf{Neutral}$ for an abstraction) and is then followed by zero or more applications of the rules (Exchange) and/or (Contraction).

   Case where the last rule different from (Exchange) and (Contraction) applied in Φ is:

   (a) ($\multimap$ Elim):

   Then at some point in Φ we have:

   $$\dfrac{\Phi_1 \rhd \Gamma_1 \vdash^{b_1} (\lambda x.M_1') : (\tau \multimap \sigma, t \multimap s) \qquad \Phi_2 \rhd \Gamma_2 \vdash^{b_2} N_1' : (\tau, t)}{\Gamma_1, \Gamma_2 \vdash^{b_1+b_2} (\lambda x.M_1')N_1' : (\sigma, s)}$$

Assume that after the application of this rule, the rule (Contraction) was applied n times (with $n \geq 0$) and (Exchange) zero or more times, and let $\mathcal{S} = [y_n/x_n, y_n/x'_n] \circ [y_{n-1}/x_{n-1}, y_{n-1}/x'_{n-1}] \circ \cdots \circ [y_1/x_1, y_1/x'_1]$ be the substitution that reflects the n applications of the rule (Contraction). Then we have:

- $M_1 = \mathcal{S}(M'_1)$;
- $N_1 = \mathcal{S}(N'_1)$;
- $b = b_1 + b_2$.

Since $\Phi_1 \rhd \Gamma_1 \vdash^{b_1} (\lambda x.M'_1) : (\tau \multimap \sigma, t \multimap s)$, at some point in the derivation Φ_1, the rule ($\multimap$ Intro) is applied, followed by zero or more applications of the rules (Exchange) and/or (Contraction). So at some point in Φ_1 we have:

$$\frac{\Phi'_1 \rhd \Gamma'_1, x : (\tau, t) \vdash^{b'_1} M''_1 : (\sigma, s)}{\Gamma'_1 \vdash^{b'_1+1} \lambda x.M''_1 : (\tau \multimap \sigma, t \multimap s)}$$

Assume that after the application of this rule, the rule (Contraction) was applied m times (with $m \geq 0$) and (Exchange) zero or more times, and let $\mathcal{S}_1 = [y'_n/z_m, y'_n/z'_m] \circ [y'_{m-1}/z_{m-1}, y'_{m-1}/z'_{m-1}] \circ \cdots \circ [y'_1/z_1, y'_1/z'_1]$ be the substitution that reflects the m applications of the rule (Contraction). Then we have:

- $M'_1 = \mathcal{S}_1(M''_1)$;
- $b_1 = b'_1 + 1$.

We can then apply Lemma 4.1.4 for the derivations $\Phi'_1 \rhd \Gamma'_1, x : (\tau, t) \vdash^{b'_1} M''_1 : (\sigma, s)$ and $\Phi_2 \rhd \Gamma_2 \vdash^{b_2} N'_1 : (\tau, t)$ to obtain

$$\Phi'' \rhd \Gamma'_1, \Gamma_2 \vdash^{b'_1+b_2} M''_1[N'_1/x] : (\sigma, s).$$

Note that we can assume Γ'_1, Γ_2 to be consistent by Convention 4.1.1 and the fact that Γ_1, Γ_2 is consistent.

If we perform the m applications of (Contraction) (and the necessary applications of (Exchange)) over the same variables over which they were performed in Φ_1 after the application of ($\multimap$ Intro), i.e., over the variables $z_1, z'_1, z_2, z'_2, \ldots, z_m, z'_m$, we can obtain:

$$\Gamma'_3 \vdash^{b'_1+b_2} \mathcal{S}_1(M''_1[N'_1/x]) : (\sigma, s)$$

where $\Gamma'_3 \equiv (\Gamma_1, \Gamma_2)$. (By Convention 4.1.1, the variables in Γ_2 are not substituted.)

If we now perform the n applications of (Contraction) (and the necessary applications of (Exchange)) over the same variables over which they were performed in Φ after the application of ($\multimap$ Elim), i.e., over the variables $x_1, x'_1, x_2, x'_2, \ldots, x_n, x'_n$, we can obtain:

$$\Gamma_3 \vdash^{b'_1+b_2} \mathcal{S}(\mathcal{S}_1(M''_1[N'_1/x])) : (\sigma, s)$$

where $\Gamma_3 \equiv \Gamma$.

Finally, since $\Gamma_3 \equiv \Gamma$, we can apply (Exchange) as many times as needed to get the final judgment we wanted:

$$\Gamma \vdash^{b_1'+b_2} \mathcal{S}(\mathcal{S}_1(M_1''[N_1'/x])) : (\sigma, s)$$

Since $b = b_1 + b_2$ and $b_1 = b_1' + 1$, then

$$\begin{aligned} b_1' + b_2 &= b_1 - 1 + b_2 \\ &= b - 1. \end{aligned}$$

By Convention 4.1.1, we have $\mathcal{S}_1(M_1''[N_1'/x]) = (\mathcal{S}_1(M_1''))[N_1'/x]$.
Also, $M_1' = \mathcal{S}_1(M_1'')$, so

$$\begin{aligned} \mathcal{S}_1(M_1''[N_1'/x]) &= (\mathcal{S}_1(M_1''))[N_1'/x] \\ &= M_1'[N_1'/x]. \end{aligned} \tag{1}$$

Because x cannot be in $\mathcal{S}$, then $\mathcal{S}(M_1'[N_1'/x]) = (\mathcal{S}(M_1'))[\mathcal{S}(N_1')/x]$.
And since $M_1 = \mathcal{S}(M_1')$ and $N_1 = \mathcal{S}(N_1')$, we have $(\mathcal{S}(M_1'))[\mathcal{S}(N_1')/x] = M_1[N_1/x]$, so

$$\begin{aligned} \mathcal{S}(M_1'[N_1'/x]) &= (\mathcal{S}(M_1'))[\mathcal{S}(N_1')/x] \\ &= M_1[N_1/x]. \end{aligned} \tag{2}$$

Then, by (1) and (2) we have:

$$\begin{aligned} \mathcal{S}(\mathcal{S}_1(M_1''[N_1'/x])) &= \mathcal{S}(M_1'[N_1'/x]) \\ &= M_1[N_1/x]. \end{aligned}$$

So there is indeed $\Phi' \rhd \Gamma \vdash^{b-1} M_1[N_1/x] : (\sigma, s)$.

(b) ($\to$ Elim):

Then at some point in Φ we have:

$$\frac{\Phi_1' \rhd \Gamma_1' \vdash^{b_1'} (\lambda x.M_1') : (\tau_1 \cap \cdots \cap \tau_k \to \sigma, t_1 \cap \cdots \cap t_k \to s) \quad \Phi_1 \rhd \Gamma_1 \vdash^{b_1} N_1' : (\tau_1, t_1) \ \cdots \ \Phi_k \rhd \Gamma_k \vdash^{b_k} N_1' : (\tau_k, t_k) \qquad k \geq 2}{\Gamma_1', \sum_{i=1}^{k} \Gamma_i \vdash^{b_1'+b_1+\cdots+b_k} (\lambda x.M_1')N_1' : (\sigma, s)}$$

Assume that after the application of this rule, the rule (Contraction) was applied n times (with $n \geq 0$) and (Exchange) zero or more times, and let $\mathcal{S} = [y_n/x_n, y_n/x_n'] \circ [y_{n-1}/x_{n-1}, y_{n-1}/x_{n-1}'] \circ \cdots \circ [y_1/x_1, y_1/x_1']$ be the substitution that reflects the n applications of the rule (Contraction). Then we have:

- $M_1 = \mathcal{S}(M_1')$;
- $N_1 = \mathcal{S}(N_1')$;
- $b = b_1' + b_1 + \cdots + b_k$.

Since $\Phi_1' \rhd \Gamma_1' \vdash^{b_1'} (\lambda x.M_1') : (\tau_1 \cap \cdots \cap \tau_k \to \sigma, t_1 \cap \cdots \cap t_k \to s)$, at some point in the derivation Φ_1', the rule ($\to$ Intro) is applied, followed by zero or more applications of the rules (Exchange) and/or (Contraction). So at some point in Φ_1' we have:

$$\frac{\Phi_1'' \rhd \Gamma_1'', x : (\tau_1 \cap \cdots \cap \tau_k, t_1 \cap \cdots \cap t_k) \vdash^{b_1''} M_1'' : (\sigma, s) \qquad k \geq 2}{\Gamma_1'' \vdash^{b_1''+1} \lambda x.M_1'' : (\tau_1 \cap \cdots \cap \tau_k \to \sigma, t_1 \cap \cdots \cap t_k \to s)}$$

Assume that after the application of this rule, the rule (Contraction) was applied m times (with $m \geq 0$) and (Exchange) zero or more times, and let $\mathcal{S}_1 = [y_n'/z_m, y_n'/z_m'] \circ [y_{m-1}'/z_{m-1}, y_{m-1}'/z_{m-1}'] \circ \cdots \circ [y_1'/z_1, y_1'/z_1']$ be the substitution that reflects the m applications of the rule (Contraction). Then we have:

- $M_1' = \mathcal{S}_1(M_1'')$;
- $b_1' = b_1'' + 1$.

We can then apply Lemma 4.1.4 for the derivations $\Phi_1'' \rhd \Gamma_1'', x : (\tau_1 \cap \cdots \cap \tau_k, t_1 \cap \cdots \cap t_k) \vdash^{b_1''} M_1'' : (\sigma, s)$ and $\Phi_1 \rhd \Gamma_1 \vdash^{b_1} N_1' : (\tau_1, t_1), \ldots, \Phi_k \rhd \Gamma_k \vdash^{b_k} N_1' : (\tau_k, t_k)$ to obtain

$$\Phi'' \rhd \Gamma_1'', \sum_{i=1}^{k} \Gamma_i \vdash^{b_1''+b_1+\cdots+b_k} M_1''[N_1'/x] : (\sigma, s).$$

Note that we can assume $\Gamma_1'', \sum_{i=1}^{k} \Gamma_i$ to be consistent by Convention 4.1.1 and the fact that $\Gamma_1', \sum_{i=1}^{k} \Gamma_i$ is consistent.

If we perform the m applications of (Contraction) (and the necessary applications of (Exchange)) over the same variables over which they were performed in Φ_1' after the application of ($\to$ Intro), i.e., over the variables $z_1, z_1', z_2, z_2', \ldots, z_m, z_m'$, we can obtain:

$$\Gamma'' \vdash^{b_1''+b_1+\cdots+b_k} \mathcal{S}_1(M_1''[N_1'/x]) : (\sigma, s)$$

where $\Gamma'' \equiv (\Gamma_1', \sum_{i=1}^{k} \Gamma_i)$. (By Convention 4.1.1, the variables in $\sum_{i=1}^{k} \Gamma_i$ are not substituted.)

If we now perform the n applications of (Contraction) (and the necessary applications of (Exchange)) over the same variables over which they were performed in Φ after the application of ($\to$ Elim), i.e., over the variables $x_1, x_1', x_2, x_2', \ldots, x_n, x_n'$, we can obtain:

$$\Gamma' \vdash^{b_1''+b_1+\cdots+b_k} \mathcal{S}(\mathcal{S}_1(M_1''[N_1'/x])) : (\sigma, s)$$

where $\Gamma' \equiv \Gamma$.

Finally, since $\Gamma' \equiv \Gamma$, we can apply (Exchange) as many times as needed to get the final judgment we wanted:

$$\Gamma \vdash^{b_1''+b_1+\cdots+b_k} \mathcal{S}(\mathcal{S}_1(M_1''[N_1'/x])) : (\sigma, s)$$

Since $b = b_1' + b_1 + \cdots + b_k$ and $b_1' = b_1'' + 1$, then

$$\begin{aligned} b_1'' + b_1 + \cdots + b_k &= b_1' - 1 + b_1 + \cdots + b_k \\ &= b - 1. \end{aligned}$$

By Convention 4.1.1, we have $\mathcal{S}_1(M_1''[N_1'/x]) = (\mathcal{S}_1(M_1''))[N_1'/x]$.
Also, $M_1' = \mathcal{S}_1(M_1'')$, so

$$\begin{aligned} \mathcal{S}_1(M_1''[N_1'/x]) &= (\mathcal{S}_1(M_1''))[N_1'/x] \\ &= M_1'[N_1'/x]. \end{aligned} \tag{1}$$

Because x cannot be in $\mathcal{S}$, then $\mathcal{S}(M_1'[N_1'/x]) = (\mathcal{S}(M_1'))[\mathcal{S}(N_1')/x]$.
And since $M_1 = \mathcal{S}(M_1')$ and $N_1 = \mathcal{S}(N_1')$, we have $(\mathcal{S}(M_1'))[\mathcal{S}(N_1')/x] = M_1[N_1/x]$, so

$$\begin{aligned} \mathcal{S}(M_1'[N_1'/x]) &= (\mathcal{S}(M_1'))[\mathcal{S}(N_1')/x] \\ &= M_1[N_1/x]. \end{aligned} \tag{2}$$

Then, by (1) and (2) we have:

$$\begin{aligned} \mathcal{S}(\mathcal{S}_1(M_1''[N_1'/x])) &= \mathcal{S}(M_1'[N_1'/x]) \\ &= M_1[N_1/x]. \end{aligned}$$

So there is indeed $\Phi' \triangleright \Gamma \vdash^{b-1} M_1[N_1/x] : (\sigma, s)$.

2. Rule $\dfrac{M_1 \longrightarrow M_2}{\lambda x.M_1 \longrightarrow \lambda x.M_2}$:

Assume $\Phi \triangleright \Gamma \vdash^{b} \lambda x.M_1 : (\sigma, s)$, $\mathsf{tight}(\Gamma)$ and the premise $M_1 \longrightarrow M_2$.

Since $\mathsf{abs}(\lambda x.M_1)$, we must have hypothesis $\mathsf{tight}(s)$.

So at some point in the derivation Φ, either the rule ($\multimap$ Intro$_\mathrm{t}$) or ($\to$ Intro$_\mathrm{t}$) is applied and is then followed by zero or more applications of the rules (Exchange) and/or (Contraction).

As the two cases are similar, we will only show the case in which the last rule different from (Exchange) and (Contraction) applied in Φ is ($\multimap$ Intro$_\mathrm{t}$):

Then at some point in Φ we have:

$$\frac{\Phi_1 \rhd \Gamma', x : (\tau, \mathsf{tight}) \vdash^{b'} M_1' : (\sigma_1, \mathsf{tight})}{\Gamma' \vdash^{b'} \lambda x. M_1' : (\tau \multimap \sigma_1, \mathsf{Abs})}$$

where $\sigma = \tau \multimap \sigma_1$ and $s = \mathsf{Abs}$.

Assume that after the application of this rule, the rule (Contraction) was applied n times (with $n \geq 0$) and (Exchange) zero or more times, and let $\mathcal{S} = [y_n/x_n, y_n/x_n'] \circ [y_{n-1}/x_{n-1}, y_{n-1}/x_{n-1}'] \circ \cdots \circ [y_1/x_1, y_1/x_1']$ be the substitution that reflects the n applications of the rule (Contraction). Then we have:

- $M_1 = \mathcal{S}(M_1')$;
- $b = b'$.

Since $\mathsf{tight}(\Gamma)$, we have $\mathsf{tight}(\Gamma', x : (\tau, \mathsf{tight}))$.

Since $M_1 \longrightarrow M_2$ and $M_1 = \mathcal{S}(M_1')$, by applying Lemma 4.1.5 n times for the substitutions resulting from the n contractions, we get a term M_2' such that $M_2 = \mathcal{S}(M_2')$ and $M_1' \longrightarrow M_2'$.

Then we can apply the induction hypothesis on Φ_1 and get a derivation ending with

$$\Gamma', x : (\tau, \mathsf{tight}) \vdash^{b'-1} M_2' : (\sigma_1, \mathsf{tight}).$$

By rule ($\multimap$ Intro$_\mathrm{t}$), we have:

$$\frac{\Gamma', x : (\tau, \mathsf{tight}) \vdash^{b'-1} M_2' : (\sigma_1, \mathsf{tight})}{\Gamma' \vdash^{b'-1} \lambda x. M_2' : (\tau \multimap \sigma_1, \mathsf{Abs})}$$

If we now perform the n applications of (Contraction) (and the necessary applications of (Exchange)) over the same variables over which they were performed in Φ after the application of ($\multimap$ Intro$_\mathrm{t}$), i.e., over the variables $x_1, x_1', x_2, x_2', \ldots, x_n, x_n'$, we can obtain:

$$\Gamma_2 \vdash^{b'-1} \mathcal{S}(\lambda x. M_2') : (\tau \multimap \sigma_1, \mathsf{Abs})$$

where $\Gamma_2 \equiv \Gamma$.

Finally, since $\Gamma_2 \equiv \Gamma$, we can apply (Exchange) as many times as needed to get the final judgment we wanted:

$$\Gamma \vdash^{b'-1} \mathcal{S}(\lambda x. M_2') : (\tau \multimap \sigma_1, \mathsf{Abs})$$

Since $b = b'$, then $b' - 1 = b - 1$.

And since $M_2 = \mathcal{S}(M_2')$ and x cannot be in $\mathcal{S}$, we have $\mathcal{S}(\lambda x.M_2') = \lambda x.M_2$.

So there is indeed $\Phi' \rhd \Gamma \vdash^{b-1} \lambda x.M_2 : (\sigma, s)$.

3. Rule $\dfrac{M_1 \longrightarrow M_2 \qquad \neg\mathsf{abs}(M_1)}{M_1 N_1 \longrightarrow M_2 N_1}$:

Assume $\Phi \rhd \Gamma \vdash^{b} M_1 N_1 : (\sigma, s)$, $\mathsf{tight}(\Gamma)$ and the premises $M_1 \longrightarrow M_2$ and $\neg\mathsf{abs}(M_1)$.

So at some point in the derivation Φ, either the rule ($\multimap$ Elim), or ($\multimap$ Elim_t), or ($\to$ Elim), or ($\to$ Elim_t) is applied and is then followed by zero or more applications of the rules (Exchange) and/or (Contraction).

As the four cases are similar, we will only show the case in which the last rule different from (Exchange) and (Contraction) applied in Φ is ($\multimap$ Elim):

Then at some point in Φ we have:

$$\dfrac{\Phi_1 \rhd \Gamma_1 \vdash^{b_1} M_1' : (\tau \multimap \sigma, t \multimap s) \qquad \Phi_2 \rhd \Gamma_2 \vdash^{b_2} N_1' : (\tau, t)}{\Gamma_1, \Gamma_2 \vdash^{b_1 + b_2} M_1' N_1' : (\sigma, s)}$$

Assume that after the application of this rule, the rule (Contraction) was applied n times (with $n \geq 0$) and (Exchange) zero or more times, and let $\mathcal{S} = [y_n/x_n, y_n/x_n'] \circ [y_{n-1}/x_{n-1}, y_{n-1}/x_{n-1}'] \circ \cdots \circ [y_1/x_1, y_1/x_1']$ be the substitution that reflects the n applications of the rule (Contraction). Then we have:

- $M_1 = \mathcal{S}(M_1')$;
- $N_1 = \mathcal{S}(N_1')$;
- $b = b_1 + b_2$.

Since $\mathsf{tight}(\Gamma)$, we have $\mathsf{tight}(\Gamma_1)$. And also, as $\neg\mathsf{abs}(M_1)$, then $\neg\mathsf{abs}(M_1')$ ($\mathcal{S}$ simply renames free variables).

Since $M_1 \longrightarrow M_2$ and $M_1 = \mathcal{S}(M_1')$, by applying Lemma 4.1.5 n times for the substitutions resulting from the n contractions, we get a term M_2' such that $M_2 = \mathcal{S}(M_2')$ and $M_1' \longrightarrow M_2'$.

Then we can apply the induction hypothesis on Φ_1 and get a derivation ending with

$$\Gamma_1 \vdash^{b_1 - 1} M_2' : (\tau \multimap \sigma, t \multimap s).$$

By rule ($\multimap$ Elim), with $\Gamma_2 \vdash^{b_2} N_1' : (\tau, t)$ from Φ_2, we have:

$$\frac{\Gamma_1 \vdash^{b_1-1} M_2' : (\tau \multimap \sigma, t \multimap s) \qquad \Gamma_2 \vdash^{b_2} N_1' : (\tau, t)}{\Gamma_1, \Gamma_2 \vdash^{b_1+b_2-1} M_2'N_1' : (\sigma, s)}$$

If we now perform the n applications of (Contraction) (and the necessary applications of (Exchange)) over the same variables over which they were performed in Φ after the application of ($\multimap$ Elim), i.e., over the variables $x_1, x_1', x_2, x_2', \ldots, x_n, x_n'$, we can obtain:

$$\Gamma_3 \vdash^{b_1+b_2-1} \mathcal{S}(M_2'N_1') : (\sigma, s)$$

where $\Gamma_3 \equiv \Gamma$.

Finally, since $\Gamma_3 \equiv \Gamma$, we can apply (Exchange) as many times as needed to get the final judgment we wanted:

$$\Gamma \vdash^{b_1+b_2-1} \mathcal{S}(M_2'N_1') : (\sigma, s)$$

Since $b = b_1 + b_2$, then $b_1 + b_2 - 1 = b - 1$.

And since $M_2 = \mathcal{S}(M_2')$ and $N_1 = \mathcal{S}(N_1')$, we have $\mathcal{S}(M_2'N_1') = M_2N_1$.

So there is indeed $\Phi' \rhd \Gamma \vdash^{b-1} M_2N_1 : (\sigma, s)$.

4. Rule $\dfrac{\mathsf{neutral}(N_1) \qquad M_1 \longrightarrow M_2}{N_1M_1 \longrightarrow N_1M_2}$:

Assume $\Phi \rhd \Gamma \vdash^{b} N_1M_1 : (\sigma, s)$, $\mathsf{tight}(\Gamma)$ and the premises $\mathsf{neutral}(N_1)$ and $M_1 \longrightarrow M_2$.

So at some point in the derivation Φ, either the rule ($\multimap$ Elim), or ($\multimap$ $\mathrm{Elim_t}$), or ($\rightarrow$ Elim), or ($\rightarrow$ $\mathrm{Elim_t}$) is applied, giving $\Gamma' \vdash^{b} N_1'M_1' : (\sigma, s)$, and is then followed by zero or more applications of the rules (Exchange) and/or (Contraction).

Assume that the rule (Contraction) is applied n times (with $n \geq 0$) and (Exchange) zero or more times, and let $\mathcal{S} = [y_n/x_n, y_n/x_n'] \circ [y_{n-1}/x_{n-1}, y_{n-1}/x_{n-1}'] \circ \cdots \circ [y_1/x_1, y_1/x_1']$ be the substitution that reflects the n applications of the rule (Contraction). Then we have:

- $N_1 = \mathcal{S}(N_1')$;
- $M_1 = \mathcal{S}(M_1')$.

Then as a premise for any of those four rules, we have a derivation for N_1' of the form:

$$\Phi_1 \rhd \Gamma_1 \vdash^{b_1} N_1' : (\sigma', s')$$

Since $\mathsf{neutral}(N_1)$, we have $\mathsf{neutral}(N_1')$ ($\mathcal{S}$ simply renames free variables).

Also, since $\mathsf{tight}(\Gamma)$, independently from the elimination rule that was applied, we have $\mathsf{tight}(\Gamma_1)$.

Then by Lemma 4.1.1, we have $\mathsf{tight}(s')$.

So this means that actually, the last rule different from (Exchange) and (Contraction) applied in Φ must be either ($\multimap$ Elim$_\mathrm{t}$) or ($\rightarrow$ Elim$_\mathrm{t}$), and not ($\multimap$ Elim) nor ($\rightarrow$ Elim).

And as the two cases are similar, we will only show the case in which the last rule different from (Exchange) and (Contraction) applied in Φ is ($\multimap$ Elim$_\mathrm{t}$):

Then, before the n applications of the rule (Contraction) and possible applications of (Exchange) what we have is:

$$\frac{\Phi_1 \rhd \Gamma_1 \vdash^{b_1} N_1' : (\tau \multimap \sigma, \mathsf{Neutral}) \qquad \Phi_2 \rhd \Gamma_2 \vdash^{b_2} M_1' : (\tau, \mathsf{tight})}{\Gamma_1, \Gamma_2 \vdash^{b_1+b_2} N_1' M_1' : (\sigma, \mathsf{Neutral})}$$

and

- $s = \mathsf{Neutral}$;
- $N_1 = \mathcal{S}(N_1')$;
- $M_1 = \mathcal{S}(M_1')$;
- $b = b_1 + b_2$.

Since $\mathsf{tight}(\Gamma)$, we have $\mathsf{tight}(\Gamma_2)$.

Since $M_1 \longrightarrow M_2$ and $M_1 = \mathcal{S}(M_1')$, by applying Lemma 4.1.5 n times for the substitutions resulting from the n contractions, we get a term M_2' such that $M_2 = \mathcal{S}(M_2')$ and $M_1' \longrightarrow M_2'$.

Then we can apply the induction hypothesis on Φ_2 and get a derivation ending with

$$\Gamma_2 \vdash^{b_2-1} M_2' : (\tau, \mathsf{tight}).$$

By rule ($\multimap$ Elim$_\mathrm{t}$), with $\Gamma_1 \vdash^{b_1} N_1' : (\tau \multimap \sigma, \mathsf{Neutral})$ from Φ_1, we have:

$$\frac{\Gamma_1 \vdash^{b_1} N_1' : (\tau \multimap \sigma, \mathsf{Neutral}) \qquad \Gamma_2 \vdash^{b_2-1} M_2' : (\tau, \mathsf{tight})}{\Gamma_1, \Gamma_2 \vdash^{b_1+b_2-1} N_1' M_2' : (\sigma, \mathsf{Neutral})}$$

If we now perform the n applications of (Contraction) (and the necessary applications of (Exchange)) over the same variables over which they were performed in Φ after the application of ($\multimap$ Elim$_\mathrm{t}$), i.e., over the variables $x_1, x_1', x_2, x_2', \ldots, x_n, x_n'$, we can obtain:

$$\Gamma_3 \vdash^{b_1+b_2-1} \mathcal{S}(N_1' M_2') : (\sigma, \mathsf{Neutral})$$

where $\Gamma_3 \equiv \Gamma$.

Finally, since $\Gamma_3 \equiv \Gamma$, we can apply (Exchange) as many times as needed to get the final judgment we wanted:

$$\Gamma \vdash^{b_1+b_2-1} \mathcal{S}(N_1' M_2') : (\sigma, \mathsf{Neutral})$$

Since $b = b_1 + b_2$, then $b_1 + b_2 - 1 = b - 1$.

Also, $s = \mathsf{Neutral}$.

And since $N_1 = \mathcal{S}(N_1')$ and $M_2 = \mathcal{S}(M_2')$, we have $\mathcal{S}(N_1' M_2') = N_1 M_2$.

So there is indeed $\Phi' \rhd \Gamma \vdash^{b-1} N_1 M_2 : (\sigma, s)$.

□

Theorem 4.1.7 (Tight correctness)**.** If $\Phi \rhd \Gamma \vdash^{b} M : (\sigma, s)$ is a tight derivation, then there exists N such that $M \longrightarrow^{b} N$ and $\mathsf{normal}(N)$. Moreover, if $s = \mathsf{Neutral}$ then $\mathsf{neutral}(N)$.

Proof. By induction on the evaluation length of $M \longrightarrow^{k} N$.

If M is a (leftmost-outermost) normal form, then by taking $N = M$ and $k = 0$, the statement follows from the tightness property of tight typings of normal forms (Lemma 4.1.2(i)). The *moreover* part follows from the neutrality property (Lemma 4.1.2(ii)).

Otherwise, $M \longrightarrow M'$ and by quantitative subject reduction (Lemma 4.1.6) there exists a derivation $\Phi' \rhd \Gamma \vdash^{b-1} M' : (\sigma, s)$. By induction, there exists N such that $\mathsf{normal}(N)$ and $M' \longrightarrow^{b-1} N$. Note that $M \longrightarrow M' \longrightarrow^{b-1} N$, that is, $M \longrightarrow^{b} N$. □

4.2 Type Inference Algorithm

We now extend the type inference algorithm defined in Chapter 3 (Definition 3.3.7) to also infer the number of reduction steps of the typed term to its normal form, when using the leftmost-outermost evaluation strategy.

This is done by slightly modifying the unification algorithm in Definition 3.3.6 and the algorithm in Definition 3.3.7, which will now carry and update a measure b that relates to the number of reduction steps.

First, recall the following definition, presented in Chapter 3:

Definition 4.2.1 (Type unification)**.** We define the following relation $\Rightarrow$ on type unification problems (for types in $\mathbb{T}_{\mathbb{L}0}$):

$$
\begin{array}{lll l}
\{\tau = \tau\} \cup P & \Rightarrow & P & \\
\{\tau_1 \multimap \tau_2 = \tau_3 \multimap \tau_4\} \cup P & \Rightarrow & \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup P & \\
\{\tau_1 \multimap \tau_2 = \alpha\} \cup P & \Rightarrow & \{\alpha = \tau_1 \multimap \tau_2\} \cup P & \\
\{\alpha = \tau\} \cup P & \Rightarrow & \{\alpha = \tau\} \cup P[\tau/\alpha] & \text{if } \alpha \in \mathsf{fv}(P) \setminus \mathsf{fv}(\tau) \\
\{\alpha = \tau\} \cup P & \Rightarrow & \mathsf{FAIL} & \text{if } \alpha \in \mathsf{fv}(\tau) \text{ and } \alpha \neq \tau
\end{array}
$$

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

Definition 4.2.2 (Quantitative Unification Algorithm)**.** Let P be a unification problem (with types in $\mathbb{T}_{\mathbb{L}0}$). The new unification function $\mathsf{UNIFY_Q}(P)$, which decides whether P has a solution and, if so, returns the MGU of P and an integer b used for counting purposes in the inference algorithm, is defined as:

function $\mathsf{UNIFY_Q}(P)$
  $b := 0$;
  while $P \Rightarrow P'$ **do**
    if $P = \{\tau_1 \multimap \tau_2 = \tau_3 \multimap \tau_4\} \cup P_1$ **and** $P' = \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup P_1$ **then**
      $b := b + 1$;
    $P := P'$;
  if P is in solved form **then**
    return $(\mathbb{S}_P, b)$;
  else
    FAIL;

Let us call $\mathbb{T}_{\mathbb{L}1}$-environment to an environment as defined in Chapter 3, i.e., just like the definition we use in the current chapter, but the predicates are only the first element of the pair (i.e., a sequence from $\mathbb{T}_{\mathbb{L}1}$).

Definition 4.2.3 (Quantitative Type Inference Algorithm)**.** Let Γ be a $\mathbb{T}_{\mathbb{L}1}$-environment, M a λ-term, σ a linear rank 2 intersection type, b a quantitative measure and $\mathsf{UNIFY_Q}$ the function in Definition 4.2.2. The function $\mathsf{T_Q}(M) = (\Gamma, \sigma, b)$ defines a new type inference algorithm that gives a quantitative measure for the λ-calculus in the Linear Rank 2 Quantitative Type System, in the following way:

1. If $M = x$, then $\Gamma = [x : \alpha]$, $\sigma = \alpha$ and $b = 0$, where α is a new variable;

2. If $M = \lambda x.M_1$ and $\mathsf{T_Q}(M_1) = (\Gamma_1, \sigma_1, b_1)$ then:

   (a) if $x \notin \mathsf{dom}(\Gamma_1)$, then FAIL;

   (b) if $(x : \tau) \in \Gamma_1$, then $\mathsf{T_Q}(M) = (\Gamma_{1x}, \tau \multimap \sigma_1, b_1)$;

   (c) if $(x : \tau_1 \cap \cdots \cap \tau_n) \in \Gamma_1$ (with $n \geq 2$), then $\mathsf{T_Q}(M) = (\Gamma_{1x}, \tau_1 \cap \cdots \cap \tau_n \to \sigma_1, b_1)$.

3. If $M = M_1 M_2$, then:

   (a) if $\mathsf{T_Q}(M_1) = (\Gamma_1, \alpha_1, b_1)$ and $\mathsf{T_Q}(M_2) = (\Gamma_2, \tau_2, b_2)$,
   then $\mathsf{T_Q}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3), b_1 + b_2)$,
   where $(\mathbb{S}, \_) = \mathsf{UNIFY_Q}(\{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\})$ and α_2, α_3 are new variables;

   (b) if $\mathsf{T_Q}(M_1) = (\Gamma'_1, \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1, b_1)$ (with $n \geq 2$) and, for each $1 \leq i \leq n$, $\mathsf{T_Q}(M_2) = (\Gamma_i, \tau_i, b_i)$,
   then $\mathsf{T_Q}(M) = (\mathbb{S}(\Gamma'_1 + \sum_{i=1}^n \Gamma_i), \mathbb{S}(\sigma'_1), b_1 + \sum_{i=1}^n b_i + b_3 + 1)$,
   where $(\mathbb{S}, b_3) = \mathsf{UNIFY_Q}(\{\tau_i = \tau'_i \mid 1 \leq i \leq n\})$;

   (c) if $\mathsf{T_Q}(M_1) = (\Gamma_1, \tau \multimap \sigma_1, b_1)$ and $\mathsf{T_Q}(M_2) = (\Gamma_2, \tau_2, b_2)$,
   then $\mathsf{T_Q}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\sigma_1), b_1 + b_2 + b_3 + 1)$,
   where $(\mathbb{S}, b_3) = \mathsf{UNIFY_Q}(\{\tau_2 = \tau\})$;

   (d) otherwise FAIL.

Note that b is only increased by 1 and added the quantity given by $\mathsf{UNIFY_Q}$ in rules 3.(b) and 3.(c), since these are the only cases in which the term M is a redex.

Example 4.2.1. Let us show the type inference process for the λ-term $\lambda x.xx$.

- By rule 1., $\mathsf{T_Q}(x) = ([x : \alpha_1], \alpha_1, 0)$.
- By rule 1., again, $\mathsf{T_Q}(x) = ([x : \alpha_2], \alpha_2, 0)$.
- Then by rule 3.(a), $\mathsf{T_Q}(xx) = (\mathbb{S}([x : \alpha_1] + [x : \alpha_2]), \mathbb{S}(\alpha_4), 0+0) = (\mathbb{S}([x : \alpha_1 \cap \alpha_2]), \mathbb{S}(\alpha_4), 0)$,
  where $(\mathbb{S}, \_) = \mathsf{UNIFY_Q}(\{\alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 = \alpha_3\}) = ([\alpha_3 \multimap \alpha_4/\alpha_1, \alpha_3/\alpha_2], 0)$.
  So $\mathsf{T_Q}(xx) = ([x : (\alpha_3 \multimap \alpha_4) \cap \alpha_3], \alpha_4, 0)$.
- Finally, by rule 2.(c), $\mathsf{T_Q}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4, 0)$.

Example 4.2.2. Let us now show the type inference process for the λ-term $(\lambda x.xx)(\lambda y.y)$.

- From the previous example, we have $\mathsf{T_Q}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4, 0)$.
- By rules 1. and 2.(b), for the identity, the algorithm gives $\mathsf{T_Q}(\lambda y.y) = ([\,], \alpha_1 \multimap \alpha_1, 0)$.
- By rules 1. and 2.(b), again, for the identity, $\mathsf{T_Q}(\lambda y.y) = ([\,], \alpha_2 \multimap \alpha_2, 0)$.

- Then by rule 3.(b), $\mathsf{T_Q}((\lambda x.xx)(\lambda y.y)) = (\mathbb{S}([\,] + [\,] + [\,]), \mathbb{S}(\alpha_4), 0 + 0 + 0 + b_3 + 1) = ([\,], \mathbb{S}(\alpha_4), b_3 + 1)$,

  where $(\mathbb{S}, b_3) = \mathsf{UNIFY_Q}(\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\})$, calculated by performing the following transformations:

$$\begin{aligned}\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\} &\Rightarrow \{\alpha_1 = \alpha_3, \alpha_1 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\ &\Rightarrow \{\alpha_1 = \alpha_3, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\ &\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_4\}\\ &\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_4 = \alpha_2 \multimap \alpha_2\}\\ &\Rightarrow \{\alpha_1 = \alpha_2 \multimap \alpha_2, \alpha_3 = \alpha_2 \multimap \alpha_2, \alpha_4 = \alpha_2 \multimap \alpha_2\}\end{aligned}$$

  So $\mathbb{S} = [(\alpha_2 \multimap \alpha_2)/\alpha_1, (\alpha_2 \multimap \alpha_2)/\alpha_3, (\alpha_2 \multimap \alpha_2)/\alpha_4]$

  and $b_3 = 1$ because there was performed one transformation (the first) of the form $\{\tau_1 \multimap \tau_2 = \tau_3 \multimap \tau_4\} \cup P \Rightarrow \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup P$.

  And then, $\mathsf{T_Q}((\lambda x.xx)(\lambda y.y)) = ([\,], \alpha_2 \multimap \alpha_2, 1 + 1) = ([\,], \alpha_2 \multimap \alpha_2, 2)$.

Since the Quantitative Type Inference Algorithm only differs from the algorithm in Chapter 3 on the addition of the quantitative measure, and only infers a linear rank 2 intersection type and not a multi-type, the typing soundness (Theorem 4.2.1) and completeness (Theorem 4.2.2) are formalized in a similar way.

Theorem 4.2.1 (Typing soundness)**.** If $\mathsf{T_Q}(M) = ([x_1 : \vec{\tau}_1, \dots, x_n : \vec{\tau}_n], \sigma, b)$, then $[x_1 : (\vec{\tau}_1, \vec{t}_1), \dots, x_n : (\vec{\tau}_n, \vec{t}_n)] \vdash^{b'} M : (\sigma, s)$ (for some measure b' and multi-types $s, \vec{t}_1, \dots, \vec{t}_n$).

Proof. The proof follows as in Theorem 3.3.5 (only the non-t-indexed rules are necessary). □

Theorem 4.2.2 (Typing completeness)**.** If $\Phi \rhd [x_1 : (\vec{\tau}_1, \vec{t}_1), \dots, x_n : (\vec{\tau}_n, \vec{t}_n)] \vdash^{b} M : (\sigma, s)$, then $\mathsf{T_Q}(M) = (\Gamma', \sigma', b')$ (for some $\mathbb{T}_{\mathbb{L}1}$-environment Γ', type σ' and measure b') and there is a substitution $\mathbb{S}$ such that $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv [x_1 : \vec{\tau}_1, \dots, x_n : \vec{\tau}_n]$.

Proof. The proof follows similarly to the proof of Theorem 3.3.8 (note that even when t-indexed rules are used in the derivation, the resulting linear rank 2 intersection type is the same as when the correspondent non-t-indexed rules are used). □

As for the quantitative measure given by the algorithm, we conjecture that it corresponds to the number of evaluation steps of the typed term to normal form, when using the leftmost-outermost evaluation strategy. We strongly believe the conjecture holds, based on the attempted proofs so far and because it holds for every experimental results obtained by our implementation. We have not yet proven this property, which we formalize, in part, in the second point of the strong soundness:

Conjecture 4.2.1 (Strong soundness)**.** If $\mathsf{T_Q}(M) = ([x_1 : \vec{\tau}_1, \ldots, x_n : \vec{\tau}_n], \sigma, b)$, then:

1. There is a derivation $\Phi \rhd [x_1 : (\vec{\tau}_1, \vec{t_1}), \ldots, x_n : (\vec{\tau}_n, \vec{t_n})] \vdash^{b'} M : (\sigma, s)$ (for some measure b' and multi-types $s, \vec{t_1}, \ldots, \vec{t_n}$);
2. If Φ is a tight derivation, then $b = b'$.

Note that the second point implies, by Theorem 4.1.7, that there exists N such that $M \longrightarrow^{b} N$ and $\mathsf{normal}(N)$, which is what we conjecture.

We believe that proving this conjecture is not a trivial task. A first approach could be to try to use induction on the definition of $\mathsf{T_Q}(M)$. However, this does not work because the subderivations within a tight derivation are not necessarily tight. For that same reason, it is also not trivial to construct a tight derivation from the result given by the algorithm or from a non-tight derivation.

Thus, in order to prove this conjecture, it will be necessary to establish a relation between the algorithm and tight derivations, a result that we do not have yet and for which we think that we would need a technical development that goes beyond the scope of this thesis.

Chapter 5

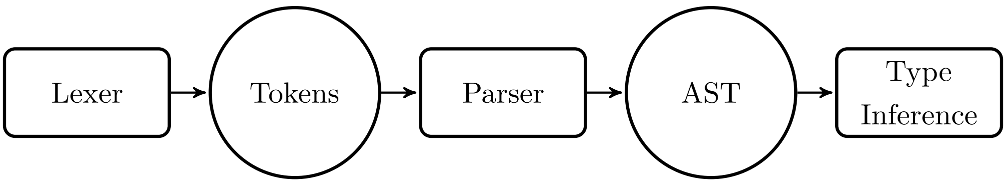
Implementation and Experimental Results

Here, we briefly describe the implementation (in Haskell) of the previously defined systems.

5.1 Implementation Overview

Besides the theoretical work presented in the previous chapters, we implemented the Quantitative Type Inference Algorithm in Haskell, as well as Milner's type inference algorithm for simple types [24], Trevor Jim's algorithm for rank 2 intersection types [19] and functions to evaluate terms to normal form for different evaluation strategies. This way, we were able to experimentally compare and verify the correctness of the empirical results.

Our final software package, in addition to the type inference algorithms, which are a tool of semantic analysis, is also composed of a lexer and a parser that were made with the parser generator Happy. 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, which then perform the semantic analysis.



The full Haskell implementation, along with input examples, can be found in https://github.com/toko18/LinearRankIntersectionTypes-MastersThesis.

In Appendix A, we include the code for the main modules of the project, which are organized in the following way:

- `LambdaCalculus` (A.1) defines λ-terms;
- `LinearTypes` (A.2) defines linear types and includes the implementation `unifyQ` of the Quantitative Unification Algorithm $\mathsf{UNIFY_Q}$;
- `LinearRank2QuantitativeTypes` (A.3) defines linear rank 1 and 2 intersection types and includes the implementation `quantR2typeInf` of the Quantitative Type Inference Algorithm $\mathsf{T_Q}$;
- `Reductions` (A.4) defines the functions `maximal`, `normal` and `applicative` that are called by the `reduce` function to reduce terms to normal form using the maximal, normal and applicative evaluation strategies, respectively;
- `parser.y` (A.5) is the description of the parser to be generated with Happy.

5.2 Experimental Results

We tested the Quantitative Type Inference Algorithm for several λ-terms in order to access the correctness of the inferred measure. The table below shows some of those results, which we obtained by running the implemented type inference algorithm and the reduction functions for the leftmost-outermost strategy (also known as normal order) and the leftmost-innermost strategy (also known as applicative order).

λ-Term	**Environment, Type**	**Count**	**Steps** (normal)	**Steps** (applicative)
$(xy)y$	$[(x, \alpha_2 \multimap \alpha_5 \multimap \alpha_6), (y, \alpha_5 \cap \alpha_2)], \alpha_6$	0	0	0
$\lambda x.xx$	$[\,], (\alpha_2 \cap (\alpha_2 \multimap \alpha_3)) \to \alpha_3$	0	0	0
$\lambda fx.f(fx)$	$[\,], ((\alpha_3 \multimap \alpha_5) \cap (\alpha_5 \multimap \alpha_6)) \to (\alpha_3 \multimap \alpha_6)$	0	0	0
$(\lambda fx.f(fx))(\lambda x.x)$	$[\,], \alpha_6 \multimap \alpha_6$	3	3	3
$(\lambda fx.f(fx))((\lambda x.xx)y)$	$[(y, (\alpha_{15} \multimap \alpha_5 \multimap \alpha_6) \cap \alpha_{15} \cap (\alpha_9 \multimap \alpha_3 \multimap \alpha_5) \cap \alpha_9)], \alpha_3 \multimap \alpha_6$	3	3	2
$(\lambda x.xx)(\lambda y.y)$	$[\,], \alpha_4 \multimap \alpha_4$	2	2	2
$(\lambda x.xxx)(\lambda y.y)$	$[\,], \alpha_7 \multimap \alpha_7$	3	3	3
$(\lambda x.xxx)(\lambda y.y)(\lambda fx.fx)$	$[\,], (\alpha_{12} \multimap \alpha_{13}) \multimap \alpha_{12} \multimap \alpha_{13}$	4	4	4
$(\lambda x.xxx)(\lambda fx.fx)(\lambda y.y)$	$[\,], \alpha_{10} \multimap \alpha_{10}$	7	7	7
$(\lambda fx.f(f(fx)))(\lambda fx.fx)$	$[\,], (\alpha_{12} \multimap \alpha_{13}) \multimap \alpha_{12} \multimap \alpha_{13}$	6	6	6
$(\lambda x.x(x(xx)))(\lambda y.y)$	$[\,], \alpha_{10} \multimap \alpha_{10}$	4	4	4
$(\lambda y.(\lambda x.xxx)y)(\lambda x.x)$	$[\,], \alpha_{12} \multimap \alpha_{12}$	4	4	4
$(\lambda x.xxx)y$	$[(y, (\alpha_2 \multimap \alpha_5 \multimap \alpha_6) \cap \alpha_2 \cap \alpha_5)], \alpha_6$	1	1	1
$(\lambda x.xxx)((\lambda x.x)y)$	$[(y, (\alpha_2 \multimap \alpha_5 \multimap \alpha_6) \cap \alpha_2 \cap \alpha_5)], \alpha_6$	4	4	2
$(\lambda x.xxxx)((\lambda x.x)y)$	$[(y, (\alpha_2 \multimap \alpha_5 \multimap \alpha_8 \multimap \alpha_9) \cap \alpha_2 \cap \alpha_5 \cap \alpha_8)], \alpha_9$	5	5	2
$(\lambda x.x)((\lambda x.x)(\lambda x.x))$	$[\,], \alpha_2 \multimap \alpha_2$	2	2	2
$(\lambda x.x)((\lambda x.x)(\lambda x.x))(\lambda x.x)$	$[\,], \alpha_3 \multimap \alpha_3$	3	3	3
$(\lambda y.y(\lambda x.x))(\lambda x.x)$	$[\,], \alpha_1 \multimap \alpha_1$	2	2	2
$(\lambda xyz.xz(yz))(\lambda x.x)$	$[\,], (\alpha_6 \multimap \alpha_8) \multimap ((\alpha_6 \cap (\alpha_8 \multimap \alpha_9)) \to \alpha_9)$	2	2	2
$(\lambda x.(\lambda y.yx)x)(\lambda x.x)$	$[\,], \alpha_6 \multimap \alpha_6$	3	3	3

Table 5.1: Environment, type and quantitative measure (Count) given by the inference algorithm, and number of reduction steps to normal form when using normal order (leftmost-outermost strategy) and applicative order (leftmost-innermost strategy), for each λ-term tested.

As shown in these results, as expected, the algorithm correctly gives the number of evaluation steps of the terms to normal form, for the leftmost-outermost evaluation strategy. Although we still need to prove the correctness of the quantities inferred, the results obtained are promising.

Chapter 6

Conclusions and Future Work

Quantitative type systems are a powerful tool for static program analysis, but in addition to the qualitative information, they also provide quantitative information about program evaluations that can be used to estimate their time and space complexities in compile time, which allows us to know in advance the computational resources that will be required to run the program.

Intersection type systems characterize termination so, in order to make typability decidable, one can restrict the intersection types by using the notion of finite rank introduced by Daniel Leivant [23]. When developing a non-idempotent intersection type system capable of obtaining quantitative information about a λ-term while inferring its type, we realized that the classical notion of rank was not a proper fit for non-idempotent intersection types, and that the ranks could be quantitatively more useful if the base case was changed to types that give more quantitative information in comparison to simple types, which is the case for linear types. We then came up with a new definition of rank for intersection types based on linear types, which we call *linear rank* [25].

Based on the notion of linear rank, we defined a new intersection type system for the λ-calculus, restricted to linear rank 2 non-idempotent intersection types, and a new type inference algorithm (based on Trevor Jim's [19]), which we proved to be sound and complete with respect to the type system.

We then merged that intersection type system with the system for the leftmost-outermost evaluation strategy presented in [1] in order to use the linear rank 2 non-idempotent intersection types to obtain quantitative information about the typed terms, and we proved that the resulting type system gives the correct number of evaluation steps for a kind of derivations. We also extended the type inference algorithm we had defined, in order to also give that measure, and showed that it is sound and complete with respect to the type system for the inferred types, and conjectured that the inferred measures correspond to the ones given by the type system.

Finally, in order to test the new inference algorithm, we implemented it in Haskell, as well as other type inference algorithms and procedures to evaluate terms to normal form for different evaluation strategies.

Although we left unproven the correctness of the quantities inferred by the Quantitative Type Inference Algorithm, the goals of this work were fulfilled, and we believe that it comprises a fair contribution to the area. We argue that our Quantitative Type Inference Algorithm is a first step towards the automatic inference of truly quantitative types, and we also believe that the Linear Rank 2 Intersection Type System alone can have interesting properties that can be used in other topics, such as linearity.

Regarding the proofs we presented, we believe that many of them would be much simpler if we had defined environments as sets and used solely the (+) operation for concatenation, instead of defining environments as lists and having the rules (Exchange) and (Contraction) in the type systems. But that was the price we had to pay in order to have a system that is closer to a linear type system, makes us have more control over linearity and non-linearity, and is more easily extensible for other algebraic properties of intersection.

In the future, we would like to:

- prove Conjecture 4.2.1;
- further explore the relation between our definition of linear rank and the classical definition of rank;
- extend the type systems and the type inference algorithms for the affine terms;
- adapt the Linear Rank 2 Quantitative Type System and the Quantitative Type Inference Algorithm for other evaluation strategies;
- extend them for a simple programming language like Core Haskell (the intermediate language used by the Haskell compiler GHC).

Bibliography

[1] Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *Proc. ACM Program. Lang.*, 2(ICFP), jul 2018. doi:10.1145/3236789.

[2] H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*, page 117–309. Oxford University Press, Inc., 1993. ISBN: 0198537611.

[3] Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Log. J. IGPL*, 25(4):431–464, 2017.

[4] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33 (2):346–366, 1932. ISSN: 0003486X.

[5] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936. ISSN: 00029327, 10806377.

[6] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. ISSN: 00224812.

[7] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 10 1980. doi:10.1305/ndjfl/1093883253.

[8] Mario Coppo. An extended polymorphic type system for applicative languages. In Piotr Dembinski, editor, *Mathematical Foundations of Computer Science 1980 (MFCS'80), Proceedings of the 9th Symposium, Rydzyna, Poland, September 1-5, 1980*, volume 88 of *Lecture Notes in Computer Science*, pages 194–204. Springer, 1980.

[9] H. B. Curry. Functionality in combinatory logic*. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934. doi:10.1073/pnas.20.11.584.

[10] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

[11] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*,

POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery. ISBN: 0897910656. doi:10.1145/582153.582176.

[12] Ferruccio Damiani. Rank 2 intersection for recursive definitions. *Fundamenta Informaticae*, 77(4):451–488, 2007.

[13] Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul*. Phd thesis, Université Aix-Marseille II, 2007.

[14] E. Engeler. H. p. barendregt. the lambda calculus. its syntax and semantics. studies in logic and foundations of mathematics, vol. 103. north-holland publishing company, amsterdam, new york, and oxford, 1981, xiv 615 pp. *Journal of Symbolic Logic*, 49(1):301–303, 1984. doi:10.2307/2274112.

[15] Mário Florido and Luís Damas. Linearization of the lambda-calculus and its relation with intersection type systems. *J. Funct. Program.*, 14(5):519–546, 2004. doi:10.1017/S0956796803004970.

[16] Philippa Gardner. Discovering needed reductions using type theory. In *TACS*, volume 789 of *LNCS*, pages 555–574. Springer, 1994.

[17] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

[18] J. Roger Hindley. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1997. doi:10.1017/CBO9780511608865.

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

[20] A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–174. ACM, 1999.

[21] Assaf Kfoury. A linearization of the lambda-calculus and consequences. *Journal of Logic and Computation*, 10(3):411–436, 2000.

[22] S. C. Kleene and J. B. Rosser. The inconsistency of certain formal logics. *Annals of Mathematics*, 36(3):630–636, 1935. ISSN: 0003486X.

[23] Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 88–98, 1983.

[24] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978. ISSN: 0022-0000. doi:10.1016/0022-0000(78)90014-4.

[25] Fábio Reis, Sandra Alves, and Mário Florido. Linear rank intersection types. In *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, 2022.

[26] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1): 23–41, jan 1965. ISSN: 0004-5411. doi:10.1145/321250.321253.

[27] Steffen van Bakel. *Intersection type disciplines in lambda calculus and applicative term rewriting systems*. Phd thesis, Mathematisch Centrum, Katholieke Universiteit Nijmegen, 1993.

[28] Steffen van Bakel. Rank 2 intersection type assignment in term rewriting systems. *Fundam. Informaticae*, 26(2):141–166, 1996.

Appendix A

Haskell Implementation

A.1 Lambda Calculus

```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
```

A.2 Linear Types

```haskell
module LinearTypes where

import LambdaCalculus

-- Linear rank 0 intersection types (linear types).
data TLinearRank0 = TVar TyVar | TAp TLinearRank0 TLinearRank0
            deriving Eq

data TyVar = TyVar String
             deriving Eq

type EqSet = [(TLinearRank0, TLinearRank0)]

type Sub = (TyVar, TLinearRank0)

type Subst = [Sub]

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

instance Show TLinearRank0 where
    show (TVar a)    = show a
    show (TAp t1 t2) = ('(':show t1) ++ (' ':'-':'o':' ':show t2) ++ [')']

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


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

-- Given a Sub s=(a,t) and a linear type t0, replaces all free
-- occurrences of the type variable a in the type t0 with type t.
subst :: Sub -> TLinearRank0 -> TLinearRank0
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 linear type.
isFVType :: TyVar -> TLinearRank0 -> 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 with counting of quantitative information.
-- (The third element of the tuples is the counter of the quantitative information.)
unifyQ :: (EqSet, Unifier, Int) -> (EqSet, Unifier, Int)
unifyQ ([], u, count)                             = ([], u, count)
unifyQ ((t1, t2):ts, u, count) | t1 == t2        = unifyQ (ts, u, count)
unifyQ ((TAp t1 t2, TAp t1' t2'):ts, u, count) = unifyQ ((t1, t1'):(t2, t2'):ts, u, count+1)
```

```
unifyQ ((TAp t1 t2, TVar a):ts, u, count)       = unifyQ ((TVar a, TAp t1 t2):ts, u, count)
unifyQ ((TVar a, t):ts, Uni s, count)
      | isFVType a t                       = error ("FAIL - trying to unify: " ++ 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 unifyQ (ts', Uni ((a, t):s'), count)
      | otherwise                          = unifyQ (ts, Uni ((a, t):s), count)
```

A.3 Linear Rank 2 Quantitative Types

```haskell
module LinearRank2QuantitativeTypes where

import LambdaCalculus
import LinearTypes

-- Linear rank 1 intersection types.
data TLinearRank1 = T1_0 TLinearRank0 | Inters TLinearRank1 TLinearRank1
             deriving Eq

-- Linear rank 2 intersection types.
data TLinearRank2 = T2_0 TLinearRank0 | T2ApL TLinearRank0 TLinearRank2 | T2Ap TLinearRank1
    ↪ TLinearRank2
             deriving Eq

-- Environments.
type LEnv = [(TeVar, TLinearRank1)]

instance Show TLinearRank1 where
    show (T1_0 t0)       = show t0
    show (Inters t1 t2) = ('(':show t1) ++ ('/':'\\':show t2) ++ [')']

instance Show TLinearRank2 where
    show (T2_0 t)       = show t
    show (T2ApL t0 t2) = ('(':show t0) ++ (' ':'-':'o':' ':show t2) ++ [')']
    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, ...).


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

-- Given a list of linear rank 1 intersection types, returns a single type
-- consisting of the intersection of the types in the given list.
listToInters :: [TLinearRank1] -> TLinearRank1
listToInters [t1]     = t1
listToInters (t1:ts) = Inters (listToInters ts) t1

-- Checks if a linear rank 1 type is also a linear rank 0 type (i.e., if it does not have
    ↪ intersections).
t1isT0 :: TLinearRank1 -> Bool
t1isT0 (T1_0 _) = True
t1isT0 _         = False

-- Converts a linear rank 1 type t to a linear rank 0 type, if t is also a linear rank 0 type
    ↪ (i.e., if
-- it does not have intersections); otherwise, fails.
t1toT0 :: TLinearRank1 -> TLinearRank0
t1toT0 (T1_0 t0) = t0
t1toT0 t1         = error ("FAIL - t1toT0 error: the type " ++ show t1 ++ " is not a linear rank 0
    ↪ type.\n")

-- Converts a linear rank 2 type t to a linear rank 0 type, if t is also a linear rank 0 type
    ↪ (i.e., if
-- it does not have any intersections); otherwise, fails.
t2toT0 :: TLinearRank2 -> TLinearRank0
t2toT0 (T2_0 t0)      = t0
t2toT0 (T2ApL t0 t2) = TAp t0 (t2toT0 t2)
t2toT0 (T2Ap t1 t2)  = error ("FAIL - t2toT0 error: the type " ++ show (T2Ap t1 t2) ++ " is not a
    ↪ linear rank 0 type.\n")

-- Given a Sub s=(a,t) and a linear rank 1 intersection type t1, replaces all free
-- occurrences of the type variable a in the type t1 with type t.
subst1 :: Sub -> TLinearRank1 -> TLinearRank1
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 linear rank 2 intersection type t2, replaces all free
-- occurrences of the type variable a in the type t2 with type t.
subst2 :: Sub -> TLinearRank2 -> TLinearRank2
subst2 s (T2_0 t0)      = T2_0 (subst s t0)
subst2 s (T2ApL t0 t2) = T2ApL (subst s t0) (subst2 s t2)
subst2 s (T2Ap t1 t2)  = T2Ap (subst1 s t1) (subst2 s t2)

-- Applies a substitution to a linear rank 2 intersection type.
```

```
substty2 :: Subst -> TLinearRank2 -> TLinearRank2
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 -> LEnv -> LEnv
substEn _ []         = []
substEn s ((x,t):es) = (x, subst1 s t):substEn s es

-- Applies a substitution to an environment.
substEnv :: Subst -> LEnv -> LEnv
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 -> LEnv -> 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 -> LEnv -> [TLinearRank1]
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 -> LEnv -> TLinearRank1
findInEnv x1 ((x2, t):es) | x1 == x2  = t
                          | otherwise = findInEnv x1 es

-- Removes all occurrences of a term variable from an environment.
rmFromEnv :: TeVar -> LEnv -> LEnv
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 :: LEnv -> LEnv
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 linear types of the given linear rank 1 sequence, and returns the environment
-- and the generated equations described in the algorithm.
-- (The third element of the returning tuple is the counter of the quantitative information.)
genEqs :: TLinearRank1 -> Term -> Int -> (LEnv, EqSet, Int, Int)
genEqs (T1_0 tau) m n0             = (env, [(t2toT0 t, tau)], b, n1) -- fails if M has a linear
    ↪ rank 2 type
                                   where (env, t, b, n1) = quantR2typeInf m n0
genEqs (Inters tseq1 tseq2) m n0 = (mergeEnv (envs1++envs2), eqs1++eqs2, bs1+bs2, n2)
                                   where (envs1, eqs1, bs1, n1) = genEqs tseq1 m n0
                                         (envs2, eqs2, bs2, n2) = genEqs tseq2 m n1

-- Type inference algorithm for linear 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 -> (LEnv, TLinearRank2, Int, Int)
quantR2typeInf (Var x) n0      = let a = TVar (TyVar ('a':(show n0))) in
                                 ([(x, T1_0 a)], T2_0 a, 0, n0+1)
                                     ↪
                                     ↪ -- Rule 1.

quantR2typeInf (Abs x m1) n0  = let (env1, sig1, b1, n1) = quantR2typeInf m1 n0 in
                                 if (isInEnv x env1)
                                 then let t1    = findInEnv x env1
                                          env1x = rmFromEnv x env1
                                      in if (t1isT0 t1)
                                         then (env1x, T2ApL (t1toT0 t1) sig1, b1, n1)
                                              ↪
                                              ↪ -- Rule 2.b
                                         else (env1x, T2Ap t1 sig1, b1, n1)
                                              ↪
                                              ↪ -- Rule 2.c
```

```haskell
                                          else error ("FAIL - Rule 2.(a): " ++ show x ++ " not in " ++
                                              ↪ show env1 ++ "\n")                              -- Rule 2.a

quantR2typeInf (App m1 m2) n0 = let (env1, sig1, b1, n1) = quantR2typeInf m1 n0 in
                                    case sig1 of
                                        T2_0 (TVar a1)      -> (substEnv s env, substty2 s (T2_0
                                            ↪ a3), b1+b2, n2+2)                                -- Rule
                                            ↪ 3.a
                                                        where (env2, tau2, b2, n2) =
                                                            ↪ quantR2typeInf m2 n1
                                                              env                     = mergeEnv
                                                                  ↪ (env1++env2)
                                                              a2                      = TVar
                                                                  ↪ (TyVar ('a':(show n2)))
                                                              a3                      = TVar
                                                                  ↪ (TyVar ('a':(show (n2+1))))
                                                              eqs                     = [(TVar a1,
                                                                  ↪ TAp a2 a3), (t2toT0 tau2,
                                                                  ↪ a2)]
                                                              ([], Uni s, _)          = unifyQ
                                                                  ↪ (eqs, Uni [], 0)

                                        T2Ap tseq sig1'     -> (substEnv s env, substty2 s sig1',
                                            ↪ b1+bs+b3+1, n2)                                  -- Rule 3.b
                                                        where (envs, eqs, bs, n2)  = genEqs
                                                            ↪ tseq m2 n1
                                                              env                     = mergeEnv
                                                                  ↪ (env1++envs)
                                                              ([], Uni s, b3)         = unifyQ
                                                                  ↪ (eqs, Uni [], 0)

                                        T2ApL tau sig       -> (substEnv s env, substty2 s sig,
                                            ↪ b1+b2+b3+1, n2)                                  -- Rule
                                            ↪ 3.c
                                                        where (env2, tau2, b2, n2) =
                                                            ↪ quantR2typeInf m2 n1
                                                              env                     = mergeEnv
                                                                  ↪ (env1++env2)
                                                              eqs                     = [(t2toT0
                                                                  ↪ tau2, tau)]
                                                              ([], Uni s, b3)         = unifyQ
                                                                  ↪ (eqs, Uni [], 0)

                                        T2_0 (TAp tau sig) -> (substEnv s env, substty2 s (T2_0
                                            ↪ sig), b1+b2+b3+1, n2)                            -- Rule
                                            ↪ 3.c
                                                        where (env2, tau2, b2, n2) =
                                                            ↪ quantR2typeInf m2 n1
                                                              env                     = mergeEnv
                                                                  ↪ (env1++env2)
                                                              eqs                     = [(t2toT0
                                                                  ↪ tau2, tau)]
                                                              ([], Uni s, b3)         = unifyQ
                                                                  ↪ (eqs, Uni [], 0)
```

A.4 Reductions

```haskell
module Reductions where

import LambdaCalculus
import Data.List

type Sub = (TeVar, Term)

removeAll :: Eq a => a -> [a] -> [a]
removeAll x = filter (/= x)

intersection :: Eq a => [a] -> [a] -> [a]
intersection l1 l2 = nub (intersect l1 l2)

inBetaNF :: Term -> Bool
inBetaNF (Var _)             = True
inBetaNF (App (Abs _ _) _) = False
inBetaNF (Abs _ m)           = inBetaNF m
inBetaNF (App m1 m2)         = inBetaNF m1 && inBetaNF m2

renameFree1 :: Term -> TeVar -> Term
renameFree1 (Var (TeVar x)) y | (TeVar x) == y = Var (TeVar (x++"'"))
                              | otherwise      = Var (TeVar x)
renameFree1 (Abs x m) y | x == y               = Abs x m
                        | otherwise            = Abs x (renameFree1 m y)
renameFree1 (App m1 m2) y                      = App (renameFree1 m1 y) (renameFree1 m2 y)

renameBV :: Term -> [TeVar] -> Term
renameBV (Var x) _                                = Var x
renameBV (Abs (TeVar x) m) xs | elem (TeVar x) xs = Abs (TeVar (x++"'")) (renameBV (renameFree1 m
    ↪ (TeVar x)) xs)
                              | otherwise         = Abs (TeVar x) (renameBV m xs)
renameBV (App m1 m2) xs                           = App (renameBV m1 xs) (renameBV m2 xs)

substitute :: Sub -> Term -> Term
substitute (TeVar x1, m1) m2 | length (inter) > 0 = substitute (TeVar x1, m1) m2'
                             where inter = intersection (boundVars m2) (freeVars m1)
                                   m2'   = renameBV m2 inter
substitute (x1, m) (Var x2) | x1 == x2            = m
                            | otherwise           = Var x2
substitute (x1, m1) (Abs x2 m2) | x1 == x2        = Abs x2 m2
                                | otherwise       = Abs x2 (substitute (x1, m1) m2)
substitute s (App m1 m2)                          = App (substitute s m1) (substitute s m2)

boundVars :: Term -> [TeVar]
boundVars (Var _)     = []
boundVars (Abs x m)   = x:boundVars m
boundVars (App m1 m2) = boundVars m1 ++ boundVars m2

freeVars :: Term -> [TeVar]
freeVars (Var x)     = [x]
freeVars (Abs x m)   = removeAll x (freeVars m)
freeVars (App m1 m2) = freeVars m1 ++ freeVars m2

isFV :: TeVar -> Term -> Bool
isFV x t = elem x (freeVars t)


-------Maximal Beta-reduction Strategy (based on Def. 3.21 (Cap. 3.5) from 'Perpetual Reductions
    ↪ in Lambda-Calculus'-------

isXParrow :: Term -> Bool
isXParrow (Var x)     = True
isXParrow (App m1 m2) = inBetaNF m2 && isXParrow m1
isXParrow _           = False

maximal :: Term -> (Term, Int)
maximal m | inBetaNF m                        = (m, 0)
maximal (Abs x m)                             = let (m', n) = maximal m
                                                in (Abs x m', n)
maximal (App (Abs x m1) m2)
            | isFV x m1 || inBetaNF m2        = (substitute (x, m2) m1, 1)
            | otherwise                       = let (m2', n) = maximal m2
                                                in (App (Abs x m1) m2', n)
maximal (App m1 m2) -- Rule 1
            | not (inBetaNF m2) && isXParrow m1 = let (m2', n) = maximal m2
                                                  in (App m1 m2', n)
```

```haskell
maximal (App m1 m2)                                = let (m1', n) = maximal m1
                                                     in (App m1' m2, n)


-------Normal Order Beta-reduction Strategy-------

normal :: Term -> (Term, Int)
normal m | inBetaNF m                = (m, 0)
normal (Abs x m)                     = let (m', n) = normal m in (Abs x m', n)
normal (App (Abs x m1) m2)           = (substitute (x, m2) m1, 1)
normal (App m1 m2) | not (inBetaNF m1) = let (m1', n) = normal m1 in (App m1' m2, n)
                   | otherwise         = let (m2', n) = normal m2 in (App m1 m2', n)


-------Applicative Order Beta-reduction Strategy-------

applicative :: Term -> (Term, Int)
applicative m | inBetaNF m                              = (m, 0)
applicative (Abs x m)                                   = let (m', n) = applicative m in (Abs x m', n)
applicative (App (Abs x m1) m2) | not (inBetaNF m2) = let (m2', n) = applicative m2 in (App (Abs
    ↪ x m1) m2', n)
                                | not (inBetaNF m1) = let (m1', n) = applicative m1 in (App (Abs
                                    ↪ x m1') m2, n)
                                | otherwise         = (substitute (x, m2) m1, 1)
applicative (App m1 m2) | not (inBetaNF m1)         = let (m1', n) = applicative m1 in (App m1'
    ↪ m2, n)
                        | otherwise                 = let (m2', n) = applicative m2 in (App m1
                            ↪ m2', n)


maxSteps :: Int
maxSteps = 1000

reduce :: (Term -> (Term, Int)) -> [Term] -> Int -> ([Term], Int)
reduce strat (m:ms) n0 | n0 > maxSteps = (((Var (TeVar ("Limit Exceeded. Current term: " ++ show
    ↪ m))):ms), n0)
                       | inBetaNF m    = ((m:ms), n0)
                       | otherwise     = reduce strat (m':m:ms) (n0+n1)
                     where (m', n1) = strat m
```

A.5 Parser

```
{
module Main where

import Data.Char
import LambdaCalculus
import Reductions
import SimpleTypes
import Rank2IntersectionTypes
import LinearTypes
import LinearRank2QuantitativeTypes
}

%name parse
%tokentype { Token }
%error { parseError }

%token
  '\\'          { TokenLambda     }
  '.'           { TokenPoint      }
  ' '           { TokenSpace      }
  var           { TokenVar $$     }
  '('           { TokenOB         }
  ')'           { TokenCB         }
  typeinf0      { TokenInf0       }
  typeinf2      { TokenInf2       }
  qtypeinf2     { TokenQInf2      }
  reduceMax     { TokenReduceMax  }
  reduceNorm    { TokenReduceNorm }
  reduceApp     { TokenReduceApp  }
  steps         { TokenSteps      }
  count         { TokenCount      }

%left '.'
%left ' '

%%

Exp    : TyInf                                  { TyInf $1     }
       | Term                                   { Term $1      }
       | Reduction                              { Reduction $1 }

Term   : var                                    { Var (TeVar $1) }
       | Abs                                    { $1             }
       | App                                    { $1             }
       | '(' Term ')'                           { $2             }

Abs    : '\\' var '.' Term                      { Abs (TeVar $2) $4 }

App    : Term ' ' Term                          { App $1 $3 }

TyInf : typeinf0 ' ' '(' Term ')'               { TyInf0 $4 (simpleTypeInf $4 0)            }
       | typeinf2 ' ' '(' Term ')'              { TyInf2 $4 (r2typeInf $4 0)                }
       | qtypeinf2 ' ' '(' Term ')'             { QTyInf2 $4 (quantR2typeInf $4 0) Default }
       | qtypeinf2 ' ' '(' Term ')' ' ' count   { QTyInf2 $4 (quantR2typeInf $4 0) Count   }

Reduction : reduceMax ' ' '(' Term ')'          { Reduct "maximal strategy" $4 (reduce maximal
    ↪ [$4] 0) Default         }
          | reduceNorm ' ' '(' Term ')'         { Reduct "normal strategy" $4 (reduce normal
              ↪ [$4] 0) Default           }
          | reduceApp ' ' '(' Term ')'          { Reduct "applicative strategy" $4 (reduce
              ↪ applicative [$4] 0) Default }
          | reduceMax ' ' '(' Term ')' ' ' steps   { Reduct "maximal strategy" $4 (reduce maximal
              ↪ [$4] 0) Steps           }
          | reduceNorm ' ' '(' Term ')' ' ' steps  { Reduct "normal strategy" $4 (reduce normal
              ↪ [$4] 0) Steps             }
          | reduceApp ' ' '(' Term ')' ' ' steps   { Reduct "applicative strategy" $4 (reduce
              ↪ applicative [$4] 0) Steps   }
          | reduceMax ' ' '(' Term ')' ' ' count   { Reduct "maximal strategy" $4 (reduce maximal
              ↪ [$4] 0) Count           }
          | reduceNorm ' ' '(' Term ')' ' ' count  { Reduct "normal strategy" $4 (reduce normal
              ↪ [$4] 0) Count             }
          | reduceApp ' ' '(' Term ')' ' ' count   { Reduct "applicative strategy" $4 (reduce
              ↪ applicative [$4] 0) Count   }

{
parseError :: [Token] -> a
```

```haskell
parseError _ = error "Parse error"

data Exp
      = TyInf TyInf
      | Reduction Reduction
      | Term Term

data TyInf
      = TyInf0 Term (Basis, Type0, Int)
      | TyInf2 Term (Env, Type2, Int)
      | QTyInf2 Term (LEnv, TLinearRank2, Int ,Int) Mode

data Reduction  -- Reduct Reduction_strategy Term Initial_term (Reverse_list_of_reductions,
    ↪ Number_reductions) Mode_of_printing
      = Reduct String Term ([Term], Int) Mode

data Mode
      = Default  -- shows everything (except reduction steps, in the case of Reduct)
      | Count    -- only shows the counters
      | Steps    -- (only for Reduct) shows everything, with reduction steps

instance Show Exp where
    show (TyInf x)     = show x
    show (Reduction x) = show x
    show (Term x)      = "Term: " ++ show x ++ ['\n']

instance Show TyInf where
    show (TyInf0 term (basis, t0, _))            = "\t[--- Inference (simple types) ---]" ++
        ↪ "\n\tTerm        = " ++ show term ++ "\n\tBasis       = " ++ show basis ++ "\n\tType
        ↪         = " ++ show t0 ++ ['\n']
    show (TyInf2 term (env, t0, _))              = "\t[--- Inference (rank 2 intersection types)
        ↪ ---]" ++ "\n\tTerm        = " ++ show term ++ "\n\tEnvironment = " ++ show env ++
        ↪ "\n\tType        = " ++ show t0 ++ ['\n']
    show (QTyInf2 term (env, t2, c, _) Default) = "\t[--- Inference (linear rank 2 quantitative
        ↪ types) ---]" ++ "\n\tTerm        = " ++ show term ++ "\n\tEnvironment = " ++ show env
        ↪ ++ "\n\tType        = " ++ show t2 ++ "\n\tCount       = " ++ show c ++ ['\n']
    show (QTyInf2 term (env, t2, c, _) Count)   = "\t[--- Inference (linear rank 2 quantitative
        ↪ types) ---]" ++ "\n\tTerm        = " ++ show term ++ "\n\tCount       = " ++ show c ++
        ↪ ['\n']

instance Show Reduction where
    show (Reduct strat term (terms, c) Default) = "\t[--- Reduction (" ++ strat ++ ") ---]" ++
        ↪ "\n\tTerm        = " ++ show term ++ "\n\tNormal form = " ++ show (head terms) ++
        ↪ "\n\tCount       = " ++ show c ++ ['\n']
    show (Reduct strat term (terms, c) Steps)   = "\t[--- Reduction (" ++ strat ++ ") ---]" ++
        ↪ "\n\tTerm        = " ++ show term ++ "\n\tNormal form = " ++ show (head terms) ++
        ↪ "\n\tCount       = " ++ show c ++ "\n\tReductions:    " ++ show (head (reverse terms))
        ↪ ++ "\n" ++ concat (map (\x -> "\t              -> " ++ show x ++ "\n") (tail (reverse
        ↪ terms)))
    show (Reduct strat term (terms, c) Count)   = "\t[--- Reduction (" ++ strat ++ ") ---]" ++
        ↪ "\n\tTerm        = " ++ show term ++ "\n\tCount       = " ++ show c ++ ['\n']

data Token
      = TokenLambda
      | TokenPoint
      | TokenSpace
      | TokenVar String
      | TokenOB
      | TokenCB
      | TokenInf0
      | TokenInf2
      | TokenQInf2
      | TokenReduceMax
      | TokenReduceNorm
      | TokenReduceApp
      | TokenSteps
      | TokenCount
      deriving Show

lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
 | isAlphaNum c = lexVar (c:cs)
lexer ('\\':cs) = TokenLambda : lexer cs
lexer ('.':cs)  = TokenPoint : lexer cs
lexer (' ':cs)  = TokenSpace : lexer cs
lexer ('(':cs)  = TokenOB : lexer cs
lexer (')':cs)  = TokenCB : lexer cs

lexVar cs =
```

```
    case span isAlphaNum cs of
        ("ti0",rest)          -> TokenInf0 : lexer rest
        ("ti2",rest)          -> TokenInf2 : lexer rest
        ("qti2",rest)         -> TokenQInf2 : lexer rest
        ("reduceMax",rest)    -> TokenReduceMax : lexer rest
        ("reduceNorm",rest)   -> TokenReduceNorm : lexer rest
        ("reduceApp",rest)    -> TokenReduceApp : lexer rest
        ("steps",rest)        -> TokenSteps : lexer rest
        ("count",rest)        -> TokenCount : lexer rest
        (var,rest)            -> TokenVar var : lexer rest

main = do line <- getLine
          let action | all isSpace line || line!!0 == '' = main| line!!0 == '-' = putStrLn (tail line)| otherwise =
              (print . parse . lexer) lineactionmain
```