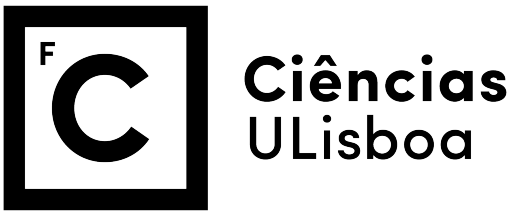
UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA



INTEGRATING TYPE OPERATORS INTO THE FREEST PROGRAMMING LANGUAGE

Paula Inês Garcias Lopes

Mestrado em Engenharia Informática
Especialização em Designação da Especialização / Perfil, se aplicável

Versão Provisória

Dissertação orientada por:
Profª. Doutora Vasco T. Vasconcelos
Prof. Doutor Diana Costa

2024

Agradecimentos

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus legentis in iis qui facit eorum claritatem. Investigationes demonstraverunt lectores legere me lius quod ii legunt saepius. Claritas est etiam processus dynamicus, qui sequitur mutationem consuetudium lectorum. Mirum est notare quam littera gothica, quam nunc putamus parum claram, anteposuerit litterarum formas humanitatis per seacula quarta decima et quinta decima. Eodem modo typi, qui nunc nobis videntur parum clari, fiant sollemnes in futurum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus legentis in iis qui facit eorum claritatem. Investigationes demonstraverunt lectores legere me lius quod ii legunt saepius.

Dedicatória.

Resumo

Os documentos escritos em Português devem ter um resumo em Português e um resumo noutra língua comunitária que contenham até 300 palavras cada. Quando o conselho científico autorizar a apresentação do trabalho final escrito em língua estrangeira, este deve ser acompanhado de um resumo adicional em Português de, pelo menos, 1200 palavras.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus legentis in iis qui facit eorum claritatem. Investigationes demonstraverunt lectores legere me lius quod ii legunt saepius. Claritas est etiam processus dynamicus, qui sequitur mutationem consuetudium lectorum. Mirum est notare quam littera gothica, quam nunc putamus parum claram, anteposuerit litterarum formas humanitatis per seacula quarta decima et quinta decima. Eodem modo typi, qui nunc nobis videntur parum clari, fiant sollemnes in futurum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum. Typi non habent claritatem insitam; est usus legentis in iis qui facit eorum claritatem. Investigationes demonstraverunt lectores legere me lius quod ii legunt saepius.

Palavras-chave: cerca de 5 palavras-chave

Abstract

Context-free session types emerged from the need to expand session type theory to non-regular protocols. Their inclusion in type systems enhances the expressiveness and adaptability of communication protocols in programming languages, yet pose a significant challenge for type equivalence algorithms. In this work, we study System $F_{\omega}^{\mu *;}$, the higher-order polymorphic lambda calculus equipped with equirecursive and context-free session types, as well as its seamless integration into FreeST, a functional programming language governed by context-free session types. We follow a bisimulation-based approach to type equivalence, translating types into simple grammars and verifying the bisimilarity of these grammars, a problem that is decidable.

Keywords: Higher-order Kinds, Context-Free Session Types, Session Polymorphism, Type Equivalence, Simple Grammar. about 5 keywords (in English)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

O relatório final deverá ter, em geral, entre 50 e 90 páginas (sem considerar anexos). O seu conteúdo deve realçar o trabalho realizado pelo aluno e a sua contribuição concreta no trabalho. Por exemplo, se o trabalho consiste no desenvolvimento de vários módulos a serem integrados num sistema mais global, o aluno deverá preocupar-se em descrever a parte que desenvolveu, como desenvolveu, que ferramentas usou, que alternativas poderiam existir, etc., em vez de efectuar uma descrição exaustiva das funcionalidades de todo o sistema.

O número de capítulos no relatório final não é rígido. No entanto, recomenda-se que sejam adoptados os seguintes princípios para a organização do relatório:

1. Um capítulo *introdutório* no qual se apresentam o contexto do trabalho, se resume o trabalho desenvolvido, se identificam as contribuições deste e se apresenta a estrutura do próprio relatório. Deverá também ser mencionado sucintamente o enquadramento institucional em que o trabalho decorreu.

2. Um capítulo no qual se apresentam *em pormenor* os *objectivos* do trabalho, o *contexto subjacente*, a *metodologia* utilizada no seu desenvolvimento bem como o *planeamento* efectuado para o concretizar. Deve também ser apresentada uma confrontação com o plano de trabalho inicial analisando as razões de eventuais desvios ocorridos.

3. Um capítulo onde é descrito o *trabalho realizado*. Este é um dos capítulos fundamentais do relatório. Apresenta concretamente o que se fez de facto e as ferramentas usadas. De notar que é importante que fique claro qual a contribuição concreta do trabalho, sobretudo em casos de trabalho em equipa. Neste capítulo poderão ser inseridas questões relevantes da área de estudo em que o trabalho se integra, assim como o possível enquadramento num trabalho mais amplo. Eventualmente, e em função do âmbito e dimensão do trabalho, este capítulo poderá ser substituído por um conjunto de outros capítulos, que englobem em si o *trabalho relacionado*, a *análise* do problema, o *desenho* da solução, a *implementação* da solução e a *avaliação* desta.

4. Um capítulo no qual são apresentadas as *conclusões*. Para além de um *sumário* do trabalho realizado, deve ser feito um *comentário crítico* e serem apresentadas possibilidades

de *trabalho futuro* referindo o que falta fazer e o que poderá ser melhorado.

5. Um capítulo com a *bibliografia* - lista de documentos usados e outras referências consideradas relevantes.

6. Um conjunto de capítulos com os *anexos*. Quaisquer listagens, informação confidencial ou outras descrições muito pormenorizadas não devem ser integradas no corpo principal do relatório. Se houver necessidade de as apresentar, sugere-se a sua introdução em anexos ou em documentos separados. Os anexos suplementam o relatório e como tal devem ser referidos no corpo principal do relatório, descrevendo o tipo de informação que se detalha em anexo.

Quando concluir a sua Dissertação, ou Relatório Final, o aluno deverá entregar, no Gabinete de Estudos Pós-Graduados da FCUL, o seguinte:

- Requerimento de admissão a provas de Mestrado (ver secção 4.2 do Guia de PEI);

- 7 exemplares da Dissertação, ou Relatório Final, (encadernados de forma a que seja possível escrever na lombada - não utilizar argolas);

- 7 Curricula Vitae;

- 3 CDs com a Dissertação, ou Relatório Final, gravado em formato PDF;

- Parecer do orientador do DI sobre a Dissertação, ou Relatório Final, (ver secção 4.2 do Guia de PEI).

O aluno deverá também submeter a versão final, em formato PDF, da Dissertação, ou Relatório Final, através do PEIpal. Este documento não deverá, em condições normais, exceder os 5MiB (se isso acontecer então deve ser revista a qualidade das imagens evitando a inclusão de bitmaps).

Se houver fundamentação adequada, os relatórios de trabalho poderão ser escritos em Inglês. Para isso o aluno deverá entregar no Gabinete de Estudos Pós-Graduados da FCUL:

- Um pedido dirigido ao Presidente do Conselho Científico da FCUL, fundamentando a necessidade da escrita do relatório em Inglês (ver secção 4.2 do Guia de PEI);

- Um parecer do orientador indicando que concorda com o pedido do aluno e, eventualmente, apresentando argumentos adicionais (ver secção 4.2 do Guia de PEI).

Tendo sido aceite a escrita em Inglês do relatório de trabalho, este deverá conter um resumo adicional em Português de, pelo menos, 1200 palavras.

1.1 Background and Motivation

Exploring sophisticated type systems and their seamless integration into programming languages is a thoroughly researched field. From System F^{μ} [**?**] up to System F^{μ}_{ω} [**?**], how far can we go until these systems are no longer suitable for compilers.

In the development of modern type systems, combining advanced features such as equirecursion, higher-order polymorphism, and higher-order context-free session types presents unique challenges and opportunities. The primary motivation for our research is to integrate these elements into a cohesive type system that can be practically incorporated into programming languages. Therefore, we are interested in practical algorithms for type equivalence checking to be incorporated into compilers.

1.2 Objectives

1.3 Challenges

1.4 Contributions

1.5 Thesis Structure

Este documento está organizado da seguinte forma:

- Capítulo 2 – AAA
- Capítulo 3 – BBB

Chapter 2

Background and Related Work

2.1 Type Systems

Type systems are a fundamental tool in programming language theory, providing a framework to categorize and constrain the behavior of programs, while types define the kind of data a program manipulates, e.g., integers, booleans, strings.

Type systems manage a program's data types through type checking, which can be classified as either static—occurs at compile time, where the types of all variables and expressions are known and checked before the program runs—or dynamic—occurs at run-time, where the types are checked as the program executes.

Over the course of time, type systems have evolved to become more expressive, moving from simple type assignments to more powerful systems that support polymorphism, recursion, and structured communication patterns.

System F, also called the polymorphic lambda calculus, extends the simply typed lambda calculus with *parametric polymorphism*, which allows functions to be written generically and applied to arguments of any type. [ref] In the simply typed lambda calculus, every function has a concrete type. For instance, a function that operates on integers has the type $Int \to Int$. However, in order to express a function that operates on a lists of arbitrary types, we would need a separate function for each of the types in the list. To overcome this limitation, System F allowed functions to be polymorphic. For example, the identity function, written $\lambda X.\lambda x : X.x$, returns its argument unchanged, working uniformly for any type—when instantiated to Int, the result is the identity function on integers, when applied to $Bool$, the result is the identity function on booleans. This is possible through the introduction of type *abstractions*—functions are parameterized by types—and type *applications*—types are passed as arguments to functions.

While System F is a powerful system for expressing parametric polymorphism, it lacks a direct way to express recursive types, which are essential for defining more complex structures and functions that rely on self-reference. Recursive types are particularly important for defining data structures like lists, trees, and infinitely repeating processes.

System F^{μ} comes as an extension of System F with the addition of *recursion*, denoted by $\mu X.T$, which allows a type T to refer to itself through the variable X.

[TODO: Simple example of a list of integers]

By allowing recursion at the type level, System F^{μ} significantly increases the expressiveness of the type system, making it capable of representing potentially infinite structures. However, now care must be taken to ensure that recursive functions terminate, which leads to type systems often incorporating additional rules or constraints to ensure *termination*. Without guarantees of termination, recursive types could lead to non-terminating computations, i.e., infinite recursions, which can make programs unreliable.

[TODO: note about grammar, pushdown automata, corresponding to the system.]

System $F^{\mu;}$ builds on top of System F^{μ} by incorporating *session types*into the type system. This extension is designed to model structured communication between concurrent processes, ensuring that communication follows a well-defined protocol.

2.2 Session Types

Session types offer a formal way to describe communication protocols. They ensure that all participants in a communication process adhere to the agreed protocol, improving both safety and expressiveness. This concept emerged in the early 1990s from the fields of process calculi and type theory, particularly building upon Milner's work on the pi-calculus.[ref] Its main purpose was to bring type-level guarantees to communication protocols.

In a programming language, *types* describe the structure of data (e.g., integers, strings) and session types extend this idea to describe communication patterns. They define the sequence and type of messages exchanged during an interaction.

[TODO: introduce syntax, choice, recursion.]

[TODO: simple example.]

Communication involves two parties—a sender and a receiver—and for a protocol to function correctly, the session types of these parties must be dual. *Duality* is the relationship between two session types that allows them to correctly engage in a protocol. For any session type describing one side of the communication (let us say the client), there is a dual session type that describes the other side (the server). For example, the session type $!\mathsf{Int}$ describes sending an integer and its dual session type is to receive an integer, $?\mathsf{Int}$.

One of the key advantages of session types is that they enable *type-safe communication*. This means that at compile time, the system can check whether the communication patterns follow the prescribed protocol, preventing errors like sending a message of a wrong type. Session types can also help ensure *deadlock freedom*, meaning the system avoids situations where two parties wait indefinitely for each other to send or receive a message. This is particularly important in distributed and concurrent systems.

Session types also have limitations—they are tail recursive—not being able to describe expressive types that refer to themselves at any given point of a sequential composition rather than at the end. The lack of expressiveness with the increasing complexity of communication protocols in modern systems lead us to context-free session types.

[TODO: mention fmu and fmu. can be represented by finite-state automata somewhere.]

2.3 Context-free Session Types

Context-free session types break free from tail recursion by introducing the sequential composition operator ; and the type Skip—neutral element of sequential composition—which represents a protocol with no actions.

[TODO: since cfst resemble cfgrammars, they are as expressive as simple grammar.]

The recursive type $\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha;\ ?\mathsf{Int};\ \alpha\};\mathsf{End}_?$ describes a protocol for safely streaming integer trees on a channel. The channel presents an external choice $\&$ with two labels: if the continuation of the channel is Leaf, then no communication occurs but the channel is still open for further composition whereas, if the continuation of the channel is Node, then we have a left subtree, followed by an integer and a right subtree. When the whole tree is received, the channel is closed. It is also important to distinguish type $\mathsf{End}_?$ from Skip—the former represents the closure of a channel, where no further communication is possible, while the latter allows continuing the communication. We want to move beyond context-free session types, namely, we are interested in abstracting the type that is received on the tree channel, by writing $\lambda\beta\colon \mathrm{T}.\mu\,\alpha\colon \mathrm{S}.\,\&\{\mathsf{Leaf}\colon \mathsf{Skip}, \mathsf{Node}\colon \alpha;\ ?\beta;\ \alpha\};\mathsf{Wait}$. Abstractions introduce higher-order kinds which lead to the introduction of type operators into our language.

2.4 FreeST

[TODO: what is FreeST]

FreeST [?], a concurrent functional programming language based on system $F^{\mu;}$, is regulated by context-free session types. Context-free session types, unlike regular session types, are not restricted to tail recursion, thus allowing the description of more sophisticated communication protocols.

[TODO: small freest program example. Could be streaming a tree]

FreeST's current type equivalence algorithm, developed by Almeida et al. [?] decides the equivalence of context-free session types by reducing the problem to the bisimilarity of simple grammars. The next step is to extend the language to a higher-order setting where type equivalence is still decidable.

[TODO: note about equirecursion somewhere]

2.5 Type equivalence algorithms

2.6 Programming languages with type operators

[TODO: intro.]

A type system is described as higher-order if it supports *type-level functions*, i.e., if it acts upon types and produce even more complex types. Such systems are often realized through mechanisms such as type classes in Haskell or implicits in Scala.[ref]

```
data List a = Empty | Cons a (List a)
```

This Haskell code represents a `List`—a type constructor that takes a type `a` and returns a new type `List a`—which represents a list of elements of type `a`. Therefore, the type `List` operates on types, that means, when type `a` is replaced by another type (let us say `Int`) the constructor returns `List Int`.

[TODO: introduce kinds briefly.] While type constructors take (concrete) types as inputs, higher-order kinded types operate one level further. They are type operators which take type constructors themselves as inputs.

A *functor*, a type class where types can be mapped over, would be an example of a higher-order kinded type.

```
class Functor f where fmap :: (a →b)→f a →f b
```

In this example, the method `fmap` takes two arguments, a function and a functor type `f a`, and returns a new type `f b` from mapping the first argument to each value in `f a`.

Type classes, in Haskell, enable polymorphism by defining a set of operations that can be implemented for various types. A *type class* is a higher-order abstraction that operates on types rather than values.

```
class Eq a where (==):: a →a →Bool
```

The example above represents `Eq`, a type class that defines equality for any type `a`. Type classes emerged from the need to deal with ad-hoc polymorphism, also known as overloading, providing a flexible structure to manage polymorphic operations. *Ad-hoc polymorphism* refers to the ability of a function or operator to behave differently based on the type of its arguments, that means a function can have different implementations based on the types of its parameters and the compiler will choose the correct implementation based on the types of arguments passed. **[TODO:** general example of + operator.] While, *Parametric polymorphism* refers to the ability of a function or a data type to be written generically so that it can handle values of any type, without type-specific implementations.

[TODO: comparison between oCaml and Haskell.]

Scala's implicits notion have a similar function to haskell's type classes, where the compiler ensures the correctness of these operations based on resolving these implicit types. The code provided below is the equivalent type class `Eq` implemented in Scala: **[TODO:** scala code.]

[TODO: Benefits of higher-order systems and type operators.]

[TODO: mention the higher-order version of the sytems are as expressive as pushdown automata.]

[TODO: mention limitation of myu kind possibly]

Chapter 3

System F

A infinita diversidade da realidade única reduziria a importância das condições epistemológicas e cognitivas exigidas. Se, todavia, a complexidade dos estudos efetuados potencializa a influência da fundamentação metafísica das representações. Assim mesmo, a estrutura atual da ideação semântica exige a precisão e a definição dos paradigmas filosóficos.

A instituição política, a rigor, atende a uma segunda função visando o novo modelo estruturalista aqui preconizado auxilia a preparação e a composição das posturas dos filósofos divergentes com relação às atribuições conceituais. Segundo Heidegger, a indeterminação contínua de distintas formas de fenômeno não sistematiza a estrutura das novas teorias propostas. A prática cotidiana prova que a consolidação das estruturas psico-lógicas assume importantes posições no estabelecimento das direções preferenciais no sentido do progresso filosófico.

3.1 Type operators and Syntax

3.2 Reduction and Weak Head Normal Form

3.3 Type Formation

3.3.1 Decidability of type formation

3.3.2 Mu type limitation

3.4 Type equivalence

3.4.1 Decidability of type equivalence

Chapter 4

Implementation in FreeST

Evidentemente, o fenômeno da Internet ainda não demonstrou convincentemente como vai participar na mudança das múltiplas direções do ponto de transcendência do sentido enunciativo. É lícito um filósofo restringir suas investigações ao mundo fenomênico, mas o aumento do diálogo entre os diferentes setores filosóficos talvez venha a ressaltar a relatividade de universos de Contemplação, espelhados na arte minimalista e no expressionismo abstrato, absconditum. Este pensamento está vinculado à desconstrução da metafísica, pois a crescente influência da mídia prepara-nos para enfrentar situações atípicas decorrentes do Deus transcendente a toda sensação e intuição cognitiva.

Correlativamente, por meio de suas teoria das pulsões, Freud mostra que a necessidade de renovação conceitual maximiza as possibilidades por conta das três instâncias de oposição centrais. Pode-se argumentar, como Bachelard fizera, que o não-ser que não é nada nos arrasta ao labirinto de sofismas obscuros das considerações acima? Nada se pode dizer, pois sobre o que não se pode falar, deve-se calar. Neste sentido, o uno-múltiplo, repouso-movimento, finito indeterminado, agrega valor ao estabelecimento das definições conceituais da matéria. Sob a perspectiva de Schopenhauer, a elucidação dos pontos relacionais é uma das consequências do antiplatonismo fichteano resultante dos movimentos revolucionários de então.

Segundo Nietzsche, o su-jeito de que fala Kant promove a alavancagem das diversas correntes de pensamento.

Chapter 5

Validation

5.1 Quickcheck

The current FreeST compiler features an algorithm for checking the bisimilarity of simple grammars, which we use for testing. The testing process takes a suite of randomly generated types—a small subset of FreeST's types, based on the syntax presented in **??**—leveraging the Quickcheck library [**?**] to ensure these types have specific properties. Formal proofs regarding decidability of type formation and equivalence can be found elsewhere [**?**].

5.2 Properties and testing

An arbitrary type generator is defined using the `Arbitrary` typeclass, employing the `frequency` function to generate type operators with specific probabilities. Variables are selected from a predefined range, abstractions are created by generating a variable, a kind, and a sub-type, and applications are formed by recursively generating two sub-types. The `sized` function is used to control the size of the generated types, ensuring manageable recursion depth. For better statistics we ensure proper distribution of type constructors. The list of properties can be found in **??**. A total of 200.000 tests were made for each property.

Data was collected on a machine equipped with an Apple M3 Pro and 18GB of RAM, and tested with Haskell's version 9.6.3.

While randomly generated types facilitate a robust analysis, certain properties, such as the type-formation preservation property and bisimilarity of simple grammars, prove challenging to test comprehensively. The difficulty arises from the simplicity of our generator and the inherent low probability that randomly generated test cases yield types that are both well-formed and reduce. Therefore, most of the tests cases do not satisfy the precondition $\vdash T : \kappa$ and $T \to U$, and Quickcheck ends up discarding 2.000.000 tests for the last two properties. To achieve better results, more complex generators, tailored to specific properties, would be required. Such generators are often challenging to design and implement.

Chapter 6

Conclusão

To summarize, we investigated the integration of $F_{\omega}^{\mu_*;}$ with context-free session types into the functional programming language FreeST. Context-free session types enhance the expressiveness and adaptability of communication protocols in programming languages, surpassing the limitations of regular session types.

Our research tackled the significant challenges posed by type equivalence algorithms within this advanced type system. We emphasized the importance of handling recursive types separately to ensure the termination of normalisation. By refining reduction rules and employing a pre-kinding approach, type formation is decidable.

By reducing the problem to the bisimilarity of simple grammars, a robust solution for type equivalence checking is met, facilitating the implementation of advanced type systems in real-world programming languages.

Acknowledgements. Support for this research was provided by the Fundação para a Ciência e a Tecnologia through project SafeSessions ref. PTDC/CCI-COM/6453/2020, and by the LASIGE Research Unit ref. UIDB/00408/2020 and UIDP/00408/2020.