Compiling General Recursive Functions into Finite Depth Pattern Matching

by

Maycon José Jorge Amaro

Departamento de Computação

Universidade Federal de Ouro Preto

Ouro Preto, Brazil

Compiling General Recursive Functions into Finite Depth Pattern Matching

by

Maycon José Jorge Amaro

A Dissertation submitted for the degree of Master in Computer Science.

Departamento de Computação

Universidade Federal de Ouro Preto

Ouro Preto, Brazil

February, 2023

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

A485c Amaro, Maycon Jose Jorge.
Compiling General Recursive Functions into Finite Depth Pattern Matching. [manuscrito] / Maycon Jose Jorge Amaro. - 2023.
37 f.: il.: tab.. + Código de Linguagem de Programação.

Orientador: Prof. Dr. Rodrigo Geraldo Ribeiro.
Dissertação (Mestrado Acadêmico). Universidade Federal de Ouro Preto. Departamento de Computação. Programa de Pós-Graduação em Ciência da Computação.
Área de Concentração: Ciência da Computação.

1. Recursion. 2. Program transformation. 3. Lambda calculus. I. Ribeiro, Rodrigo Geraldo. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004

Bibliotecário(a) Responsável: Luciana De Oliveira - SIAPE: 1.937.800

FOLHA DE APROVAÇÃO

Maycon José Jorge Amaro

Compiling general recursive functions into finite depth pattern matching

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Mestre em Ciência da Computação

Aprovada em 17 de fevereiro de 2023

Membros da banca

Prof. Dr. Rodrigo Geraldo Ribeiro - Orientador - Universidade Federal de Ouro Preto
Prof. Dr. Bruno Lopes Vieira - Universidade Federal Fluminense
Prof. Dr. Leonardo Vieira dos Santos Reis - Universidade Federal de Juiz de Fora

Prof. Dr. Rodrigo Geraldo Ribeiro, orientador do trabalho, aprovou a versão final e autorizou seu depósito no Repositório Institucional da UFOP em 28/03/2023

Abstract

Programming languages are popular and diverse, and the convenience of extending or changing the behavior of complex systems is attractive even for the systems with stringent security requirements, which often impose restrictions on the programs. A very common restriction is that the program must terminate, which is very hard to check in general because the Halting Problem is undecidable. In this work, we proposed a technique to unroll recursive programs in functional languages to create terminating versions of them. We prove that our strategy is total and we also formalize term generation and run property-based tests to build confidence that the semantics is preserved through the transformation. This strategy can be used to compile general purpose functional languages to targets such as the eBPF and smart contracts for blockchain networks.

Keywords: recursion, program transformation, lambda calculus.

Resumo

Linguagens de programação são populares e diversas, e a conveniência de estender o comportamento de sistemas complexos é atrativo mesmo para aqueles com rígidos requisitos de segurança, que frequentemente impõem restrições aos programas. Uma restrição comum é a de que o programa deve terminar, o que é impossível de se verificar no caso geral, devido à indecidibilidade do Problema da Parada. Neste trabalho, é proposto uma técnica para desenrolar funções recursivas em linguagens funcionais para criar versões terminantes delas. É provado que essa estratégia é total e é formalizada a geração de termos aleatórios, que possibilitam a execução de testes baseados em propriedades para construir confiança de que a semântica é preservada na transformação das funções. A técnica proposta pode ser usada para compilar linguagens funcionais de propósito geral para eBPF, contratos inteligentes de redes de *blockchain* e outros alvos igualmente restritivos.

Palavras-chave: recursão, transformação de programas, cálculo lambda.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor, teacher, and friend Rodrigo Ribeiro, for his patience, motivation, knowledge, and empathy. The completion of this work was only possible because of him. To my mother and sister for their support in my academic journey. To my friends, who helped me endure the last two years. To my colleagues, for the great moments in the classes. To Lucilia Figueiredo and Dayanne Coelho, who welcomed and guided me during my graduation. To the dogs and cats that brought joy to my days. Finally, I am grateful to the Department of Computing of the Federal University of Ouro Preto, for the opportunity to engage in this graduate program; and to CAPES for the financial support.

Contents

List of Grammars

List of Tables

List of Examples

"Our histories never unfold in isolation. We cannot truly tell what we consider to be our own histories without knowing the other stories. And often we discover that those other stories are actually our own stories."

Angela Davis

1 Introduction

Since the creation of personal computers, Computer Science has evolved a lot. Computer scientists developed more efficient data structures and algorithms, more complex Operating Systems and solutions to several security issues. Also, programming has become quite popular with a small group of languages always hitting the charts[1]. Unfortunately, one of the most basic features of programming languages is often seen as a prominent cause of trouble: loops. Either in form of recursion or iterative statements, non-termination is at best a necessary tool to keep a web-server or OS running, and at worst a serious security or logical concern. It is the reason why some languages and technologies try to avoid it at all costs, being very restricted due to the undecidability of the Halting Problem [39]. Some examples include dependently-typed languages that are used as proof assistants, such as Coq and Agda; smart contract languages for blockchain systems and the technology to run sandboxed programs into the Linux Kernel—the eBPF[2].

Dependently-typed languages usually have a termination checker enabled by default, making sure that only terminating programs are accepted. From a logical point of view, an infinite loop can be used to prove anything, turning the whole system unsound. Apart from logical predicates, these proof assistants can also be used to guarantee that your particular implementation of some algorithm halts for all valid inputs. Then, it is easy to see that translating this implementation into a general purpose language will keep its halting property. In fact, both Coq and Agda offer Haskell as a target for compiling their programs. Compiling from a more restricted scenario to a less restricted one is quite trivial, but the converse is not true.

Some tasks, specially involving networks, are better made by modifying some behavior in the system's kernel. Waiting for this change to come up eventually or even writing and maintaining a kernel module is burdensome. For this reason, people find it useful to be able to programmatically change the behavior of some subsystem in the kernel. The eBPF allows the execution of programs in the kernel. But the kernel is an extremely sensitive place. One mistake could be enough to put the entire system at risk. A non-terminating program is out of question, it could crash the system or be used to perform a Denial of Service (DoS) attack. A similar situation happens on smart contracts for blockchain systems: a non-terminating code could collapse the whole system and a lot of strategies are implemented to avoid it [22]. These two environments actually have an intersection.

[1]See `statisticsanddata.org/data/the-most-popular-programming-languages-1965-2021` for a brief history on popularity of programming languages.

[2]extended Berkeley Packet Filter

For instance, the blockchain Solana implements a Virtual Machine and a just-in-time compiler for eBPF programs as a Rust crate[3].

Programs in eBPF are usually written indirectly, using tools that create the code from some definitions the users provide in their interface. But many programs have to be written directly, be it in eBPF's bytecode or using some compiler. BCC[4] is a well-known tool that uses LLVM[5] backend to compile pseudo-C code into eBPF. To ensure termination for such programs, eBPF forbids any kind of back edge in the programs' control flow graph, which include recursive functions. Prior to version 5.3 of the Linux Kernel, not even bounded loops were allowed, forcing users of BCC to explicitly ask `clang` to unroll their loops via `pragma`. This situation imposes a barrier for creating compilers from general purpose languages to eBPF.

One of the main aspects involved in Turing-completeness—a property shared by most famous programming languages—is the ability to run general recursive functions. While imperative languages have additional constructs for repetition, functional languages like Haskell and Elixir can only count with recursion for repeating computations. Using these languages to write programs for eBPF requires caution and strategies to ensure termination when using recursion.

In this work, we formalize, using the dependently-typed language Agda, an algorithm to unroll and transform recursive functions of functional languages into finite depth pattern matching. The resulting function is *equivalent* to the original, in the sense that both functions yield the same results when given the same inputs, but only if the original function halts and the non-recursive function has enough nested pattern matchings to produce a value. This flexibility of equivalency is necessary because a non-terminating program cannot formally have the same semantics of a function that always halts. By using this strategy, compilers from functional languages can be written targeting eBPF and others targets that restrict loops and recursion.

1.1 Objectives

The main objective of this work is the definition and verification of a technique for unrolling general recursion into an *equivalent* terminating pattern matching. More specifically, we intend to:

- Present a core language with recursion and its unrolling algorithm in Agda.
- Present a core language with no recursion and the translation algorithm, in Agda.

[3]Crates are the name for Rust's published packages. Solana's module for eBPF can be found in `docs.rs/solana_rbpf/0.2.11/solana_rbpf`

[4]BPF Compiler Collection

[5]This name used to be an acronym for Low Level Virtual Machine. Today, the LLVM Developer Group claims that LLVM is the full name of the project, rather than an acronym. This information can be checked at their official website https://llvm.org/.

- Describe an algorithm to generate random well-typed terminating programs.
- Build confidence of the correctness of the strategy, formally proving some properties regarding the presented algorithms, such as soundness and termination, and applying property-based tests otherwise.

1.2 Contributions

Our contributions are:

- An algorithm to unroll general recursive functions and transform them into finite nesting of pattern matchings.
- A mechanized proof that the algorithm always terminates (Corollary 1).
- A mechanized proof that every output function always terminates (Theorem 7).
- An experiment supporting that, when the original and resulting functions produce some value, it will always be the same.
- Ringell, a proof-of-concept interpreter using this technique.

1.3 Published Material

This dissertation is built upon a paper published in a peer-reviewed conference. "A Sound Strategy to Compile General Recursive Functions into Finite Depth Pattern Matching" is described in our SBMF 2022 paper [6].

1.4 Dissertation Structure

The remaining content of this dissertation is structured as follows: Chapter 2 covers the background knowledge used in this work, Chapter 3 presents the content concerning the syntactic transformation, Chapter 4 discusses semantic properties, and Chapter 5 concludes everything. The Agda code is found in `https://github.com/lives-group/terminating-expansion`.

2 Background

This chapter presents the key concepts necessary to the development of this work. The examples in these sections will make use of a language of arithmetics expressions, called $\mathbb{A}$, which abstract syntax is pictured in Grammar 2.1. It features numbers, booleans, a test and a conditional expression. This language is largely inspired by Pierce's $\mathbb{BN}$ [32].

Grammar 2.1: Syntax of $\mathbb{A}$ language for arithmetic expressions

$$
\begin{array}{rl}
\langle e \rangle ::= & \texttt{zero} \mid \texttt{true} \mid \texttt{false} \mid \texttt{suc}\ \langle e \rangle \mid \texttt{pred}\ \langle e \rangle \\
\mid & \texttt{iszero}\ \langle e \rangle \mid \texttt{if}\ \langle e \rangle\ \texttt{then}\ \langle e \rangle\ \texttt{else}\ \langle e \rangle
\end{array}
$$

2.1 Type Systems

Although some languages implement meaning to expressions like 2+`'v'`, it has no apparent meaning on its own, because adding is supposed to be an operation over numbers. Classifying data according to the operations they are compatible with is one of most praised features of programming languages. This classification is what we call **types**. A **Type System** is a set of rules of how the syntactical constructions of a language should be typed [32]. **Typechecking** a program means verifying if the syntax has a valid derivation of the type system.

One main property we often use to categorize programming languages is with respect to whether or not we need to annotate types, and whether these types are checked during compilation-time (static), runtime (dynamic) or a mix of both. Annotating types of function parameters helps the compiler catch, before the program is even executed, errors that would lead to undefined behavior. Also, previously knowing the structure offers the compiler great opportunities for optimizations. Siek & Taha's work [36] has an interesting discussion about pros and cons of static and dynamic type systems.

We can easily define a Type System for $\mathbb{A}$, classifying `zero` and `suc` as constructions for natural numbers, `true` and `false` as constructions for booleans and typing the other constructions according to how they are supposed to behave. Rules for a Type System are usually represented in Gentzen's natural deduction style, in which the premisses are on top of a line above the conclusion. Table 2.1 pictures such representation for $\mathbb{A}$'s type system. Judgment $e : \tau$ means expressions e has type τ. Since we did not define a syntax for types, we'll be using $\mathbb{B}$ to denote the type of booleans, and $\mathbb{N}$ to denote the type of natural numbers. Although we do not annotate types, this type system can be used in a static analysis, because the language is simple enough for us to infer the correct types for every valid expression. It can also happen with languages in which the inversion of the type system remains deterministic.

Table 2.1: Type System for $\mathbb{A}$

$$\frac{}{\mathrm{zero} : \mathbb{N}} \qquad \frac{\mathrm{e} : \mathbb{N}}{\mathrm{suc\ e} : \mathbb{N}} \qquad \frac{}{\mathrm{true} : \mathbb{B}}$$

$$\frac{}{\mathrm{false} : \mathbb{B}} \qquad \frac{\mathrm{e} : \mathbb{N}}{\mathrm{iszero\ e} : \mathbb{B}} \qquad \frac{\mathrm{e} : \mathbb{N}}{\mathrm{pred\ e} : \mathbb{N}}$$

$$\frac{\mathrm{e}_1 : \mathbb{B} \qquad \mathrm{e}_2 : \tau \qquad \mathrm{e}_3 : \tau}{\mathrm{if\ e}_1\ \mathrm{then\ e}_2\ \mathrm{else\ e}_3 : \tau}$$

Consider the expression `if true then (suc zero) else zero`. It has a valid type derivation from the rules defined, shown in Example 2.1, and so we say it is well typed. Regardless of its meaning, we know that it will produce a value. No invalid expression such as `suc false` will have a typing derivation, but some expressions that would produce a value can be ruled out by the type system. If we define `if` construction to become its second argument when the first is `true`, then the expression `if true then zero else suc false` would always produce a value, but the Type System will not allow its second and third argument to be of different types, nor it will allow any of them to be of an undefined type.

Example 2.1: Typing derivation for `if true then suc zero else zero`

$$\frac{\dfrac{}{\mathrm{true} : \mathbb{B}} \qquad \dfrac{\dfrac{}{\mathrm{zero} : \mathbb{N}}}{\mathrm{suc\ zero} : \mathbb{N}} \qquad \dfrac{}{\mathrm{zero} : \mathbb{N}}}{\mathrm{if\ true\ then\ (suc\ zero)\ else\ zero} : \mathbb{N}}$$

Type Systems are useful to avoid several kinds of errors, but it is inevitable to reject some programs that would run without problems. A Type System that rejects all incorrect programs—and only them—would solve the Halting Problem, thus its existence is impossible. Type Theory research includes balancing restrictiveness and flexibility. More deep content about Type Systems is found in [32].

2.2 Formal Semantics

In a broad sense, *semantics* is the meaning of symbols and sentences. In Computer Science, Formal Semantics is the rigorous specification of the meaning or behavior of programs [27]. The use of formal methods when defining semantics of programming languages can reveal ambiguities in informal specifications and serve as the basis for implementation or proof of correctness. Semantics are defined for syntactically valid and well-typed programs alone, because invalid programs are not supposed to have meaning. There are three main approaches to formal semantics, known as Operational, Denotational and Axiomatic. The Axiomatic approach involves defining a logical system for partial correctness in the language, and then prove properties in form of assertions. Due to its specificity, we do not get in details about it. Operational and Denotational approaches are explained in the following sections.

2.2.1 Operational Semantics

The Operational approach focuses on defining *how* the computations are done, regarding some abstract machine, and with that defining *what* they mean. With an operational semantics, it is straightforward to implement an interpreter for the language. It is subdivided in Natural Semantics and Structural Operational Semantics.

Natural Semantics [20], also known as *big-step* semantics, describes how the overall results are obtained [27], relating the constructions of the language with the final value or effect they produce. Table 2.2 presents a natural semantics for $\mathbb{A}$. The judgment $e \Downarrow v$ means expression e evaluates to final value v. Values are `true`, `false`, `zero` or the `suc` of a value. The first rule, indicating the reflexivity of $\Downarrow$, is useful for proving premises in derivations. Example 2.2 represents a derivation of the natural semantics proving that the expression `if true then (pred (suc (pred zero))) else zero` evaluates to `zero`.

Table 2.2: Natural Semantics for $\mathbb{A}$

$$\frac{}{v \Downarrow v} \qquad \frac{e \Downarrow v}{\text{suc } e \Downarrow \text{suc } v} \qquad \frac{e \Downarrow \text{suc } v}{\text{pred } e \Downarrow v}$$

$$\frac{e \Downarrow \text{zero}}{\text{pred } e \Downarrow \text{zero}} \qquad \frac{e \Downarrow \text{zero}}{\text{iszero } e \Downarrow \text{true}} \qquad \frac{e \Downarrow \text{suc } v}{\text{iszero } e \Downarrow \text{false}}$$

$$\frac{e_1 \Downarrow \text{true} \qquad e_2 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v} \qquad \frac{e_1 \Downarrow \text{false} \qquad e_3 \Downarrow v}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Downarrow v}$$

Example 2.2: Big-step evaluation of `if true then pred suc pred zero else zero`

$$\dfrac{\dfrac{}{\text{true} \Downarrow \text{true}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\text{zero} \Downarrow \text{zero}}}{\text{pred zero} \Downarrow \text{zero}}}{\text{suc (pred zero)} \Downarrow \text{suc zero}}}{\text{pred (suc (pred zero))} \Downarrow \text{zero}}}{\text{if true then (pred (suc (pred zero))) else zero} \Downarrow \text{zero}}$$

Structural Operational Semantics [34], also known as *small-step* semantics, describes how the individual steps are performed, with details on how each construction achieves its final value or effect. Table 2.3 shows the small-step semantics for $\mathbb{A}$. Judgment $e \longrightarrow e'$ means expression e reduces to e' in one step, and $\longrightarrow^*$ represents the reflexive and transitive closure of $\longrightarrow$. Example 2.3 describes the computation steps that reduce expression `if true then (pred (suc (pred zero))) else zero` to `zero`.

Table 2.3: Structural Operational Semantics for $\mathbb{A}$

$$\dfrac{}{\text{pred (suc } v) \longrightarrow v} \qquad \dfrac{}{\text{iszero zero} \longrightarrow \text{true}} \qquad \dfrac{e \longrightarrow e'}{\text{iszero } e \longrightarrow \text{iszero } e'}$$

$$\dfrac{}{\text{iszero (suc } v) \longrightarrow \text{false}} \qquad \dfrac{}{\text{if true then } v \text{ else } e \longrightarrow v} \qquad \dfrac{}{\text{if false then } e \text{ else } v \longrightarrow v}$$

$$\dfrac{}{\text{pred zero} \longrightarrow \text{zero}} \qquad \dfrac{e_1 \longrightarrow e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \longrightarrow \text{if } e_1' \text{ then } e_2 \text{ else } e_3}$$

$$\dfrac{e \longrightarrow e'}{\text{suc } e \longrightarrow \text{suc } e'} \qquad \dfrac{e_2 \longrightarrow e_2'}{\text{if true then } e_2 \text{ else } e_3 \longrightarrow \text{if true then } e_2' \text{ else } e_3}$$

$$\dfrac{e \longrightarrow e'}{\text{pred } e \longrightarrow \text{pred } e'} \qquad \dfrac{e_3 \longrightarrow e_3'}{\text{if false then } e_2 \text{ else } e_3 \longrightarrow \text{if false then } e_2 \text{ else } e_3'}$$

Example 2.3: Small-step evaluation of `if true then pred suc pred zero else zero`

$$\begin{array}{lll}
 & \text{if true then pred (suc (pred zero)) else zero} & \\
\longrightarrow^* & \text{if true then (pred (suc zero)) else zero} & \langle \text{ pred zero} \longrightarrow \text{zero } \rangle \\
\longrightarrow^* & \text{if true then zero else zero} & \langle \text{ pred suc } v \longrightarrow v \ \rangle \\
\longrightarrow^* & \text{zero} &
\end{array}$$

If done carefully, it is possible to define a big-step and a small-step semantics for one language and then prove both semantics equivalent, in the sense that every program that terminates with a value in one will necessarily terminate with the same value in the other. However, there are some essential differences between them. For example, big-step semantics cannot distinguish between non-termination and abnormal termination, unless the abnormal termination is modelled as a value or a termination state; while small-step semantics will derive an infinite sequence of steps for non-terminating programs and get stuck for abnormal states. Some other examples include the inability of small-step to suppress loops in non-determinism and the inability to express interleaving of parallel computations in big-step semantics. The details about this information can be found in [27, 32].

2.2.2 Denotational Semantics

The Denotational approach concerns only the effect of executing a program, completely abstracting how it is obtained. This is made by mapping the syntactical constructions of the language to mathematical objects [27]. In this way, reasoning about a program is just reasoning about these mathematical objects, and their known properties are immediately applicable. Let $\mathbb{D}$ be the domain of $\mathbb{A}$'s syntax and let $\mathbb{N}$ be the set of natural numbers. We can define a **semantic function** $\mathcal{S} : \mathbb{D} \to \mathbb{N}$ to act as the denotation. Table 2.4 describes $\mathcal{S}$. One remarkable characteristic of denotational semantics is that it is *compositional*: every basic construction has a denotation and every composite construction has a denotation that makes use of its immediate constituents' denotation. In other words, semantic functions for denotational semantics are often recursive.

Table 2.4: Denotational Semantics of $\mathbb{A}$

$$\begin{aligned}
\mathcal{S}[\![\text{zero}]\!] &= 0 \\
\mathcal{S}[\![\text{suc } e]\!] &= \mathcal{S}[\![e]\!] + 1 \\
\mathcal{S}[\![\text{pred } e]\!] &= \max(0, \mathcal{S}[\![e]\!] - 1) \\
\mathcal{S}[\![\text{true}]\!] &= 1 \\
\mathcal{S}[\![\text{false}]\!] &= 0 \\
\mathcal{S}[\![\text{iszero } e]\!] &= \begin{cases} 1, & \text{if } \mathcal{S}[\![e]\!] = 0 \\ 0, & \text{otherwise} \end{cases} \\
\mathcal{S}[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!] &= \begin{cases} \mathcal{S}[\![e_2]\!], & \text{if } \mathcal{S}[\![e_1]\!] = 1 \\ \mathcal{S}[\![e_3]\!], & \text{otherwise} \end{cases}
\end{aligned}$$

Using the same example of operational semantics, we show in Example 2.4, through mathematical reasoning, that $\mathcal{S}[\![$ `if true then (pred (suc (pred zero))) else zero` $]\!]$ equals 0. Notice that this definition only applies to expressions with no syntactical or typing errors. If this denotation is applied to an expression with type error, it could actually answer with a number, though it would not correspond to the expression's meaning.

A powerful concept relating denotational and operational semantics is **full abstraction**. A fully abstracted semantics constitutes a denotational and an operational semantics that are equivalent, in the sense that the denotation of the values computed by the operational semantics is exactly the denotation of the expression itself. In this way, we can implement and reason about the language using the formalizations interchangeably. More information on denotational semantics and full abstraction can be found in [27].

Example 2.4: Denotation of `if true then pred suc pred zero else zero`

$$\begin{aligned} & \mathcal{S}[\![\text{if true then (pred (suc (pred zero))) else zero}]\!] \\ = {} & \mathcal{S}[\![\text{pred (suc (pred zero))}]\!], \text{ because } \mathcal{S}[\![\text{true}]\!] = 1 \\ = {} & \max(0, \mathcal{S}[\![\text{suc (pred zero)}]\!] - 1) \\ = {} & \max(0, \mathcal{S}[\![\text{pred zero}]\!] + 1 - 1) \\ = {} & \max(0, \max(0, \mathcal{S}[\![\text{zero}]\!] - 1) + 1 - 1) \\ = {} & \max(0, \max(0, 0 - 1) + 1 - 1) \\ = {} & \max(0, 0 + 1 - 1) \\ = {} & 0 \end{aligned}$$

2.2.3 A Look Around

Curiously, while Standard ML does have a complete formal semantics defined [26], most programming languages actually do not. As said Appel [8]:

> ML has a formally defined semantics that is complete in the sense that each legal program has a deterministic result, and all illegal programs are recognizable as such by a compiler. This is in contrast to Ada, for which formal semantics have been written but are not complete, in the sense that some "erroneous" programs must be accepted by compilers; Pascal, which is recognized to have certain "ambiguities and insecurities"; and C, in which it is very easy for programmers to trash arbitrary parts of the runtime environment by careless use of pointers. Lisp and Scheme are not too bad in this respect; in principle, "erroneous" programs are detected either at compile time or at runtime, though certain things (such as order of evaluation of arguments) are left unspecified.

A notable case is that of JavaScript, with several attempts to give it a formal operational semantics in literature. We highlight that of Park et al. [31], that presents a complete operational semantics for JavaScript, admitting "JavaScript is an unusual language, full of tricky corner cases" and that "seemingly nonsensical programs work by design." Their formalization passed all ECMAScript 5.1 conformance tests, something achieved only by Chrome's V8. The authors report that those conformance tests fail to cover several semantic rules, and that they found several bugs in all major JavaScript engines. At least four new versions of the language were released since then.

2.3 Property Based Testing

By using formal reasoning, with the proper tools, one can effectively prove properties of programs such as correctness and complexity. But sometimes it can be quite hard or expensive, creating an appealing situation to weaken the claim or the method. Testing is largely used in Software Engineering, and although they cannot prove the absence of errors [32], they are a good tool to assure some quality and can be used to build confidence before actually trying to prove some property. An interesting form of testing is known as Property Based Testing [29], where the developer writes the properties and then some automatic checker generates the cases and test them. Such generator must be good at checking corner cases, doing massive unit tests if asked and shrinking the counterexamples to a minimum version. QuickCheck is a famous property-based testing library, and it's available for several languages, including Haskell [29]. Successfully passing the tests with QuickCheck and ensuring that all lines of code got covered is a good evidence that the algorithm is correct, but it is still no proof. Property based testing is still testing, and therefore cannot prove the absence of errors.

"Real World Haskell" book [29] provides good basic examples and tutorials on QuickCheck, and we reproduce one below. It is a sorting function, called `qsort` and a property stating *idempotency*[1].

```
qsort :: Ord a => [a] -> [a]
qsort []     = []
qsort (x:xs) = qsort lhs ++ [x] ++ qsort rhs
  where lhs = filter (< x) xs
        rhs = filter (>= x) xs

prop_idempotent xs = qsort (qsort xs) == qsort xs
```

When asking QuickCheck to verify the property, it will generate random lists and check if `prop_idempotent` is true for each one of them. In negative case, it will reduce the counterexample size as much as possible before reporting. The default number of generations is 100, but can easily be changed through `withMaxSuccess` parameter. QuickCheck has default generators implemented for numbers, lists and tuples. Custom datatypes require that the developer writes the generator using QuickCheck's API[2]. The official QuickCheck manual is hosted at professor John Hughes' webpage[3] from Chalmers University.

[1]A property where the function can be applied several times without changing the result beyond the initial application

[2]Application Programming Interface

[3]Available in http://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html

2.4 Agda

A dependently-typed language has a type system expressive enough so types can depend on terms. Due to the Curry-Howard Isomorphism [40], that establishes the correspondence between type systems and logical systems, types in such languages can be saw as propositions of second order logic, and programs as their proofs. Typechecking the program means verifying if the proof is sound. However, an infinite loop can be used to prove anything, and thus a dependently-typed language can only be used as a proof assistant if the compiler rejects everything that it cannot guarantee termination. Also, all functions must be defined for every possible case, i.e., incomplete patterns are usually not allowed.

Agda is a dependently-typed functional language created by Norell [28], and since it features an enabled-by-default termination checker, it is also a proof assistant. Agda's syntax is quite similar to Haskell's. To prevent a paradox, Agda has an infinite hierarchy of types, in which basic types are of type `Set` or $\texttt{Set}_0$, which in turn is of type $\texttt{Set}_1$ and so forth. Below is a basic type declaration of natural numbers in Peano notation and its correspondent adding operation. Notice Agda has a great support for unicode symbols.

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

_+_ : ℕ → ℕ → ℕ
zero    + m = m
(suc n) + m = suc (n + m)
```

The cliché example to demonstrate the use of dependent types is the `Vec` datatype, a list that contains its size on its type. Below is the definition of `Vec` and a function `head` that returns its first element. Since the size of the list is on its type, we can define `head` so it will only accept non-empty lists as input, and this constraint is verified during typechecking. Without dependent types, it would be harder or even impossible to let the compiler know that empty lists shouldn't be allowed, forcing the developer to check it at runtime.

```
data Vec (A : Set) : ℕ → Set where
  []  : Vec A zero
  _::_ : ∀{n : ℕ} → A → Vec A n → Vec A (suc n)

head : ∀{A : Set}{n : ℕ} → Vec A (suc n) → A
head (x :: xs) = x
```

Finally, a proof is just a function as any other, with the proposition being the stated type. Before we show a proof, consider the following example with the definition of a traditional list type, a concatenation operation and the `length` function that computes the size of a list.

```
data List (A : Set) : Set where
  []   : List A
  _::_ : A → List A → List A

_++_ : ∀{A : Set} → List A → List A → List A
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)

length : ∀{A : Set} → List A → ℕ
length []        = zero
length (x :: xs) = suc (length xs)
```

The proof that the concatenation of two lists with sizes n and m produces a list with size $n+m$ is presented next. The proof proceeds by induction on the first list. The proof cases are pattern matching equations, and the inductive step is represented by the recursive call under a `rewrite` clause. This clause tells the typechecker to use an equivalency to substitute terms and normalize them again. By using the induction hypothesis as this equivalency, the proof goal becomes easier or even trivial to achieve.

```
++-length : ∀{A : Set}(xs ys : List A)
  → length (xs ++ ys) ≡ length xs + length ys
++-length []        ys     = refl
++-length (x :: xs)  ys
  rewrite ++-length xs ys = refl
```

With the use of dependent types we can even define an operation that is **correct by construction**, in which the desired property is part of the definition and we don't need to prove it separately. Below, the same concatenation operation for `Vec`'s. Since the type of the operation states that it produces a `Vec` with the sum of the sizes, this property is guaranteed to hold as long as the compiler accepts the definition. Correct by construction definitions are often easier and shorter.

```
_++_ : ∀{A : Set}{n m : ℕ}
  → Vec A n → Vec A m → Vec A (n + m)
[]         ++ ys = ys
(x :: xs)  ++ ys = x :: (xs ++ ys)
```

A Standard Library for Agda is publicly available in the official site[4], although it does not come in the installation. The Standard Library features definitions for naturals, equality, foundations for relations, algebra definitions and a lot more. On using Agda as both programming language and proof assistant, see [41, 38].

[4]Available in https://agda.readthedocs.io/en/v2.6.2.1/getting-started/installation.html

2.5 Lambda Calculus

In 1928, David Hilbert and Wilhem Ackermann posed a challenge to the mathematics community, that became known as the *Entscheidungsproblem*: find an algorithm that could answer if a given statement of first order logic is universally valid. To complete the challenge, it was necessary to precisely define *algorithms*, which was achieved by two main works: Turing's abstract machines [39] and Church's untyped λ-calculus [12], both answering that no general solution to the Entscheidungsproblem is possible.

The lambda calculus is the underlying core theory of many programming languages we use today, specially functional languages. The untyped version consists of a language with only three constructions: variables, abstractions (function definitions) and applications; and the usual syntax is shown in Grammar 2.2. The identity function $\lambda x.x$ is an example.

Grammar 2.2: Syntax of untyped λ-calculus

$$\langle e \rangle ::= \quad v \mid \lambda v.\langle e \rangle \mid \langle e \rangle \; \langle e \rangle$$

We say that an abstraction *binds* a variable, and a variable with no binding lambda is called free. Bound variables can be renamed without changing the meaning of the expression, which is known as α-equivalency. When the left expression of an application is an abstraction, it is called a bf redex and we can substitute the free ocurrences of the bound variable by the right expression, which is known as β-reduction. This substitution must avoid capturing names that are already used, so α-equivalency must applied to rename variables sometimes. For example, the right expression in the application $(\lambda x.\lambda y.y\; x)\; y$ is a free variable, which is bound in the left. So, we first use α-equivalency to get $(\lambda x.\lambda z.z\; x)\; y$ and then reduce it to $(\lambda z.z\; y)$. The order in which redexes are evaluated configures an evaluation strategy. The most common strategy is *call-by-value*, in which the leftmost-outermost redex is evaluated first, but only after both sides are already reduced to a normal form. Some expressions like $(\lambda x.x\; x)\; (\lambda x.x\; x)$ can β-reduce forever, and thus the untyped λ-calculus cannot be used as a logical system. The details about β-reduction are illustrated in the small step semantics presented in Table 2.5, following Pierce [32]. Notation $[x \mapsto v]e$ means capture avoiding substitution of x by v in e.

Table 2.5: Call-by-value, small step semantics for untyped λ-calculus

$$\frac{e_1 \longrightarrow e_1'}{e_1\; e_2 \longrightarrow e_1'\; e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\; e_2 \longrightarrow v_1\; e_2'} \qquad \frac{}{(\lambda x.e_1)\; v_2 \longrightarrow [x \mapsto v_2]e_1}$$

Capture-avoiding substitution can be hard to implement, which created the need for a nameless representation of bound variables. De Bruijn indices [15] omit names in the abstractions and write natural numbers in the occurrences of the bound variables. The number represents the distance, in the scope, of the variable and the lambda that bound it. In this way, $(\lambda x.\lambda y.y\,x)\,y$ can be written as $(\lambda\lambda.0\,1)\,y$ and its β-reduction no longer needs an α-equivalency to avoid the capture of names.

The simply typed λ-calculus (STLC) [13], shown in Grammar 2.3, adds simple types to the calculus, and its type system, shown in Table 2.6, rules out expressions that β-reduce forever. γ is base type constructor. Judgement $\Gamma \vdash e : \tau$ means expression e has type τ in context Γ. A context is a list of information about the types of variables in a expression. The empty context is represented by $\varnothing$ and notation $\Gamma, v : \tau$ represents that context Γ is extended with information that v has type τ. The simply typed λ-calculus is **strongly normalizing**, i.e., every expression finitely β-reduces to a normal form [32, 41]. STLC can be used as a logical system, being equivalent to intuitionistic propositional logic [40].

Grammar 2.3: Syntax of simply typed λ-calculus

$$\begin{aligned} \langle\tau\rangle &::= \gamma \mid \tau \to \tau \\ \langle e\rangle &::= v \mid \lambda v : \tau.\langle e\rangle \mid \langle e\rangle\ \langle e\rangle \end{aligned}$$

Table 2.6: Type System of simply typed λ-calculus

$$\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \qquad \frac{\Gamma, v : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda v : \tau_1.e\ : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2\ : \tau_2}$$

When defining λ-calculi in dependently typed languages, one can go on two approaches: extrinsic and intrinsic. The extrinsic approach is to define terms and types separately and then define how to type the terms. The intrinsic approach defines terms intertwined with their types, so only well-typed terms can even be expressed. The Agda fragments presented next picture an extrinsic formalization of STLC.

First we define types, following our definition in Grammar 2.3.

```
data Type : Set where
  γ    : Type
  _⇒_  : Type → Type → Type
```

Then we define terms, also based on our grammar.

```
data Term : Set where
  var : Name → Term
  abs : Name → Type → Term → Term
  app : Term → Term → Term
```

Finally, we define how to type terms, strictly following Table 2.6. With this, we can write a typing derivation and have Agda to typecheck if it is correct.

```
data _⊢_:_ : Context → Term → Type → Set where
  tvar : ∀{v t Γ}
         → v : t ∈ Γ
         → Γ ⊢ (var v) : t
  tabs : ∀{v e t t' Γ}
         → (Γ , v : t) ⊢ e : t'
         → Γ ⊢ (abs v t e) : (t ⇒ t')
  tapp : ∀{e e' t t' Γ}
         → Γ ⊢ e : (t ⇒ t')
         → Γ ⊢ e' : t
         → Γ ⊢ (app e e') : t'
```

In the intrinsic approach, presented next, the type system practically becomes the definition of terms. Only well typed terms can be expressed.

```
data _⊢_ : Context → Type → Set where
  var : ∀{t Γ}
        → t ∈ Γ
        → Γ ⊢ t
  abs : ∀{t t' Γ}
        → (Γ , t) ⊢ t'
        → Γ ⊢ (t ⇒ t')
  app : ∀{t t' Γ}
        → Γ ⊢ (t ⇒ t')
        → Γ ⊢ t
        → Γ ⊢ t'
```

The intrinsic approach is highly compatible with correct by construction proofs, because the definitions already contain some of the desired properties. Intrinsic definitions are often more compact and lead to easier proofs, specially with the use of De Bruijn indices. Notice that in Agda, the use of dependent types allow us to define that a variable can only be expressed if we have a proof that it is in the context. It is known as well scoped De Bruijn indices [14]. More content on λ-calculi can be found in [32], and some Agda formalizations for them, both extrinsic and intrinsic, can be found in [41]. For a brief history on the Entscheidungsproblem, see [40, 16]. On Turing machines, see [37].

2.6 Related Work

There are some papers closely related to ensuring termination of functional programs, although the systems they describe do not focus on syntactically transforming the programs. Even with a proof of the program termination, targets such as eBPF will still reject the program if it contains any loop. Our approach creates non-recursive functions and thus is more suitable to create compilers for those targets.

Developed by Andreas Abel in 1998, Foetus [1, 2, 3, 4] is a simplification of Munich Type Theory Implementation (MuTTI), and features a termination checker for simple functional programs. It builds an hypergraph that represents the transitive closure of function calls. Functions related to themselves in this hypergraph are considered recursive and their *dependencies* are analyzed. It means that foetus tries to find a lexicographic order in the parameters, accepting the function as terminating if positive. The identification of dependencies is quite limited, being unable to store dependency information from let assignments or consider components of tuples in the same way it considers parameters. To get around the problem, one can define the functions in a specific way to help foetus identify the dependencies. For example, when defining a function that only decreases the arguments in a even number of calls, foetus will not accept the function as terminating, but it is possible to create a copy of the function and mutual recursion them, creating a more obvious dependency that foetus will identify and accept.

Type based termination [17, 11] is a strategy in which the type system of the language ensures termination, being developed for the polymorphic λ-calculus. For this, the types of the language are modified to contain a *size* information of the inductive datatypes, and are used in typechecking to guarantee that recursive calls are always reducing the size of some argument. The actual types of data are inferred in an independent step. Despite elegant, this strategy is reasonably complicated. One minor disadvantage is the forbidding of “left recursion” on types, which is known as *negative occurrence*. Gilles Barthe has an excellent tutorial [10] on type based termination.

On syntactic transformations, Rugina & Rinard [35] defines a strategy to unroll recursion in divide-and-conquer programs of imperative languages. Their intention is to increase performance, and no contribution towards ensuring termination is made. However, the key concepts in their work are very useful. Also, LLVM’s Clang [23] offers a macro to unroll bounded loops in languages of C family, which is largely used when using pseudo-C code to compile for eBPF. GCC [19] has a command line option to unroll loops in C programs. The features in both compilers are unable to deal with recursion. At the time of this paper, Microsoft® Visual Studio C Compiler (MSVC) [25] has no way to directly instruct the compiler to unroll a loop.

Finally, our generation procedure and test approach are based on the work in [18]. Feitosa formalizes a type-directed algorithm to generate random programs of Featherweight Java, which is used to verify several properties using QuickCheck. Pałka’s thesis [30] also contains useful concepts on term generation.

3 Syntax

In order to transform recursive functions into non-recursive pattern matching, we could just define an algorithm in which both input and output are expressions of a functional language. But it is convenient to work with all forms of expressions instead of restricting ourselves to functions. Also, we are interested in the properties of the expressions both before and after the transformation, so it is convenient to have two languages and model the transformation as a transition between them. In this way, we define System R and System L. The former features general recursion while the latter has no way of expressing it. The syntactical transformation from System R to System L is the process of eliminating recursion. This chapter describes the details of both languages and the transformation procedure. All referenced Agda files are hosted in `https://mayconamaro.github.io/dissertation-agda/`. If you're reading the digital version, you can also click in the files' names.

3.1 System R

The letter R stands for *recursion*. This language is very similar to the language of Programming Computable Functions, also known as PCF [5, 33], and its syntax is shown in Grammar 3.1. System R features natural numbers, function definition and application, pattern matching over naturals and a recursion operator (fixpoint) that can only appear as the top level expression, or inside a top level function application. Top level applications can be nested. System R is a statically typed language, with syntax for type annotations in the abstractions. For convenience, this definition of the language forces the use of a *potentially* recursive function and has no way to define mutually recursive functions.

Grammar 3.1: Syntax of System R

$$\begin{aligned}
\langle \tau \rangle &::= \texttt{nat} \mid \langle \tau \rangle \to \langle \tau \rangle \\
\langle p \rangle &::= \mu v.\langle e \rangle \mid \langle p \rangle\ \langle e \rangle \\
\langle e \rangle &::= v \mid \texttt{zero} \mid \texttt{suc}\ \langle e \rangle \mid \lambda v{:}\langle \tau \rangle.\langle e \rangle \mid \langle e \rangle\ \langle e \rangle \mid \texttt{match}\ \langle e \rangle\ \langle e \rangle\ (v,\ \langle e \rangle)
\end{aligned}$$

Table 3.1 presents its type system. Notice that, although the type system does not enforce it, the grammar still forbids `rec`'s as subterms, unless it is the left expression of a top level application. This is a syntactic constraint, not a typing one. Since typechecking is only done with syntactically correct expressions, it does not pose a problem. A variable occurrence (Definition 1) is the expression with the variable.

Definition 1. A variable v occurs in expression e, represented by $v \in_v e$, if e is the variable v or if $v \in_v e'$, where e' is any of the immediate subterms of e. Otherwise, v does not occurs in e.

Table 3.1: Type System for System R

$$\frac{}{\Gamma \vdash \text{zero} : \text{nat}} \qquad \frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{suc } e : \text{nat}} \qquad \frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \qquad \frac{\Gamma, v : \tau \vdash e : \tau'}{\Gamma \vdash \lambda v : \tau . e \ : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \qquad \frac{\Gamma \vdash e_1 : \text{nat} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{match } e_1\ e_2\ (v,\ e_3) \ : \ \tau}$$

$$\frac{\Gamma, v : \tau_1 \to \tau_2 \vdash e : \tau_1 \to \tau_2 \qquad v \in_v e}{\Gamma \vdash \mu v . e : \tau_1 \to \tau_2}$$

Requiring an occurrence of a variable with the same type as the function is a minimum requirement we settle for it to be considered recursive. By design, mutual recursion cannot be expressed. The decision to establish two levels of expressions is to avoid nested recursive expressions, which would only complicate things for the unrolling procedure.

In Agda, System R is defined in the intrisically typed approach with well-scoped De Bruijn indices, and can be found in files R.Syntax and R.Syntax.Base. The definition of variable occurrence is in file R.Syntax.Properties. Types and Contexts are defined in Common.Type and Common.Context. Theorem 1 shows that System R is a subset of STLC with recursion, using STLC definition from Wadler [41].

Theorem 1. If e is an expression in System R, then exists a correspondent expression in simply typed λ-calculus with natural numbers and recursion.

Proof. By induction on the structure of e. Bottom level terms and fixpoint definitions (recursive functions) are straightforward. Applications involving recursive functions are translated into a conventional application.
See function `⊩-to-IR` in file R.Syntax.IR.Properties. □

3.2 Unrolling

For loops in imperative languages, we can put two or more copies of the body in a row to increase performance, specially in loops with small bodies where most of the time would otherwise be spent by changing the counter and testing the condition [9]. Performance of recursive functions in imperative languages can also be increased by using this technique [35]. Our approach is unrolling recursive functions in functional languages, not to increase performance but to substitute the remaining recursive occurrences with an error, creating a non-recursive function that still computes something that the original would. An essential step of unrolling is **inlining**, which means to replace a function call (the occurrence of the function's name) by its body and it is a common procedure in the implementations of compilers [9]. Implementing inlining for System R requires defining some concepts first.

3.2.1 Embeddings

An **Order Preserving Embedding** [21] (Definition 2) is a binary relation between contexts, where $\Gamma \subseteq \Delta$ means that every element of Γ is also in Δ, and the elements they have in common are in the same order. Alternatively, we can say that context Γ can be obtained from Δ by dropping zero or more elements. Order preserving embedding is a preorder relation (Theorem 2). Every context is an order preserving embedding of the empty context, since you can *drop* every element until they're both empty (see `⊆-∅` on file Common.Context). This relation is sometimes abbreviated as OPE.

Definition 2. Context Δ is an Order Preserving Embedding of context Γ, represented by $\Delta \supseteq \Gamma$ or $\Gamma \subseteq \Delta$, if:

- Both contexts are empty, i.e., $\Gamma = \varnothing$ and $\Delta = \varnothing$.
- $\Gamma = \Gamma', t$; $\Delta = \Delta', t$ and $\Gamma' \subseteq \Delta'$, i.e., both contexts are non-empty, their heads are equal and their tails are also in a order preserving embedding relation. This case is called a *keep*.
- $\Delta = \Delta', t$ and $\Gamma \subseteq \Delta'$, i.e., Δ is non-empty and Γ is in an order preserving embedding relation with Δ's tail. This case is called a *drop*.

Theorem 2. Relation $\subseteq$ is reflexive and transitive.

Proof. Reflexivity goes by induction on the structure of the context.
See `⊆-refl` on file Common.Context.
Transitivity goes by induction. The base case is when at least the first OPE fits in the first case of the definition. In the inductive steps, when both fit in the second case, the conclusion also fits in it. The conclusion fits in the third case otherwise.
See `⊆-trans` on file Common.Context □

With OPE's we can define a substitution of contexts, which is essential for inlining expressions in our intrisically-typed approach.

3.2.2 Substitutions

In typed λ-calculi a *substitution* usually refers to the capture-avoiding substitution of terms when performing β-reduction. It could also refer to the substitution lemma [32], useful as an intermediate result for proving type soundness. But there is another sense of substitution (Definition 3), in that we can keep an expression and change its context to a weaker one (with more information), and change its typing derivation[1] accordingly. The OPE relation is the precise definition of a "weaker" context and it makes this substitution possible (Lemma 1).

Definition 3. A context substitution of a well-typed term $\Gamma \vdash \tau$ means changing the context Γ by some Γ' with at least the same information.

Lemma 1. Given contexts Γ and Δ, and type τ, if $\Gamma \vdash \tau$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash \tau$.

Proof. By induction on the structure of the term $\Gamma \vdash \tau$. Variable substitution goes by induction on the structure of the OPE, incrementing the variable's De Bruijn index accordingly. The index increase also happens for abstractions and the third expression of pattern matchings.
See `⊆-subs` at file R.Syntax.Properties and `∈-subs` at file Common.Context. □

One interesting property of this substitution of contexts is that it preserves variable occurrences (Lemma 2), and will not insert occurrences of variables that did not occur with the previous context. This property is useful due to our requirement for recursive functions to have an occurrence of something of its same type. In this way, substitution of contexts will not turn recursive functions into non-recursive. We name this operation as *occurrence-preserving substitution*. Notice that in an extrinsically-typed approach, it is trivial that the term would not change, and our interest would be in adjusting the typing derivation.

Lemma 2. Every variable v that occurs in a term $\Gamma \vdash \tau$ also occurs in every correspondent context-substituted term $\Delta \vdash \tau$.

Proof. By induction on the structure of the variable ocurrence. A *keep* is made on the cases that extend the context (abstractions and pattern matchings).
See `called-in` and `call-subs` at file R.Syntax.Properties. □

[1]The text presents the language in an extrinsically-typed style, so $\Gamma \vdash t$ is a notation for a typing derivation. The formalization in Agda uses the intrisically-typed approach, so the same notation represents a term instead. Througout the text, we interchangeably refer to the notation as terms and typing derivations. In Agda, this substitution operation changes the term.

3.2.3 Inlining and Expansion

Inlining means to substitute a reference to a term or function by its body and it is largely used by compiler engineers [9]. We want that inlinings have the same occurrence-preserving property we have for substitutions. Lemma 3 shows that we can inline a term that is recursive[2] with itself and preserve the occurrence of the function's name. We name this operation as *occurrence-preserving inlining.*

Lemma 3. Let Γ and Δ be contexts, and consider the terms $\Gamma \vdash t_1 : \tau_1$ and $\Delta \vdash t_2 : \tau_2$. If $v : \tau_2$ occurs in both t_1 and t_2 and $\Delta \subseteq \Gamma$, then there exists a term $\Gamma \vdash t_3 : \tau_1$, obtained by inlining t_2 into t_1 in the place of v, such that $v : \tau_2$ occurs in t_3.

Proof. By induction on the structure of $v \in_v t_1$. Variable occurrences are preserved through Lemma 2. A *drop* is needed for `abs` and `match`, since they extend the context. See `inline` at file R.Syntax.Unrolling. □

Before proving that this method can be repeatedly applied to create a function that is as expanded as we want, we define the reflexive and transitive closure relation between expressions and their *occurrence-preserving* inlined forms in Definition 4 (see `expanded-to-in-steps` at file R.Syntax.Unrolling). The inlining is always done with respect to a specific expression, rather than inlining expressions into themselves. This decision avoids a super exponential growth of the function's code, following Rugina and Rinard [35]. In this way, each expansion step only grows the function by one copy of its original body.

Definition 4. Expression e_2 is the ocurrence-preserving expansion of e_1 in n steps with respect to expression e, represented by $e_1 \rightarrowtail^e_n e_2$, if e_2 is the result of n inlinings of e in e_1. More specifically:

- $e \rightarrowtail^e_0 e$, i.e., every expression expands to itself in zero steps.
- $e_1 \rightarrowtail^e_1 e_2$, where e_2 is the occurrence-preserving inlining of e in e_1. Inlining is one step of expansion.
- If $e_1 \rightarrowtail^e_{n_1} e_2$ and $e_2 \rightarrowtail^e_{n_2} e_3$, then $e_1 \rightarrowtail^e_{n_1+n_2} e_3$.

With all the results so far, we now show in Theorem 3 that every recursive expression of System R can be finitely expanded in n steps, for every $n \in \mathbb{N}$.

Theorem 3. For every $n \in \mathbb{N}$ and every expression $e : \tau$ that contains the occurrence of some variable $v : \tau$, there exists an expression e' such that $e \rightarrowtail^e_n e'$.

[2]From now on we omit the "potentially" for a smoother reading.

Proof. By induction on n. There are base cases for $n = 0$, in which we simply call for reflexivity, and for $n = 1$, where one inlining is performed. The inductive step applies the transitivity property to an inlining step.
See `expansion` at file R.Syntax.Unrolling. □

The expansion of a recursive function is still recursive, because the occurrences of the function's name are preserved through all steps of the process. We now need to eliminate those remaining occurrences in order to create a non-recursive version. This elimination is made by translating System R terms into the strongly normalizing System L.

3.3 System L

This language is intended to be exactly like System R minus recursion. But we have no way to guarantee that any recursive expression of System R will not loop forever. So, the idea is to include a construction into System L that models a forced termination. The expansion of System R's expressions makes use of a natural number that acts as a factor to guide the amount of expansions. We call this factor a **fuel**, inspired by Petrol Semantics [24]. The forced termination of System L is called **out of fuel**, indicating the ending of the pattern matchings' nesting, and is represented in the syntax by `error`. Grammar 3.2 presents the complete syntax of System L.

Grammar 3.2: Syntax of System L

$$\begin{array}{rcl}
\langle \tau \rangle & ::= & \texttt{nat} \mid \langle \tau \rangle \to \langle \tau \rangle \\
\langle e \rangle & ::= & v \mid \texttt{error} \mid \texttt{zero} \mid \texttt{suc}\ \langle e \rangle \\
& \mid & \lambda v{:}\langle \tau \rangle.\langle e \rangle \mid \langle e \rangle\ \langle e \rangle \mid \texttt{match}\ \langle e \rangle\ \langle e \rangle\ (v,\ \langle e \rangle)
\end{array}$$

We can see System L as the simply typed λ-calculus with natural numbers, pattern matching over numbers, and an additional basic construction whose type can be any type. The type system is described in Table 3.2.

It is possible to eliminate any variable occurrence from an expression, and the method is described in Definition 5. This is useful to eliminate the occurrences of the function's name and make the expression non-recursive. Lemma 4 shows the correctness of this method, in the sense the variable does not occur in the resulting expression.

Definition 5. The occurrence elimination of a variable $v : \tau'$ from a bottom level term $\Gamma \vdash \tau$ in System R is the correspondent term in System L where v occurrences are replaced by `error`.
See `call-elimination` at file Transform.Translation.

Table 3.2: Type System for System L

$$\frac{\tau \text{ is a type}}{\Gamma \vdash \text{error} \;:\tau} \qquad \frac{}{\Gamma \vdash \text{zero} : \text{nat}} \qquad \frac{\Gamma \vdash e : \text{nat}}{\Gamma \vdash \text{suc } e : \text{nat}} \qquad \frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau}$$

$$\frac{\Gamma, v : \tau \vdash e : \tau'}{\Gamma \vdash \lambda v : \tau.e \;:\tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \text{nat} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{match } e_1\; e_2\; (v,\, e_3) : \tau}$$

Lemma 4. For every t that is a bottom level term in System R, if $v : \tau$ is a variable that occurs in t, then v does not occur in the System L expression that resulted from the occurrence elimination of v from t.

Proof. By induction on the structure of the variable occurrence. The evidence for occurrence is transformed into evidence of non-occurrence with the use of the datatypes `called-in` and `not-called-in`. It is straightforward since it is a form of translation. See `no-call-in-elimination` at file Transform.Translation. □

We can translate closed expressions from System R to System L, preserving the type of the expression, as shown in Theorem 4. The resulting expression is non-recursive, since there are no more functions with occurrences of their own names. The composition of expansion and translation results in the algorithm for compiling general recursive functions into finite pattern matching, as shown in Corollary 1. It is important to highlight that, in Agda, the expression inside a `rec` will not contain an occurrence of its own name, but this name is still present in the context. Thus, the `rec` cannot be mapped to the occurrence-eliminated version of the expression $e : \tau_1 \to \tau_2$ it contains, because their contexts are not the same. The solution is to map it to an application of an abstraction with body e over a term t of type τ_1. For this matter, a trivial term is used as a value for t, that will simply be discarded due to Lemma 4. Trivial terms are constructed following Definition 6.

Definition 6. A trivial term of type τ is:

- `zero`, if $\tau \equiv$ `nat`.
- `abs` $v : \tau_1.e$, if $\tau \equiv \tau_1 \to \tau_2$, where e is a trivial term of type τ_2.

Theorem 4. For every closed term t of System R, there exists a closed term t' of System L, such that t and t' have the same type.

Proof. By induction on the structure of the term. All terms are translated to their direct correspondents, with μ (`rec` in Agda) being translated to an application of an abstraction of the occurrence-eliminated expression over a trivial term.
See `call-elimination`, `closure` and `translate` at file Transform.Translation. □

Corollary 1. For every closed term t of System R and every $n \in \mathbb{N}$, there exists a closed term of System L that corresponds to the expansion of t in n steps.

Proof. Compose unrolling and translation.
See `transform` at file Transform.Translation. □

In this chapter we presented the definitions and intermediate properties that allowed us to prove our first desired property: our unrolling algorithm is a total function, i.e., it always terminates and is defined for all well-typed programs of System R. The other properties are about the preservation of the programs' behaviour through the transformation process. In the following chapter, we explore those semantic properties.

4 Semantics

The syntactical transformations shown in Chapter 3 mean nothing if we have no evidence that the behavior of the expressions are preserved. One could simply map every expression of System R to the same expression of System L, and that would still be syntactically valid and probably maintain several properties. What brings confidence to our method is the justification that the expressions compute "the same thing" and that the resulting expression always terminates. For this matter, we present the semantics of both System R and System L and explore properties about them.

4.1 System R

System R's syntax is a subset of the STLC as defined by Wadler & Kokke in [41]. The difference between these two systems is that STLC does not define two levels for syntactic constructions, thus recursive functions can appear anywhere, including inside another recursive function. In this way, we elaborate System R into the mentioned STLC. Its small step semantics, via the evaluation relation $\longrightarrow$, is reproduced in Table 4.1. We define $\longrightarrow^*$ to be the reflexive and transitive closure of $\longrightarrow$.

Table 4.1: Call-by-value, small step semantics of Wadler's STLC

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'} \qquad \frac{e \longrightarrow e'}{\text{suc } e \longrightarrow \text{suc } e'}$$

$$\frac{}{(\lambda x : \tau . e_1)\ v_2 \longrightarrow [x \mapsto v_2]e_1} \qquad \frac{}{\mu v.e \longrightarrow [v \mapsto \mu v.e]e}$$

$$\frac{}{\text{match zero } e_2\ (x, e_3) \longrightarrow e_2} \qquad \frac{}{\text{match (suc } v_1)\ e_2\ (x, e_3) \longrightarrow [x \mapsto v_1]e_3}$$

$$\frac{e_1 \longrightarrow e_1'}{\text{match } e_1\ e_2\ (x, e_3) \longrightarrow \text{match } e_1'\ e_2\ (x, e_3)}$$

Example 4.1 shows a program that is an application of a function that adds up two natural numbers over the arguments 3 and 4. With an exhausting use of the semantic rules, we can see that this program is valid and evaluates to `suc suc suc suc suc suc suc zero`, i.e., it evaluates to 7.

Example 4.1: Program that sums 3 and 4.

```
(rec "sum" (abs "x" : nat (abs "y" : nat (
    match "x"
      "y"
      ("w", ("sum" "w") (suc "y"))))))
(suc suc suc zero)
(suc suc suc suc zero)
```

With semantics defined we show in Theorem 5 that progress and preservation holds for System R. But the only guarantee of termination is due to the fueling of the evaluation relation (see `⊩-eval` at file R.Semantics).

Theorem 5. If $e : \tau$ is a closed term in System R, then e is a value or there exists a term $e' : \tau$ such that $e \longrightarrow e'$.

Proof. By induction on the structure of e. This proof relies on the datatypes that model the semantic relation and its closure. The `done` construction indicates a normal form while the `step` construction with a semantic rule indicates the progress of the evaluation. The proof is a mapping between the possible states of the evaluation to normal forms or semantic rules, and it is present in detail in Wadler's work [41].
See `progress` at file R.Semantics. □

An alternative semantic formalization is by a **definitional interpreter** [7]. That means mapping the constructions of the language to the objects of the proof assistant, in a denotational semantics style. However, to map System R constructions to Agda objects, the function must be "fueled" since there is no guarantee of termination, and Agda requires everything to terminate. See `∅-eval` at file R.Semantics.Definitional. One main advantage of this approach is using the capture-avoiding substitution already implemented in the language rather than writing one from scratch.

4.2 System L

The small step semantics of System L, shown in Table 4.2, is similar to System R's. The differences lie in the drop of the rule for recursion and in the inclusion of the rules for handling `error`. System L's semantics satisfies progress and preservation as well, as shown in Theorem 6. But unlike System R, System L is strongly normalizing (Theorem 7), that is, every term finitely evaluates to a normal form[1].

[1]In System R, there is no problem in defining all normal forms as values. In System L, the error construction is a normal form, but the semantic relation $\longrightarrow$ becomes non-deterministic if we define it as a value. The details of this problem can be found in [32].

Table 4.2: Call-by-value, small step semantics of System L

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v_1\ e_2 \longrightarrow v_1\ e_2'} \qquad \frac{e \longrightarrow e'}{\text{suc}\ e \longrightarrow \text{suc}\ e'}$$

$$\frac{}{(\lambda x : \tau . e_1)\ v_2 \longrightarrow [x \mapsto v_2]e_1} \qquad \frac{e_1 \longrightarrow e_1'}{\text{match}\ e_1\ e_2\ (x, e_3) \longrightarrow \text{match}\ e_1'\ e_2\ (x, e_3)}$$

$$\frac{}{\text{match zero}\ e_2\ (x, e_3) \longrightarrow e_2} \qquad \frac{}{\text{match (suc}\ v_1)\ e_2\ (x, e_3) \longrightarrow [x \mapsto v_1]e_3}$$

$$\frac{}{\text{error}\ e \longrightarrow \text{error}} \qquad \frac{}{v\ \text{error} \longrightarrow \text{error}} \qquad \frac{}{\text{suc error} \longrightarrow \text{error}}$$

$$\frac{}{\text{match error}\ e_2\ (x, e_3) \longrightarrow \text{error}}$$

Theorem 6. If $e : \tau$ is a closed term in System L, then either e is a normal form or there exists a term $e' : \tau$ such that $e \longrightarrow e'$.

Proof. By induction on the structure of e. The idea is identical to the proof for System R. See `progress` at file L.Semantics. □

Theorem 7. If $e : \tau$ is a closed term in System L, then exists a normal form $e' : \tau$, either value or error, such that $e \longrightarrow^* e'$.

Proof. By definitional interpreter. Types from System L are mapped to a sum type in Agda. The idea is to include a $\bot$ value in every type, to model `error`. Contexts are modelled as a list of values, and the interpreter maps System L constructions to Agda constructions. The evaluation of closed terms is the interpretation of terms starting with an empty list of values (an empty context).
See `∅-eval` at file L.Semantics.Definitional. □

Theorem 7 proves our second desired property: translating a program to System L is a guarantee that the program will halt, even if it has to stop with the *out of fuel* error. It only remains to justify that the original program and the output of our algorithm have the same behavior when the former terminates and enough fuel is provided in the creation of the latter. To accomplish this, we define random term generation and run property-based tests, that are descripted in the following sections.

4.3 Term Generation

We define, for our generation procedure, a superset of types, in which our base type $\mathbb{N}$ is indexed by natural numbers. Those indices are related via subtyping, such that $\mathbb{N}^x <: \mathbb{N}^y$ whenever $x \leq y$. Functions are related in the usual way, being covariant in return type and contravariant in the argument type. Inspired in type-based termination [10], the role of these indices is to guide the depth of the generation of constants, and to act as an upper limit to the use of recursion. The motivation is that we wish to generate terminating programs, avoiding discarded cases by QuickCheck.

Our generation judgment, represented by the notation $\Gamma; d; r; \tau \rightsquigarrow e$, means that expression e of type τ can be generated from context Γ, given nonnegative integer limits d for expression depth and r for type indices. The relation $\rightsquigarrow$ is annotated with a letter to distinguish the form of the expressions it generates in each rule. This decision helps to control which rules can be used in each scenario, which is very useful to guarantee the termination of the generation procedure. Due to the complexity of the rules, they are presented separately.

In Table 4.3 we have our first rules: the generation of zero and terms that are variables. A zero can be generated in every scenario where a $\mathbb{N}$ is expected, regardless of the index. Variables can be picked from a non empty list of candidates. Those candidates are the variables from the context that are a subtype of the expected type. The notation $\xi(xs)$ means the selection of a random element from a non-empty list xs.

Table 4.3: Generation rules for zeros and variables

$$\frac{}{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_z zero}\ \{zero\} \qquad \frac{cs = \{v \,|\, v : \tau' \in \Gamma \wedge \tau' <: \tau\} \quad cs \neq \emptyset}{\Gamma; d; r; \tau \rightsquigarrow_v \xi(cs)}\ \{var\}$$

The successor construction, shown in Table 4.4, can be also generated in every scenario where a $\mathbb{N}^x$ is expected, as long as index x is not already 0. Its argument can be a zero, a variable or another successor construction. The index limit is decreased for the generation of its subterm, and thus we can be sure that it will eventually reach 0 and terminate. It is useful not to include other rules as possible subterms for this construction, because we have a constraint at the generation of pattern matchings when inside recursive definitions. This constraint is that the first branch must be a constant. The notation $\psi, \phi, \dots = \xi(\{a, b, c, d, \dots\})$ means that rules annotated with ψ, ϕ, etc., are an alias to rules annotated with any letter in the list inside ξ, and are meant to be selected randomly for each letter on the left side.

Table 4.4: Generation rule for sucessors

$$\frac{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\psi e \quad \psi = \xi(\{z, v, s\})}{\Gamma; d; r+1; \mathbb{N}^{x+1} \rightsquigarrow_s suc\, e}\ \{suc\}$$

Abstractions need to generate an expression of the return type before inserting a λ. A function *fresh* that generates a variable name that is not present in the given context is defined. In this way, we can avoid capturing of names when generating function definitions.

Notice that in intrisically-typed implementations, this function is not necessary. The rule for generating abstractions is in Table 4.5. The operator $\lfloor \tau \rfloor$ erases the index from indexed type τ.

Table 4.5: Generation rule for abstractions

$$\frac{v = \mathit{fresh}(\Gamma) \quad \Gamma, v : \tau_1; d; r; \tau_2 \leadsto_\psi e \quad \psi = \xi(\{z, v, s, a\})}{\Gamma; d; r; \tau_1 \to \tau_2 \leadsto_a \lambda v : \lfloor \tau_1 \rfloor . e} \, \{abs\}$$

Applications of type τ need the generation of an abstraction that results in τ when applied to a proper argument. This argument needs to be generated as well. The rule for applications is in Table 4.6. It uses the operator Θ that generates a type given the limits d and r, and never associates to the left. In this way, we can avoid the generation of a higher-order function and further complexity of the generation procedure. The implementation of Θ is the random selection of one out of two more specific versions: $\Theta^{\to}$ that generates only function types and $\Theta^{\mathbb{N}}$ that generates only indexed $\mathbb{N}$ types. It is important to highlight that if limits d and r are high enough (greater than 1), both base and functional types can be generated. Otherwise, it will force the generation of a base type ($\mathbb{N}$ is our only base type here). Generating applications will cut the d limit to half for generating its subterms, thus its termination is certain.

Table 4.6: Operator Θ and generation rule for applications

$$\begin{aligned}
\Theta^{\mathbb{N}}(r) &= \mathbb{N}^{\xi(\{1,\ldots,r\})} \\
\Theta^{\to}(0, r) &= (\Theta^{\mathbb{N}}(r)) \to (\Theta^{\mathbb{N}}(r)) \\
\Theta^{\to}(2d, r) &= (\Theta^{\mathbb{N}}(r)) \to (\xi(\{\Theta^{\mathbb{N}}(r), \Theta^{\to}(d, r)\})) \\
\Theta(d, r) &= \xi(\{\Theta^{\mathbb{N}}(r), \Theta^{\to}(d, r)\})
\end{aligned}$$

$$\frac{\tau' = \Theta(d, r) \quad \Gamma; d; r; \tau' \to \tau \leadsto_\phi e_1 \quad \Gamma; d; r; \tau' \leadsto_\psi e_2 \quad \phi, \psi = \xi(\{z, v, s, a\})}{\Gamma; 2d; r; \tau \leadsto_a e_1 \, e_2} \, \{app\}$$

Match constructions need to generate a term of type $\mathbb{N}$ and two terms of the expected type. The rule is in Table 4.7. For the generation of the third term e_3, the limit r is decreased and a fresh variable is added to context with a decreased index as well. The operator $_\downarrow$ decreases the index of the type by 1. If it is a function, only the leftmost atom is decreased. As in the generation of applications, matches can only be introduced if the limit d is greater than 1.

The generation of recursive functions and applications involving them require more caution. Generating a recursive function needs the generation of a proper body, which

Table 4.7: Generation rule for pattern matchings

$$\dfrac{\begin{array}{l} v \quad = \quad \mathit{fresh}(\Gamma) \quad \Gamma; d; r; \mathbb{N}^r \rightsquigarrow_\phi e_1 \\ \phi, \psi, \rho = \xi(\{z, v, s, a\}) \quad \Gamma; d; r; \tau \rightsquigarrow_\psi e_2 \quad \Gamma, v : \mathbb{N}^{r_\downarrow}; d; r_\downarrow; \tau \rightsquigarrow_\rho e_3 \end{array}}{\Gamma; 2d; r; \tau \rightsquigarrow_a \mathit{match}\, e_1\, e_2\, (v, e_3)} \{\mathit{match}\}$$

means it has to be well-typed and must terminate. Generating non-terminating terms would lead to lots of discarded cases when QuickChecking, because those terms would not satisfy the premise of the property. The simplest way of guaranteeing this is inserting abstractions until we are left with the generation of a $\mathbb{N}$, and then generate a pattern matching. This match will generate a constant natural number on its first branch and it will allow a recursive call (an occurrence of the function's name) on the second. The expression being generated for the second branch needs to be modified to make sure that, if there is a recursive call, then its first argument will be the predecessor of the matched expression. We call this process *standardization*. The generation rules and the standardization are in Table 4.8, where *bottom* is a function that returns the first variable name added to the context, i.e., the bottom of the scope stack; and *top* is a function that returns the last variable name added to the context, i.e., the top of the scope stack.

Applications involving recursive definitions are more useful if they yield a number as the final value, and so we force this to happen. After generating a functional type and a recursive definition of this type, we generate proper arguments and apply the function over them. For this, we build a list of arguments and extend the judgment to contain them. We must build the applications of the outer arguments first, applying the recursive definition to the first argument as the innermost application. For this, we need to reverse our list of arguments. This rule is formally described in Table 4.9.

Finally, generating a System R expression means to generate a type and generate a term accordingly, as described in Table 4.10. If we want only top terms of System R, then it is enough to use $\psi = g$ and $\phi = f$. But it is interesting to allow the generation of bottom level terms when testing properties, since recursive functions are optional in real-life implementations. In that case, we can let $\psi = \xi(\{z, v, s, a, g\})$ and $\phi = \xi(\{a, f\})$.

We are aware of the verbosity of the rules for term generation. Although we based our approach on elegant formalizations in the literature, as the ones seen in [30, 18], our interest in guaranteeing termination—of both the generation procedure and the generated term—greatly increases the complexity of the process. A more elegant formalization should be possible by better exploring the properties of terminating term generation, which we leave as a possibility for future work.

Table 4.8: Generation rules for recursive definitions and standardization function

$$\begin{aligned}
std(e_1\ e_2, v_1, v_2) &= \begin{cases} e_1\ v_2 & e_1 \equiv v_1 \\ (std(e_1, v_1, v_2))\ (std(e_2, v_1, v_2)) & otherwise \end{cases} \\
std(\lambda v : \tau.e, v_1, v_2) &= \lambda v : \tau.(std(e, v_1, v_2)) \\
std(match\ e_1\ e_2\ (v, e_3), v_1, v_2) &= match\ (std(e_1, v_1, v_2))\ (std(e_2, v_1, v_2)) \\
&\quad (v, std(e_3, v_1, v_2)) \\
std(e, \_, \_) &= e \\
stdz(e, \Gamma) &= std(e, bottom(\Gamma), top(\Gamma))
\end{aligned}$$

$$\frac{v = fresh(\Gamma) \quad \Gamma, v : (\tau_1 \to \tau_2)_\downarrow; d; r_\downarrow; \tau_1 \to \tau_2 \rightsquigarrow_b e}{\Gamma; 2d; r+1; \tau_1 \to \tau_2 \rightsquigarrow_f rec\, v : \lfloor \tau_1 \to \tau_2 \rfloor.e}\ \{rec\}$$

$$\frac{\begin{array}{lll} v = fresh(\Gamma) & & \\ \rho = \xi(\{z, v, s, a\}) & \Gamma; d; r; (\mathbb{N}^x)_\downarrow \rightsquigarrow_\psi e_1 & \Gamma, v : (\mathbb{N}^x)\downarrow; d; r; \mathbb{N}^x \rightsquigarrow_\rho e_3' \\ \phi, \psi = \xi(\{z, v, s\}) & \Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\phi e_2 & e_3 = stdz(e_3'; \Gamma, v : (\mathbb{N}^x)_\downarrow) \end{array}}{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_b match\ e_1\ e_2\ (v, e_3)}\ \{buildBody1\}$$

$$\frac{v = fresh(\Gamma) \quad \Gamma, v : \tau_1; d; r; \tau_2 \rightsquigarrow_b e}{\Gamma; d; r; \tau_1 \to \tau_2 \rightsquigarrow_b \lambda v : \lfloor \tau_1 \rfloor.e}\ \{buildBody2\}$$

4.4 QuickChecking Properties

The property-based tests we apply require essentially two things: the term generation procedure and the algorithms involved in the properties. The former was defined in the previous section, and the latter can be achieved by constructing a compiler. Ringell is our proof-of-concept interpreter[2] written in Haskell, featuring the unrolling technique described in this work. Its source code is available in https://github.com/mayconamaro/ringell, which contains some example programs as well.

The interpreter has two modes based on the number of arguments passed to it. The default mode expects a filename containing a System R program, which will be parsed and interpreted following the defined semantics. The other mode expects a filename and a natural number, which will be used as the fuel value to unroll the program and translate it to System L before interpreting it. Given our properties so far, the second

[2]Compilers are often seen as programs that transform a source code file into executable machine binaries, while interpreters are seen as programs that execute the instructions as they are parsed. From our point of view, the syntactic transformations over abstract syntax trees are processes of compilation, and thus we also refer to Ringell as a compiler.

Table 4.9: Generation rules for applications of recursive definitions

$$\frac{\tau = \Theta^{\rightarrow}(d, r) \quad \Gamma; d; r; \tau \rightsquigarrow_f e' \quad \Gamma; d; r; e'; \mathit{revargs}(\tau) \rightsquigarrow_c e}{\Gamma; 2d; r; \mathbb{N}^x \rightsquigarrow_g e} \{\mathit{apprec}\}$$

$$\begin{aligned} \mathit{args}(\mathbb{N}^x) &= [] \\ \mathit{args}(\mathbb{N}^x \rightarrow \tau_2) &= \mathbb{N}^x :: (\mathit{args}(\tau_2)) \\ \mathit{revargs} &= \mathit{reverse} \circ \mathit{args} \end{aligned}$$

$$\frac{\begin{array}{c}\psi = \xi(\{z, v, s, a\}) \\ \Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\psi e'\end{array}}{\Gamma; d; r; e; [\mathbb{N}^x] \rightsquigarrow_c e\ e'} \{bdA1\} \qquad \frac{\Gamma; d; r; e; ts \rightsquigarrow_c e_1 \quad \begin{array}{c}\psi = \xi(\{z, v, s, a\}) \\ \Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\psi e_2\end{array}}{\Gamma; d; r; e; \mathbb{N}^x :: ts \rightsquigarrow_c e_1\ e_2} \{bdA2\}$$

Table 4.10: Term generation rules for System R

$$\frac{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\psi e}{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow e} \qquad \frac{\Gamma; d; r; \tau_1 \rightarrow \tau_2 \rightsquigarrow_\phi e}{\Gamma; d; r; \tau_1 \rightarrow \tau_2 \rightsquigarrow e}$$

mode is guaranteed to terminate, while the default mode might eventually interpret a program with an infinite loop.

Building confidence around our tests with Ringell require at least two things: a good code coverage and wide variety of case tests. Since we are using property-based testing, the latter is handled by QuickCheck. We set the maximum number of successful test cases as 1000, opposing the default 100. To avoid losing code coverage because of other parts of the interpreter, such as lexical and syntactical analysis, a minimal and test-purpose only version of Ringell was used for tests, and its source code is available at https://github.com/mayconamaro/ringell-properties. The following properties were tested and successfully accepted by QuickCheck:

- Property 1: All generated System R programs are well typed
- Property 2: All generated System R programs terminate
- Property 3: For every generated System R term e of type $\mathbb{N}$, if e terminates with value v doing at most f recursive calls, then $\mathit{transform}(e, f)$ yields v.
- Property 4: For every generated System R term e of type $\mathbb{N}$ and some fuel f, if $\mathit{transform}(e, f)$ yields a value v, then e terminates resulting in v.

Property 2 is only true because of our generation procedure—it is obviously not a property of System R. Properties 3 and 4 are our desired semantic properties for the transformation technique: the output function results in the same value of the original if enough fuel is provided, and the output function producing a value implies that the original function terminates with the same value. Both properties concern only programs of the base type ℕ, because there are two syntatic constructors for programs of functional type in System R (abstractions and recursive definitions) and only one in System L (abstractions). We prefer using propositional equality instead of creating a more complex equivalence relation for this purpose.

The code coverage is shown in Table 4.11. The parts with no full coverage are mostly arguments that were not used in trivial cases (because Haskell uses lazy evaluation). For example, the evaluation of zero has no need for the context information, but it is available as it is in all other cases. The `ExpR` and `ExpL` modules contain the abstract syntax tree data structure for the respective languages, as well as their interpreters largely based on [32]. The module `Unroll`, which practically got full coverage, contains the unrolling, translation and transformation algorithms.

Table 4.11: Code coverage reported by QuickCheck

Module	**Top Level Definitions**	**Alternatives**	**Expressions**
`ExpL`	100%	88%	86%
`ExpR`	88%	94%	89%
`Unroll`	100%	100%	89%
Total	93%	94%	88%

Our strategy to transform recursive functions into finite depth pattern matching, and the proofs and tests we show, are a powerful toolset for creating compilers to targets that will reject programs with recursion. The only downside is forcing the programmer to inform the maximum recursion depth. From our point of view, this is equivalent to the restriction of the bounded loops available today, and thus our strategy does not insert more difficulties than necessary.

5 Conclusion

It is very useful to allow the dynamic change of behavior in computing systems. This level of adaptation is essential to complete several tasks with efficiency. For this matter, programming languages became popular and diverse, being attractive even for systems with stringent security requirements, such as the Linux kernel. In these scenarios, the risk to allow unrestricted programs is too high, so they impose some restrictions on what kinds of programs are accepted. A very common restriction is that the program must terminate, because infinite loops can be used as a tool for denial of service attacks. However, checking for infinite loops in a general manner is impossible, because of the undecidability of the Halting Problem. In this work, we proposed a technique to unroll recursive programs in functional languages. For this task, we defined two core languages: System R and System L. System R is a lambda calculus with naturals, pattern matching and recursion; and System L is the same calculus with no recursion and an additional construction to model forced termination. We defined unrolling for System R and proved that the translation from System R to System L is guaranteed to terminate, and also proved that the execution of System L programs will always terminate as well. We run tests to justify that the programs preserve their behavior through the transformation into System L, unless they originally do not terminate, or if the *fuel* provided to the unrolling process is too low. Our work can be straightforwardly used to create compilers from general purpose functional languages to restricted scenarios such as eBPF or smart contract languages for blockchain networks. Our proof-of-concept interpreter, Ringell, is available to public access. The following resources contain the artifacts produced in this dissertation:

- See `https://github.com/lives-group/terminating-expansion/` for downloadable Agda code of the formalizations.
- See `https://mayconamaro.github.io/dissertation-agda/` for the interactive web version of the Agda formalizations.
- See `https://github.com/mayconamaro/ringell` for the source code of the interpreter, including examples.
- See `https://github.com/mayconamaro/ringell-properties` for the source code of the tests.

Bibliography

[1] Andreas Abel. foetus — termination checker for simple functional programs. Technical report, Ludwigs-Maximilians-University, Munich, 1998.

[2] Andreas Abel. Specification and verification of a formal system for structurally recursive functions. In *International Workshop on Types for Proofs and Programs*, pages 1–20, Berlin, 1999. Springer.

[3] Andreas Abel and Thorsten Altenkirch. A semantical analysis of structural recursion. In *Fourth International Workshop on Termination WST*, pages 24–25, 1999.

[4] Andreas Abel and Thorsten Altenkirch. A predicative analysis of structural recursion. *Journal of Functional Programming*, 12(1):1–41, 2002.

[5] Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and computation*, 163(2):409–470, 2000.

[6] Maycon J. J. Amaro, Samuel S. Feitosa, and Rodrigo G. Ribeiro. A sound strategy to compile general recursion into finite depth pattern matching. In Lucas Lima and Vince Molnár, editors, *Formal Methods: Foundations and Applications*, volume 13768 of *Lecture Notes in Computer Science*, pages 39–54, Cham, 2022. Springer International Publishing.

[7] Nada Amin and Tiark Rompf. Type soundness proofs with definitional interpreters. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 666–679, 2017.

[8] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[9] Andrew Appel. *Modern Compiler Implementation in ML.* Cambridge University Press, 2004.

[10] Gilles Barthe, Benjamin Grégoire, and Colin Riba. A tutorial on type-based termination. In *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*, pages 100–52, 2008.

[11] Gilles Barthe, Benjamin Grégoire, and Colin Riba. Type-based termination with sized products. In *International Workshop on Computer Science Logic*, pages 493–507, Berlin, 2008. Springer.

[12] Alonzo Church. An unsolvable problem of elementary number theory. In *American Journal of Mathematics*, pages 345–363, 1936.

[13] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[14] Jesper Cocx. 1001 representations with binding. `https://jesper.sikanda.be/posts/1001-syntax-representations.html`, 2021.

[15] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*. Elsevier, 1972.

[16] Apostolos Doxiadis and Christos Papadimitriou. *LOGICOMIX: an epic search for truth*. Bloomsbury Publishing USA, 2015.

[17] Gilles Barthe et al. Type-based termination of recursive definitions. *Mathematical structures in computer science*, 14(1):97–141, 2004.

[18] Samuel Feitosa, Rodrigo Ribeiro, and Andre Du Bois. A type-directed algorithm to generate random well-typed Java 8 programs. *Science of Computer Programming*, 196:102494, 2020.

[19] GNU. GCC, the GNU Compiler Collection. `https://gcc.gnu.org/`, 2022.

[20] Gilles Kahn. Natural semantics. In *Annual symposium on theoretical aspects of computer science*, pages 22–39. Springer, 1987.

[21] András Kovács. A machine-checked correctness proof of normalization by evaluation for simply typed lambda calculus. Master's thesis, Eötvös Loránd University, Budapest, 2017.

[22] Ton Chanh Le, Lei Xu, Lin Chen, and Weidong Shi. Proving conditional termination for smart contracts. In *Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts*, BCC '18, page 57–59, New York, NY, USA, 2018. Association for Computing Machinery.

[23] LLVM Project. Clang C Language Family Frontend for LLVM. `https://clang.llvm.org/`, 2022.

[24] Conor McBride. Turing-completeness totally free. In *International Conference on Mathematics of Program Construction*, pages 257–275. Springer, 2015.

[25] Microsoft. Visual Studio Compiler for Windows. `https://visualstudio.microsoft.com/cplusplus/`, 2022.

[26] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[27] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer, 2007.

[28] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers Univeristy of Technology and Göteborg University, Sweden, 2007.

[29] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell.* O'Reilly, Sebastopol, 2008.

[30] Michał Pałka. *Testing an Optimising Compiler by Generating Random Lambda Terms.* Licenciate thesis, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2012.

[31] Daejun Park, Andrei Stefănescu, and Grigore Roşu. KJS: A complete formal semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 346–356, 2015.

[32] Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002.

[33] Gordon Plotkin. LCF considered as a programming language. *Theoretical computer science*, 5(3):223–255, 1977.

[34] Gordon Plotkin. *A structural approach to operational semantics.* Aarhus University, 1981.

[35] Radu Rugina and Martin Rinard. Recursion unrolling for divide and conquer programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 34–48. Springer, 2000.

[36] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*, 2006.

[37] Michael Sipser. *Introduction to the Theory of Computation, 3rd edition.* Cengage Learning, 2012.

[38] Aaron Stump. *Verified functional programming in Agda.* Morgan & Claypool, 2016.

[39] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. In *Proceedings of the London Mathematical Society*, pages 230–265, 1936.

[40] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.

[41] Philip Wadler, Wen Kokke, and Jeremy G. Siek. Programming language foundations in Agda. http://plfa.inf.ed.ac.uk/20.07/, July 2020.