BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

JUST-IN-TIME COMPILATION OF THE DEPENDENTLY-TYPED LAMBDA CALCULUS

JUST-IN-TIME PŘEKLAD ZÁVISLE TYPOVANÉHO LAMBDA KALKULU

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **AUTOR PRÁCE**	**Bc. JAKUB ZÁRYBNICKÝ**
SUPERVISOR **VEDOUCÍ PRÁCE**	**Ing. ONDŘEJ LENGÁL, Ph.D.**

BRNO 2021

Department of Intelligent Systems (DITS)	Academic year 2020/2021

Master's Thesis Specification

24198

Student:	**Zárybnický Jakub, Bc.**
Programme:	Information Technology
Field of study:	Intelligent Systems
Title:	**Just-in-Time Compilation of Dependently-Typed Lambda Calculus**
Category:	Compiler Construction

Assignment:

1. Investigate dependent types, simply-typed and dependently-typed lambda calculus, and their evaluation models (push/enter, eval/apply).
2. Get familiar with the Graal virtual machine and the Truffle language implementation framework.
3. Create a parser and an interpreter for a selected language based on dependently-typed lambda calculus.
4. Propose a method of normalization-by-evaluation for dependent types and implement it for the selected language.
5. Create a just-in-time (JIT) compiler for the language using the Truffle API.
6. Compare the runtime characteristics of the interpreter and the JIT compiler, evaluate the results.

Recommended literature:

- https://www.graalvm.org/
- Löh, Andres, Conor McBride, and Wouter Swierstra. "A tutorial implementation of a dependently typed lambda calculus." Fundamenta Informaticae 21 (2001): 1001-1031.
- Marlow, Simon, and Simon Peyton Jones. "Making a fast curry: push/enter vs. eval/apply for higher-order languages." Journal of Functional Programming 16.4-5 (2006): 415-449.

Requirements for the semestral defence:

- Items 1 to 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Lengál Ondřej, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	November 11, 2020

Abstract

A number of programming languages have managed to greatly improve their performance by replacing their custom runtime system with general platforms that use just-in-time optimizing compilers like GraalVM or RPython. This thesis evaluates whether such a transition would also benefit dependently-typed programming languages or theorem provers.

This thesis introduces the type-theoretic notion of dependent types and the algorithms involved in working with them, specifies a minimal dependently-typed language on the $\lambda\Pi$-calculus, and presents the implementation two interpreters for this language: a simple interpreter written in Kotlin, and a second interpreter, also written in Kotlin, that uses the Truffle language implementation framework on the GraalVM platform, which is a partial evaluation-based just-in-time compiler based on the Java Virtual Machine. The performance of these two interpreters is then compared on a number of normalization and elaboration tasks.

The results are strongly negative, however, the influence of JIT compilation is not noticeable given the large overhead of the JVM platform. This thesis concludes with a number of alternative projects that would use the capabilities of Truffle better.

Abstrakt

Řada programovacích jazyků byla schopna zvýšit svoji rychlost výměnou běhových systémů stavěných na míru za obecné platformy, které pro optimalizaci používají just-in-time překlad, jako jsou GraalVM nebo RPython. V této práci vyhodnocuji, zda je použití takovýchto platforem vhodné i pro jazyky se závislymi typy nebo důkazovými systémy.

Tato práce představuje koncepty λ-kalkulu a teorie typů potřebné pro úvod do závislých typů s relevantními algoritmy, specifikuje malý závisle typovaný jazyk založený na $\lambda\Pi$ kalkulu, a prezentuje dva interpretery tohoto jazyka. Tyto interpretery jsou psané v jazyce Kotlin, první je jednoduchý, psaný ve funkcionálním stylu a druhý používá platformu GraalVM a Truffle. GraalVM je platforma založená na virtuálním stroji Javy (JVM), která přidává just-in-time překladač založený na částečném vyhodnocení (*partial evaluation*) a Truffle je knihovna pro tvorbu programovacích jazyků využívající tento překladač. Závěr práce vyhodnocuje běhové charakteristiky těchto interpreterů na různých zátěžových testech.

Závěry práce jsou ale silně negativní. Vliv JIT překladu není znatelný ani přes snahu optimalizovat běžné algoritmy z teorie typů, které jsou zjevně nevhodné pro platformu JVM. Práce končí návrhy několika navazujících projektů, které by lépe využily možnosti Truffle a které by byly vhodnější pro implementaci závisle typovaných jazyků.

Keywords

Truffle, JVM, just-in-time compilation, compiler construction, dependent types

Klíčová slova

Truffle, Virtuální stroj JVM, just-in-time překlad, tvorba překladačů, závislé typy

Reference

ZÁRYBNICKÝ, Jakub. *Just-in-Time Compilation of the Dependently-Typed Lambda Calculus*. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Systémy používající závislé typy umožňují programátorům vytvářet programy, které jsou zaručeně správné vzhledem k vlastnostem uložených v typech. Tyto systémy je také možné použít pro logické nebo matematické důkazy, nebo pro dokazování správnosti celých systémů. Poslední roky přinesly mnoho pokroků v teorii typů, na níž jsou tyto systémy založené, jako např. kvantitativní nebo homotopické teorie typů. Vysoké nároky na důkazové schopnosti systémů s sebou ale přináší problémy s výkonem, konkrétně rychlostí kontroly typů (*type-checking*, *elaboration*). V této práci zhodnocuji vhodnost just-in-time překladu pro takové systémy, což je jeden z přístupů pro optimalizaci rychlosti výpočetních systémů obecně.

V první části vysvětluji principy typovaného λ-kalkulu, na němž jsou systémy se závislými typy založené, základy teorie typů a konkrétní specifikace několika rozšíření, které poté používám pro specifikaci malého jazyka.

V druhé sekci pokračuji představením algoritmů nezbytných pro práci se závislými typy: *normalization-by-evaluation* a *bidirectional typing*. Tyto implementuji a následně používám pro vytvoření funkčního interpreteru tohoto jazyka.

Třetí sekce představuje detaily platformy GraalVM a knihovny Truffle, které používám pro implementaci druhého interpreteru využívajícího just-in-time překladu, založeném na interpreteru prvním s nezbytnými úpravami. V závěru práce vyhodnocuji běhové charakteristiky těchto dvou interpreterů pomocí sady zátěžových testů.

Výsledky této práce jsou ale silně negativní: charakteristiky implementace běžných algoritmů teorie typů nejsou vhodné pro platformu JVM, a bylo by je zřejmě nutné od základů přepracovat, aby správně využívaly možnosti platformy JVM. i přes negativní výsledky tato práce představuje dobrý výchozí bod pro další práci v oblasti implementace závisle typovaných jazyků i jazyků založených na platformě Truffle.

Just-in-Time Compilation of the Dependently-Typed Lambda Calculus

Declaration

I hereby declare that this Master's thesis was created as an original work by the author under the supervision of Ing. Ondřej Lengál Ph.D.

I have listed all the literary sources, publications, and other sources that were used during the preparation of this thesis.

. . . .
Jakub Zárybnický
May 26, 2021

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál Ph.D. for entertaining a second one of my crazy thesis proposals.

I would like to thank Edward Kmett for suggesting this idea: it has been a challenge.

I would also like to thank my family and friends who supported me when I needed it the most throughout my studies.

Contents

List of Figures

List of Listings

n

Chapter 1

Introduction

Dependently-typed languages allow programmers to write correct-by-construction code, and they can be used as theorem provers, or proof assistants. Programs written in such languages can encode more properties than those written without dependent types, and they are considered to be one approach to formal software verification [40]: a well-known example is the computational proof of the four-color theorem in the theorem prover Coq in 2005 [20].

However, dependently-typed languages rely on their compilers or interpreters to verify, or prove, all invariants (properties) encoded in a program, which involves significant computational effort. When applied to problems or systems on a large-enough scale, type-checking performance becomes the primary obstacle to their use [23, 24]. While many of the performance issues are fundamentally algorithmic [37], a better runtime system would improve the rest.

In recent years, there have been several investigations into the performance of dependently-typed languages: Jason Gross's work on improving the performance of Coq [23, 24]; the work of András Kovács on performant Haskell-based interpreters [31, 33]; Edwin Brady's work on the Idris 2 runtime system based on Chez Scheme [9]. Kovacs, in particular, manages to outperform both Coq and Agda by a large margin in the SmallTT project [32].

However, custom runtime systems or capable optimizing compilers are time-consuming to build and maintain. This thesis seeks to answer the question of whether just-in-time compilation can help to improve the performance of such systems. Moving from custom runtime systems to general language platforms like e.g., the Java Virtual Machine (JVM) or RPython [7], has improved the performance of several dynamic languages: projects like TruffleRuby, FastR, or PyPy. It has allowed these languages to re-use the optimization machinery provided by these platforms, improve their performance, and simplify their runtime systems.

The platform to be evaluated is GraalVM and the Truffle language implementation framework, which reuse and improve upon the JIT capabilities of the Java Virtual Machine. Truffle has been used to implement an improved runtime system for a number of general-purpose languages, the most prominent of which are TruffleRuby and FastR. In both cases, replacing a custom runtime system with a JIT-based one resulted in significant performance improvements[1, 2].

In the final stages of this thesis, I have encountered a single project that attempts to apply JIT compilation to dependent types, there were no other before this one to the best of my knowledge. This project is Enso [38], a visual programming language with multi-language polyglot capabilities, that uses dependent types at its core. I have been able to incorporate and evaluate some of its improvements into the practical parts of this thesis, despite the time constraints, as it would otherwise serve as one of my primary sources.

Other than Enso, closest to this project is Cadenza [30] by Edward Kmett, who also suggested the topic of this thesis. Cadenza is an implementation of the simply-typed lambda calculus on the Truffle framework. While it is unfinished and did not show as promising performance compared to other simply-typed lambda calculus implementations as its author hoped, this thesis applies similar ideas to the dependently-typed lambda calculus, where the presence of compile-time computation should lead to larger gains.

In this thesis, I will evaluate the effect of JIT compilation on the runtime performance of the type-checking (elaboration) of a dependently-typed language based on the typed λ-calculus. In particular, the goal is to investigate the performance of β-normalization and $\beta\eta$-conversion checking. Those are among the main computational tasks in the elaboration process and they are also the tasks can most likely benefit from JIT compilation. The obtained results will be compared between JIT and non-JIT implementations, but also against state-of-the-art proof assistants: Coq, Agda, Idris.

As there are no standard benchmarks for dependently-typed languages, the first task is to design a small, dependently-typed core language, followed by an implementation of an interpreter for this language. Proof assistants use languages based on the typed λ-calculus at their core, so it is a sufficient basis for the goals of this thesis.

Truffle makes it possible to incrementally add JIT compilation to an existing interpreter, using *partial evaluation* to turn slow interpreter code into efficient machine code [55]. During partial evaluation, an interpreter is specialized together with the source code of a program, yielding executable code: parts of the interpreter could be specialized, some optimized, and some could be left off entirely, which often results in performance gains of several orders of magnitude. Having a language interpreter based on Truffle also brings other benefits: seamless interoperability with Java or JVM-based languages [47], or automatically derived language tooling [50]. Regardless of the outcome of the performance evaluation, using Truffle would benefit dependently-typed or experimental languages.

Starting from basic λ-calculus theory and building up to the systems of the λ-cube, we specify the syntax and semantics of a small language (Chapter 2). Continuing with the principles of λ-calculus evaluation, type-checking and elaboration, we implement an interpreter for Montuno in a functional style (Chapter 3). In the second part of the thesis, we evaluate the capabilities offered by Truffle and the peculiarities of Truffle languages, and implement a Truffle interpreter for Montuno (Chapter 4). After designing a set of benchmarks to evaluate the language's performance and discussing then results, we close with a large list of possible follow-up work (Chapter 5).

[1]Unfortunately, there are no officially published benchmarks, but a number of articles claim that TruffleRuby is 10-30x faster than the official C implementation. [46]

[2]FastR is between 50 to 85x faster than GNU R, depending on the source. [19]

Chapter 2

Language specification: λ⋆-calculus with extensions

2.1 Introduction

Proof assistants like Agda or Idris are built around a fundamental principle called the Curry-Howard correspondence that connects type theory and mathematical logic, demonstrated in Figure 2.1. In simplified terms it says that given a language with a self-consistent type system, writing a well-typed program is equivalent to proving its correctness [5]. It is often shown on the correspondence between natural deduction and the simply-typed λ-calculus, as in Figure 2.2. Proof assistants often have a small core language around which they are built: e.g. Coq is built around the Calculus of Inductive Constructions, which is a higher-order typed λ-calculus.

Mathematical logic	Type theory
$\top$ true	$()$ unit type
$\bot$ false	$\emptyset$ empty type
$p \wedge q$ conjunction	$a \times b$ product type
$p \vee q$ disjunction	$a + b$ sum type
$p \Rightarrow q$ implication	$a \to b$ exponential (function) type
$\forall x \in A, p$ universal quantification	$\Pi_{x:A} B(x)$ dependent product type
$\exists x \in A, p$ existential quantification	$\Sigma_{x:A} B(x)$ dependent sum type

Figure 2.1: Curry-Howard correspondence between mathematical logic and type theory

Sequent calculus	$\lambda\!\to$-calculus
$\dfrac{}{\Gamma_1,\alpha,\Gamma_2 \vdash \alpha}$ axiom	$\dfrac{}{\Gamma_1, x : \alpha, \Gamma_2 \vdash x : \alpha}$ variable
$\dfrac{\Gamma,\alpha \vdash \beta}{\Gamma \vdash \alpha \Rightarrow \beta}$ implication introduction	$\dfrac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \Rightarrow \beta}$ abstraction
$\dfrac{\Gamma \vdash \alpha \Rightarrow \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$ modus ponens	$\dfrac{\Gamma \vdash t : \alpha \to \beta \qquad \Gamma \vdash u : \alpha}{\Gamma \vdash tu : \beta}$ application

Figure 2.2: Curry-Howard correspondence between sequent calculus and λ→-calculus

Compared to the type systems in languages like Java, dependent type systems can encode much more information in types. We can see the usual example of a list with a known length in Listing 2.1: the type `Vect` has two parameters, one is the length of the list, the other is the type of its elements. Using such a type we can define safe indexing operators like `head`, which is only applicable to non-empty lists, or `index`, where the index must be in the list (`Fin len`). List concatenation uses arithmetic on the type level, and it is possible to explicitly prove that concatenation preserves list length.

```
data Vect : (len : Nat) -> (elem : Type) -> Type where
  Nil  : Vect Z elem
  (::) : (x : elem) -> (xs : Vect len elem) -> Vect (S len) elem

-- Definitions elided
head : Vect (S len) elem -> elem
index : Fin len -> Vect len elem -> elem
(++) : (xs : Vect m elem) -> (ys : Vect n elem) -> Vect (m + n) elem
proofConcatLength
  : {m, n : Nat} -> {A : Type} -> (xs : Vect n A) -> (ys : Vect m A)
    -> length (xs ++ ys) = length xs + length ys
```

Listing 2.1: Vectors with explicit length in the type[1]

On the other hand, these languages are often restricted in some ways. General Turing-complete languages allow non-terminating programs: non-termination leads to an inconsistent type system, so proof assistants use various ways of keeping the logic sound and consistent. Idris, for example, requires that functions are total and finite. It uses a termination checker, checking that recursive functions use only structural or primitive recursion in order to ensure that type-checking stays decidable.

This chapter aims to introduce the concepts required to specify the syntax and semantics of a small dependently-typed language and use these to produce such a specification, a necessary prerequisite so that we can create interpreters for this language in later chapters. This chapter, however, does not attempt to be a complete reference in the large field of type theory.

[1]Adapted from the Idris `base` library: `https://github.com/idris-lang/Idris-dev/blob/master/libs/base/Data/Vect.idr`

2.2 Languages

Type theories with dependent types are based on typed λ-calculi. To introduce them well, we first need to go through the syntax and semantics of simpler languages, starting with the untyped λ-calculus. This section summarizes the necessary concepts, drawing primarily from Barendregt [6].

2.2.1 λ-calculus

Introduced in the 1930s by Alonzo Church, the untyped λ-calculus was intended as a simple model of computation. It is a Turing-complete system, but only consists of three constructions: abstraction, application, and variables. Figure 2.3 demonstrates two possible notations for the untyped λ-calculus. We will use the standard Church notation, with right associativity of abstraction, and left associativity of application, application binding stronger than abstraction: in any following examples, an expression like $\lambda x.\lambda y.x\ y\ z$ would be parenthesized to $\lambda x.(\lambda y.((x\ y)\ z))$.

$$\begin{array}{rcll} e & ::= & v & \text{variable} \\ & \mid & M\,N & \text{application} \\ & \mid & \lambda v.\,M & \text{abstraction} \end{array}$$

(a) Standard (Church) notation

$$\begin{array}{rcl} e & ::= & v \\ & \mid & (N)\,M \\ & \mid & [v]\,M \end{array}$$

(b) De Bruijn notation

Figure 2.3: Syntax of the λ-calculus using Church and de Bruijn notation

β-reduction A *λ-abstraction* corresponds to the common notion of a function in programming languages. The λ-abstraction $\lambda x.t$ consists of a binder λx, and a body t. *Applying* a λ-abstraction to an argument, e.g., $(\lambda x.x)\ t$, corresponds to evaluating a function, and returns the result of evaluating the body. In λ-calculus, evaluating the body of a function is called *substitution*. It is written $t[x := T]$ and means that all occurrences of the variable x are replaced with the term T inside a term t. The application of a λ-abstraction to a term is called a *β-reduction*, and it is the basic *rewrite rule* of λ-calculus.

$$(\lambda x.\,t)\ u \longrightarrow_{\beta} t[x := u]$$

α-conversion If a variable inside the body of a λ-abstraction is mentioned in any binders, it is called bound, e.g., the variable x is bound in $\lambda x.\lambda y.x$. Conversely, all unbound variables are called *free*, e.g., the variable z is a free variable $\lambda x.z$. When performing a substitution, no free variable can become bound, as the term would change its meaning. We need to ensure that the variable names in the terms do not overlap and rename them if they do. The process of renaming variables is called *α-conversion* (α-renaming) and, in general, may need to happen before each β-reduction.

$$(\lambda x.\,t) \longrightarrow_{\alpha} (\lambda y.\,t[x := y])$$

η-conversion Reducing a λ-abstraction that directly applies its argument to a term or equivalently, rewriting a term in the form of $\lambda x.f\ x$ to f is called *η-reduction*. The opposite rewrite rule, from f to $\lambda x.f\ x$ is $\bar{\eta}$-expansion, and because the rewriting works in both ways, they are together called the *η-conversion*.

$$\lambda x.f\ x \longrightarrow_{\eta} f$$
$$f \longrightarrow_{\bar{\eta}} \lambda x.f\ x$$

δ-reduction β-reduction together with α-renaming are sufficient to specify λ-calculus, but there are three other rewriting rules that we will need later: *δ-reduction* is the replacement of a constant with its definition, e.g., given a constant *id* with the definition $\lambda x.x$, the expression *id t* would be δ-reduced to $(\lambda x.x)\ t$.

$$id\ t \longrightarrow_{\delta} (\lambda x.x)\ t$$

ζ-reduction If we extend the syntax of the language with a *let* − *in* construct that defines a local variable, equivalent process to δ-reduction applied to local variable is called the *ζ-reduction*.

$$let\ id = \lambda x.x\ in\ id\ t \longrightarrow_{\zeta} (\lambda x.x)\ t$$

ι-reduction Later, we will also use other types of objects than just functions. Applying a function that extracts a value from an object is called the *ι-reduction*. In this example, the object is a pair of values, and the function π_1 is a primitive operation that extracts the first value of the pair.

$$\pi_1(a,b) \longrightarrow_{\iota} a$$

Normal form Applying a reduction until a term can no longer be reduced produces a normal form: β-reduction leads to a β-normal form, β- and δ-reductions produce the βδ-normal form. All of these reduction together: applying functions to their arguments, replacing constants and local variables with their definitions, evaluating objects, and α-converting variables if necessary are called βδιζ-reduction, and produce a βδιζ-normal form, or just *normal form* for short. Every term of the λ-calculus has only a single unique normal form (up to α-conversion), according to the Church-Rossier theorem.

$$\begin{aligned}
& \text{let}\ pair = \lambda m.(m,m)\ \text{in}\ \pi_1\ (pair\ (id\ 5)) \\
\longrightarrow_{\zeta}\ & \pi_1\ ((\lambda m.(m,m))\ (id\ 5)) \\
\longrightarrow_{\beta}\ & \pi_1\ (id\ 5, id\ 5) \\
\longrightarrow_{\iota}\ & id\ 5 \\
\longrightarrow_{\delta}\ & (\lambda x.x)\ 5 \\
\longrightarrow_{\beta}\ & 5
\end{aligned}$$

		Reduce under abstraction	
		Yes	**No**
Reduce args	**Yes**	$E := \lambda x.E \mid x\ E_1 ... E_n$ Normal form	$E := \lambda x.e \mid x\ E_1 ... E_n$ Weak normal form
	No	$E := \lambda x.E \mid x\ e_1 ... e_n$ Head normal form	$E := \lambda x.e \mid x\ e_1 ... e_n$ Weak head normal form

Figure 2.4: Normal forms in λ-calculus

Other normal forms A full normal form has all sub-terms of a term fully reduced. There are also other normal forms that differ in the treatment of bodies of λ-abstractions. If we have an expression and repeatedly only use the β-reduction, we end up with a function, or a variable applied to some free variables. These other normal forms specify what happens in such a "stuck" case. In Figure 2.4, e is an arbitrary λ-term and E is a term in the relevant normal form [45]. Closely related to the concept of a normal form are *normalization strategies* that specify the order in which sub-expressions are reduced.

Strong normalization An important property of a model of computation is termination, the question of whether there are expressions for which computation does not stop. In the context of λ-calculus it means whether there are terms, where repeatedly applying rewriting rules does not produce a unique normal form in a finite sequence steps. While for some expressions this may depend on the selected rewriting strategy, the general property is as follows: if for all well-formed terms a there does not exist any infinite sequence of reductions $a \longrightarrow_\beta a' \longrightarrow_\beta a'' \longrightarrow_\beta \cdots$, then such a system is called *strongly normalizing*.

The untyped λ-calculus is not a strongly normalizing system, though, and there are expressions that do not have a normal form. When such expressions are reduced, they do not get smaller, but they *diverge*. The ω combinator:

$$\omega = \lambda x.x\ x$$

is one such example that produces an infinite term. Applying ω to itself produces a divergent term whose reduction cannot terminate:

$$\omega\ \omega \longrightarrow_\delta (\lambda x.x\ x)\omega \longrightarrow_\beta \omega\ \omega$$

The fixed-point function, the Y combinator, is also notable:

$$Y = \lambda f.(\lambda x.f\,(x\ x))\ (\lambda x.f\,(x\ x))$$

This is one possible way of encoding general recursion in λ-calculus, as it reduces by applying f to itself:

$$Y f \longrightarrow_{\delta\beta} f(Y f) \longrightarrow_{\delta\beta} f(f(Y f)) \longrightarrow_{\delta\beta} \cdots$$

This, as we will see in the following chapter, is impossible to encode in the typed λ-calculus without additional extensions.

As simple as λ-calculus may seem, it is a Turing-complete system that can encode logic, arithmetic, or data structures. Some examples include *Church encoding* of booleans, pairs, or natural numbers (Figure 2.5).

$$\begin{aligned} 0 &= \lambda f.\lambda x.\, x \\ 1 &= \lambda f.\lambda x.\, f\ x \end{aligned}$$

(a) Natural numbers

$$\begin{aligned} succ &= \lambda n.\lambda f.\lambda x.f\ (n\,f\ x) \\ plus &= \lambda m.\lambda n.m\ succ\ n \end{aligned}$$

(b) Simple arithmetic

$$\begin{aligned} true &= \lambda x.\lambda y.x \\ false &= \lambda x.\lambda y.y \\ not &= \lambda p.p\,false\ true \\ and &= \lambda p.\lambda q.p\ q\ p \\ ifElse &= \lambda p.\lambda a.\lambda b.p\ a\ b \end{aligned}$$

(c) Logic

$$\begin{aligned} cons &= \lambda f.\lambda x.\lambda y.f\ x\ y \\ fst &= \lambda p.p\ true \\ snd &= \lambda p.p\,false \end{aligned}$$

(d) Pairs

Figure 2.5: Church encoding of various concepts

2.2.2 λ→-calculus

It is often useful to describe the kinds of objects we work with, though. Already, in Figure 2.5 we could see that reading such expressions can get confusing: a boolean is a function of two parameters, whereas a pair is a function of three arguments, of which the first one needs to be a boolean and the other two contents of the pair.

The untyped λ-calculus defines a general model of computation based on functions and function application. Now we will restrict this model using types that describe the values that can be computed with.

The simply typed λ-calculus introduces the concept of types. There are two separate languages: the language of terms, and the language of types. These languages are connected by a *type judgment*, or *type assignment* $x : T$ that asserts that the term x has the type T [25]. It also called the λ→-calculus, as "→" is the connector used in types. We have a set of basic types that are connected into terms using the arrow →, and type annotation or assignment $x : A$.

Church- and Curry-style There are two ways of formalizing the simply-typed λ-calculus: λ→-Church, and λ→-Curry. Church-style is also called system of typed terms, or the explicitly typed λ-calculus, as λ-abstractions directly include the type of the argument in the binder, and we say:

$$\lambda x : A.x : A \to A,$$

or using parentheses to clarify the precedence

$$\lambda(x : A).x : (A \to A).$$

Curry-style is also called the system of typed assignment, or the implicitly typed λ-calculus as we assign types to untyped λ-terms that do not carry type information by themselves, and we say $\lambda x.x : A \to A$ [6].

There are systems that are not expressible in Curry-style, and vice versa. Curry-style is interesting for programming, we want to omit type information; and we will see how to manipulate programs specified in this way in Chapter 3. We will use Church-style in this chapter, but our language will be Curry-style, so that we incorporate elaboration into the interpreter.

Well-typed terms Before we only needed evaluation rules to fully specify the system, but specifying a system with types also requires typing rules that describe what types are allowed. We will also need to distinguish *well-formed terms* from *well-typed terms*: well-formed terms are syntactically valid, whereas well-typed terms also obey the typing rules. Terms that are well-formed but not yet known to be well typed are called *pre-terms* (pre-syntax).

These properties are ensured by type-checking algorithms that will be described in detail in the next chapter. In brief: given a pre-term and a type, *type checking* verifies if the term can be assigned the type; given just a pre-term and no type, *type inference* computes the type of an expression; and finally *type elaboration* is the process of converting a partially specified pre-term into a complete, well-typed term using the previous two [17].

e			(*terms*)
	$:=$	v	variable
	$\mid$	$M\ N$	application
	$\mid$	$\lambda x.\ t$	abstraction
	$\mid$	$x : \tau$	annotation
τ			(*types*)
	$:=$	β	base types
	$\mid$	$\tau \to \tau'$	composite type
Γ			(*typing context*)
	$:=$	$\emptyset$	empty context
	$\mid$	$\Gamma, x : \tau$	type judgement
v			(*values*)
	$:=$	$\lambda x.\ t$	closure

Figure 2.6: λ→-calculus syntax

Types and context The complete syntax of the λ→-calculus is in Figure 2.6. This time, we also include the notion of *values*, which are the result of fully reducing an expression. As there are only functions in this variant, the only possible value is a closure: a partially-evaluated function. Reduction operations are the same as in the untyped lambda calculus, but we will need to add the language of types to the previously specified language of terms.

The language of types consists of a set of *base types* which can consist of, e.g., natural numbers or booleans, and *composite types*, which describe functions between them. We also need a way to store the types of terms that are known, a typing *context*, which consists of a list of *type judgments* in the form $x : T$, which associate variables to their types.

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ (Var)}$$

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B} \text{ (App)}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x : A.b : A \to B} \text{ (Abs)}$$

Figure 2.7: λ→-calculus typing rules

Typing rules The simply-typed λ-calculus can be completely specified by the typing rules in Figure 2.7 [43]. These rules are read similarly to logic proof trees: as an example, the rule **App** can be read as "if we can infer f with the type $A \to B$ and a with the type A from the context Γ, then we can also infer that function application $f\ a$ has the type B". Given these rules and the formula

$$\lambda a : A.\lambda b : B.a : A \to B \to A$$

we can also produce a derivation tree that looks similar to logic proofs and, as mentioned before, its semantics corresponding to the logic formula "if A and B, then A" as per the Curry-Howard equivalence.

$$\cfrac{\cfrac{\cfrac{}{a : A, b : B \vdash a : A}}{a : A \vdash \lambda b : B.a : B \to A}}{\vdash \lambda a : A.\lambda b : B.a : A \to B \to A}$$

We briefly mentioned the problem of termination in the previous section; the simply-typed λ-calculus is strongly normalizing: the reduction of any well-typed term of the λ→-calculus will terminate, and produce a unique normal form. In other words, there is no way of writing a divergent term that is also well-typed; the Y combinator is impossible to type in λ→ and any of the systems in the next chapter [8].

2.2.3 λ-cube

The λ→-calculus restricts the types of arguments to functions; types are static and descriptive. When evaluating a well-typed term, the types can be erased altogether without any effect on the computation. In other words, terms can only depend on other terms.

Generalizations of the λ→-calculus can be organized into a cube called the Barendregt cube, or the λ-cube [6] (Figure 2.8). In λ→ only terms depend on terms, but there are also three other combinations represented by the three dimensions of the cube: types depending on types $(\square, \square)$, or also called type operators; types depending on terms $(\square, \star)$, called *polymorphism*; and terms depending on types $(\star, \square)$, representing *dependent types*.

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda\Pi\omega \\
\lambda 2 \arrow[ur, "{(\square,\square)}"] \arrow[rr] & & \lambda\Pi 2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda\Pi\underline{\omega} \arrow[uu] \\
\lambda\to \arrow[uu, "{(\square,\star)}"] \arrow[ur] \arrow[rr, "{(\star,\square)}"] & & \lambda\Pi \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 2.8: Barendregt cube (also λ-cube)

Sorts To formally describe the cube, we will need to introduce the notion of sorts. In brief,

$$t : T : \star : \square.$$

The meaning of the symbol : is same as before, "x has type y". The type of a term t is a type T, the type of a type T is a kind $*$, and the type of kinds is the sort $\square$. The symbols $\star$ and $\square$ are called *sorts*. As with types, sorts can be connected using arrows, e.g. $(\star \to \star) \to \star$. To contrast the syntaxes of the following languages, the syntax of λ→ is here:

$$\begin{array}{rcllll}
\textit{types} & := & T & \mid & A \to B & \\
\textit{terms} & := & v & \mid & \lambda x : A.t & \mid \; a\, b \\
\textit{values} & := & & & \lambda x : A.t &
\end{array}$$

λ$\underline{\omega}$-calculus Higher-order types or type operators generalize the concepts of functions to the type level, adding λ-abstractions and applications to the language of types.

$$\begin{array}{rclllllll}
\textit{types} & := & T & \mid & A \to B & \mid & A\, B & \mid & \Lambda A.B(a) \\
\textit{terms} & := & v & \mid & \lambda x : A.t & \mid & a\, b & & \\
\textit{values} & := & & & \lambda x : A.t & & & &
\end{array}$$

λ2-calculus The dependency of terms on types adds polymorphic types to the language of types: $\forall X : k.A(X)$, and type abstractions (Λ-abstractions) and applications to the language of terms. It is also called System F, and it is equivalent to propositional logic [6].

$$\begin{array}{rclllllll}
\textit{types} & := & T & \mid & A \to B & \mid & & & \forall A.B \\
\textit{terms} & := & v & \mid & \lambda x : A.t & \mid & a\, b & \mid & \Lambda A.t \\
\textit{values} & := & & & \lambda x : A.t & \mid & & & \Lambda A.t
\end{array}$$

λΠ-calculus Allowing types to depend on terms means that type of a function can depend on its term-level arguments, hence dependent types, represented by the type $\Pi a : A.B(a)$. This dependency is the reason for the name of dependently-typed languages. This system is well-studied as the Logical Framework (LF) [6].

$$\begin{array}{rclccccc} \textit{types} & := & T & \mid & A \to B & \mid & & \Pi a : A.B \\ \textit{terms} & := & v & \mid & \lambda x : A.b & \mid & a\ b \quad \mid & \Pi a : A.b \\ \textit{values} & := & & & \lambda x : A.b & \mid & & \Pi x : A.b \end{array}$$

Pure type system These systems can all be described by one set of typing rules instantiated with a triple (S, A, R). Given the set of sorts $S = \{\star, \square\}$ we can define relations A and R where, for example, $A = \{(\star, \square)\}$ is translated to the axiom $\vdash \star : \square$ by the rule **Start**, and $R = \{(\star, \square)\}$[2] means that a kind can depend on a type using the rule **Product**.

$$\begin{array}{rcll} S & := & \{\star, \square\} & \text{set of sorts} \\ A & \subseteq & S \times S & \text{set of axioms} \\ R & \subseteq & S \times S \times S & \text{set of rules} \end{array}$$

The typing rules in Figure 2.9 apply to all the above-mentioned type systems. The set R exactly corresponds to the dimensions of the λ-cube, so instantiating this type system with $R = \{(\star, \star)\}$ would produce the λ→-calculus, whereas including all the dependencies $R = \{(\star, \star), (\square, \star), (\star, \square), (\square, \square)\}$ produces the λΠω-calculus. If we also consider that the function arrow $A \to B$ is exactly equivalent to the type $\Pi a : A.B(a)$ if the variable a is not used in the expression $B(a)$, the similarity to Figure 2.7 should be easy to see.

$$\frac{}{\vdash s_1 : s_2}\ (s_1, s_2) \in A \qquad (\textsc{Start})$$

$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ s \in S \qquad (\textsc{Var})$$

$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash B : s}{\Gamma, y : B \vdash x : A}\ s \in S \qquad (\textsc{Weaken})$$

$$\frac{\Gamma \vdash f : \Pi_{x:A} B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash f a : B[x := a]} \qquad (\textsc{App})$$

$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash \Pi_{x:A} B(x) : s}{\Gamma \vdash (\lambda x : A.b) : \Pi_{x:A} B(x)}\ s \in S \qquad (\textsc{Abs})$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi_{x:A} B(x) : s_3}\ (s_1, s_2, s_3) \in R \qquad (\textsc{Product})$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A' : s \qquad A \longrightarrow_\beta A'}{\Gamma \vdash a : A'}\ s \in S \qquad (\textsc{Conv})$$

Figure 2.9: Typing rules of a pure type system

[2]The elements of R are written as (s_1, s_2), which is equivalent to (s_1, s_2, s_2).

Universes The notion of sorts and axioms can be generalized even more. Instantiating this system with an infinite set of sorts $S = \{Type_0, Type_1, ...\}$ instead of the set $\{\star, \Box\}$ and setting A to $\{(Type_0, Type_1), (Type_1, Type_2), ...\}$ leads to an infinite hierarchy of *type universes*. Proof assistants commonly use such a hierarchy [8].

Type in Type Going the other way around, simplifying S to $S = \{\star\}$ and setting A to $\{(\star, \star)\}$, lead to an inconsistent logic system called $\lambda\star$, also called a system with a *Type in Type* rule. This leads to paradoxes similar to the Russel's paradox in set theory.

In many pedagogic implementations of dependently-typed λ-calculi I saw, though, this was simply acknowledged: separating universes introduces complexity but the distinction is not as important for many purposes.

For the goal of this thesis–testing the characteristics of a runtime system–the distinction is unimportant. In the rest of the text we will use the inconsistent $\lambda\star$-calculus, but with all the constructs mentioned in the preceding type systems. We will now formally define these constructs, together with several extensions to this system that will be useful in the context of just-in-time compilation using Truffle, e.g., (co)product types, booleans, natural numbers.

Proof assistants and other dependently-typed programming languages use systems based on $\lambda\Pi\omega$-calculus, which is called the Calculus of Constructions. They add more extensions: induction and subtyping are common ones. We will discuss only a subset of them in the following section, as many of these are irrelevant to the goals of this thesis.

2.3 Types

With the basic concepts introduced, we can move on to specifying the syntax and semantics of the language that will be used for the implementation and evaluation part of this thesis. While it is possible to derive any types using only three constructs: Π-types (dependent product), Σ-types (dependent sum), and W-types (inductive types), that we have not seen so far; we will define specific *"wired-in"* types in addition to the Π- and Σ-types, as they are more straightforward to both use and implement.

We will specify the syntax and semantics of each type at the same time. For syntax, we will define the terms and values, for semantics we will use four parts: type formation, a way to construct new types; term introduction (constructors), ways to construct terms of these types; term elimination (destructors), ways to use them to construct other terms; and computation rules that describe what happens when an introduced term is eliminated. The algorithms to normalize and type-check these terms will be mentioned in the following chapter. In this section we will solely focus on the syntax and semantics.

2.3.1 Π-types

As mentioned above, the type $\Pi a : A.B$, also called the *dependent product type* or the *dependent function type*, is a generalization of the function type $A \to B$. Where the function type simply asserts that its corresponding function will receive a value of a certain type as its argument, the Π-type makes the value available in the rest of the type. Figure 2.10 introduces its semantics; they are similar to the typing rules of λ→-calculus function application, except for the substitution in the type of B in rule **Elim-Pi**.

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Pi x : A.B}\ (\textbf{Type-Pi})$$

$$\frac{\Gamma, a : A \vdash b : B}{\Gamma \vdash \lambda x.b\ :\ \Pi x : A.B}\ (\textbf{Intro-Pi}) \qquad \frac{\Gamma \vdash f\ :\ \Pi x : A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B[x := a]}\ (\textbf{Elim-Pi})$$

$$\frac{\Gamma, a : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x : A.b)a \longrightarrow_{\beta} b[x := a]}\ (\textbf{Eval-Pi})$$

Figure 2.10: Π-type semantics

While a very common example of a Π-type is the length-indexed vector $\Pi(n : \mathbb{N}).Vec(\mathbb{R}, n)$, it is also possible to define a function with a *"dynamic"* number of arguments like in the following listing. It is a powerful language feature also for its programming uses, as it makes it possible to, e.g., implement a well-typed function `printf` that, e.g., produces the function *Nat* → *Nat* → *String* when called as `printf "%d%d"`. The following is an example of a function, whose number of arguments changes based on the value of the first argument.

$$\begin{array}{rcl}
succOrZero & : & \Pi(b : Bool).\ if\ b\ then\ (Nat \to Nat)\ else\ Nat \\
succOrZero & = & \Pi(b : Bool).\ if\ b\ then\ (\lambda x.\ x + 1)\ else\ 0 \\
succOrZero\ true\ 0 & \longrightarrow_{\beta\delta} & 1 \\
succOrZero\ false & \longrightarrow_{\beta\delta} & 0
\end{array}$$

Implicit arguments The type-checker can infer many type arguments. Agda adds the concept of implicit function arguments [8] to ease the programmer's work and mark inferrable type arguments in a function's type signature. Such arguments can be specified when calling a function using a special syntax, but they are not required [33]. We will do the same, and as such we will split the syntax of a Π-type back into three separate constructs, which can be seen in Figure 2.11.

$$\begin{array}{rclclcll}
term & := & a \to b & \mid & (a : A) \to b & \mid & \{a : A\} \to b & \text{(abstraction)} \\
 & \mid & f\ a & \mid & & \mid & f\ \{a\} & \text{(application)} \\
value & := & \Pi a : A.b & & & & &
\end{array}$$

Figure 2.11: Π-type syntax

The plain *function type A* → *B* is simple to type but does not bind the value provided as the argument A. The *explicit Π-type* $(a : A) \to B$ binds the value a and makes it available to use inside B, and the *implicit Π-type* $\{a : A\} \to B$ marks the argument as one that type elaboration should be able to infer from the surrounding context. The following is an example of the implicit argument syntax, a polymorphic function *id*.

$$\begin{array}{rclcl}
id & : & \{A : \star\} \to A \to A & := & \Pi(x : A).x \\
id\ \{Nat\} & : & Nat \to Nat & \longrightarrow_{\beta\delta} & \lambda(x : Nat).x \\
id\ 1 & : & Nat & \longrightarrow_{\beta\delta} & 1
\end{array}$$

$$\frac{\Gamma \vdash A : \star \qquad \Gamma, x : A \vdash B : \star}{\Gamma \vdash \Sigma_{x:A} B : \star} \textbf{ (Type-Sigma)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash B : \star \qquad \Gamma \vdash b : B[x := a]}{\Gamma \vdash (a, b) : \Sigma_{x:A} B} \textbf{ (Intro-Sigma)}$$

$$\frac{\Gamma \vdash p : \Sigma_{x:A} B}{\Gamma \vdash \pi_1\ p : A} \textbf{ (Elim-Sigma1)} \qquad \frac{\Gamma \vdash p : \Sigma_{x:A} B}{\Gamma \vdash \pi_2\ p : B[x := fst\ p]} \textbf{ (Elim-Sigma2)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash B : \star \qquad \Gamma \vdash b : B[x := a]}{\Gamma \vdash \pi_1\ (a, b) \longrightarrow_\iota a : A} \textbf{ (Eval-Sigma1)}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash B : \star \qquad \Gamma \vdash b : B[x := a]}{\Gamma \vdash \pi_2\ (a, b) \longrightarrow_\iota b : B} \textbf{ (Eval-Sigma2)}$$

Figure 2.12: Σ-type semantics

2.3.2 Σ-types

The Σ-type is also called the *dependent pair type*, or alternatively the dependent tuple, dependent sum, or even the dependent product type. Like the Π-type was a generalization of the function type, the Σ-type is a generalization of a product type, or simply a *pair*. Semantically, the Σ-type is similar to the tagged union in C-like languages: the type $\Sigma(a : A).B(a)$ corresponds to a value (a, b), only the type $B(a)$ can depend on the first member of the pair. This is illustrated in Figure 2.12, where the dependency can be seen in rule **Intro-Sigma**, in the substitution $B[x := a]$.

Above, we had a function that could accept different arguments based on the value of the first argument. Below, we have a type that simply uses Σ in place of Π in the type: based on the value of the first member, the second member can be either a function or a value, and still be a well-typed term.

$$\begin{array}{rcl} FuncOrVal & : & \Sigma(b : Bool).\ if\ b\ then\ (Nat \to Nat)\ else\ Nat \\ (true, \lambda x.\ x + 1) & : & FuncOrVal \\ (false, 0) & : & FuncOrVal \end{array}$$

Pair Similar to the function type, given the expression $\Sigma(a : A).B(a)$, if a does not occur in the expression $B(a)$, then it is the non-dependent pair type. The pair type is useful to express an isomorphism also used in general programming practice: a conversion between a function of two arguments, and a function of one argument that returns a function of one argument:

$$\begin{array}{rclcll} & & A \times B \to C & \Leftrightarrow & A \to B \to C & \\ curry & := & \lambda(f : A \times B \to C). & & \lambda(x : A).\lambda(y : B). & f\ (x, y) \\ uncurry & := & \lambda(f : A \to B \to C). & & \lambda(x : A \times B). & f\ (\pi_1\ x)\ (\pi_2\ y) \end{array}$$

Tuple The n-tuple is a generalization of the pair, a non-dependent set of an arbitrary number of values, otherwise expressible as a set of nested pairs: commonly written as $(a_1, ..., a_n)$.

Record A record type is similar to a tuple, only its members have unique labels. In Figure 2.13, we see the semantics of a general record type, using the notation $\{l_i = t_i\} \ : \ \{l_i : T_i\}$ and a projection *record.member*.

$$\frac{\forall i \in \{1..n\}\ \Gamma \vdash T_i : \star}{\Gamma \vdash \{l_i : T_i^{i \in \{1..n\}}\} : \star}\ \textbf{(Type-Rec)}$$

$$\frac{\forall i \in \{1..n\} \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i \in \{1..n\}}\} : \{l_i : T_i^{i \in \{1..n\}}\}}\ \textbf{(Intro-Rec)}$$

$$\frac{\Gamma \vdash t : \{l_i : T_i^{i \in \{1..n\}}\}}{\Gamma \vdash t.l_i : T_i}\ \textbf{(Elim-Rec)}$$

$$\frac{\forall i \in \{1..n\}\ \Gamma \vdash t_i : T_i \qquad \Gamma \vdash t : \{l_i : T_i^{i \in \{1..n\}}\}}{\Gamma \vdash \{l_i = t_i^{i \in \{1..n\}}\}.l_i \longrightarrow_\iota t_i : B}\ \textbf{(Eval-Rec)}$$

Figure 2.13: Record semantics

In Figure 2.14, we have a syntax that unifies all of these concepts: a Σ-type, a pair, an n-tuple, a named record. A non-dependent n-tuple type is written as $A \times B \times C$ with values (a, b, c). Projections of non-dependent tuples use numbers, e.g., $p.1$, $p.2$, ... A dependent sum type is written in the same way as a named record: $(a : A) \times B$ binds the value $a : A$ in the rest of the type B, and on the value-level enables the projection $obj.a$.

$$\begin{array}{rcllr}
term & := & T_1 \times \cdots \times T_n & \mid \ (l_1 : T_1) \times \cdots \times (l_n : T_n) \times T_{n+1} & \text{(types)} \\
 & \mid & t.i & \mid \ t.l_n & \text{(destructors)} \\
 & \mid & (t_1, \cdots, t_n) & & \text{(constructor)} \\
value & := & (t_1, \cdots, t_n) & &
\end{array}$$

Figure 2.14: Σ-type syntax

Coproduct The sum type or the coproduct $A + B$ can have values from both types A and B, often written as $a : A \vdash inl\ A : A + B$, where *inl* means "on the left-hand side of the sum $A + B$". This can be generalized to the concept of *variant types*, with an arbitrary number of named members; shown below, using Haskell syntax:

$$data\ Maybe\ a = Nothing \mid Just\ a$$

For the purposes of our language, a binary sum type is useful, but inductive variant types would require more involved constraint checking, so we will ignore those, only using simple sum types in the form of $A + B$. This type can be derived using a dependent pair where the first member is a boolean.

$$Char + Int \quad \simeq \quad \Sigma(x : Bool).\ if\ x\ Char\ Int$$

2.3.3 Value types

Finite sets Pure type systems mentioned in the previous chapter often use types like **0**, **1**, and **2** with a finite number of inhabitants, where the type **0** (with zero inhabitants of the type) is the empty or void type. Type **1** with a single inhabitant is the unit type, and the type **2** is the boolean type. Also, the infinite set of natural numbers can be defined using induction over **2**. For our purposes, it is enough to define a fixed number of types, though.

Unit The unit type **1**, or commonly written as the 0-tuple "`()`", is sometimes used as a universal return value. As it has no evaluation rules, though, we can simply add a new type *Unit* and a new value and term *unit*, with the rule $unit : Unit$.

Booleans The above-mentioned type **2** has two inhabitants and can be semantically mapped to the boolean type. In Figure 2.15, we introduce the values *true* and *false*, and a simple eliminator *cond* that returns one of two values based on the truth value of its argument.

$$\frac{}{\vdash Bool : \star}\ \textbf{(Type-Nat)}$$

$$\frac{}{\vdash true : Bool}\ \textbf{(Intro-True)} \qquad \frac{}{\vdash false : False}\ \textbf{(Intro-False)}$$

$$\frac{\Gamma \vdash a_1 : A \qquad \Gamma \vdash a_2 : A}{\Gamma, x : Bool \vdash if\ x\ a_1\ a_2 : A}\ \textbf{(Elim-Bool)}$$

$$\frac{\Gamma \vdash a_1 : A \qquad \Gamma \vdash a_2 : A}{\Gamma \vdash cond\ true\ a_1\ a_2 \longrightarrow_\iota a_1 : A}\ \textbf{(Eval-True)} \quad \frac{\Gamma \vdash a_1 : A \qquad \Gamma \vdash a_2 : A}{cond\ false\ a_1\ a_2 \longrightarrow_\iota a_2 : A}\ \textbf{(Eval-False)}$$

Figure 2.15: `Bool` semantics

Natural numbers The natural numbers form an infinite set, unlike the above value types. On their own, adding natural numbers to a type system does not produce non-termination, as the recursion involved in their manipulation can be limited to primitive recursion as, e.g., used in Gödel's System T [8]. The constructions introduced in Figure 2.16 are simply the constructors *zero* and *succ*, and the destructor *natElim* unwraps at most one layer of *succ*.

$$\frac{}{\vdash Nat : \star}\ \textbf{(Type-Nat)}$$

$$\frac{}{\vdash zero : Nat}\ \textbf{(Intro-Zero)} \qquad \frac{\Gamma \vdash n : Nat}{\Gamma \vdash succ\ n : Nat}\ \textbf{(Intro-Succ)}$$

$$\frac{\Gamma \vdash a_1 : A \qquad \Gamma, n : Nat \vdash a_2 : A}{\Gamma, x : Nat \vdash natElim\ x\ a_1\ (\lambda x.a_2)}\ \textbf{(Elim-Nat)}$$

$$\frac{\Gamma \vdash a_1 : A \qquad \Gamma, n : Nat \vdash a_2 : A}{\Gamma \vdash natElim\ zero\ a_1\ (\lambda x.a_2) \longrightarrow_\iota a_1 : A}\ \textbf{(Eval-Zero)}$$

$$\frac{\Gamma \vdash a_1 : A \qquad \Gamma, n : Nat \vdash a_2 : A \qquad \Gamma \vdash n : Nat}{natElim\ (succ\ n)\ a_1\ (\lambda x.a_2) \longrightarrow_\iota a_2[x := n] : A}\ \textbf{(Eval-Succ)}$$

Figure 2.16: `Nat` semantics

2.4 Remaining constructs

These constructs together form a complete core language capable of forming and evaluating expressions. Already, this would be a usable programming language. However, the *surface language* is still missing: the syntax for defining constants and variables, and interacting with the compiler.

Local definitions The λ-calculus is, to use programming language terminology, a purely functional programming language: without specific extensions, any language construct is an expression. We will use the syntax of Agda, and keep local variable definition as an expression as well, using a `let-in` construct, with the semantics given in Figure 2.17.

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \text{let}\, x = a \,\text{in}\, b : B}\ (\textbf{Type-Let})$$

$$\frac{\Gamma \vdash v : A \qquad \Gamma, x : A \vdash e : B}{\text{let}\, x = v \,\text{in}\, e \longrightarrow_{\zeta} e[x := v]}\ (\textbf{Eval-Let})$$

Figure 2.17: `let-in` semantics

Global definitions Global definitions are not strictly necessary, as with local definitions and the fixed-point combinator we could emulate them. However, global definitions will be useful later in the process of elaborations, when global top-level definitions will separate blocks that we can type-check separately. We will add three top-level expressions: a declaration that only assigns a name to a type, and a definition with and without type. Definitions without types will have them inferred.

$$\begin{array}{rcl} top & := & id : term \\ & \mid & id : term = term \\ & \mid & id = term \end{array}$$

Holes A construct that serves solely as information to the compiler and will not be used at runtime is a *hole*. It can take the place of a term in an expression and marks the missing term as one to be inferred ("filled in") during elaboration[3]. In fact, the syntax for a global definition without a type will use a hole in place of its type. The semantics of a hole are omitted on purpose as they would also require specifying the type inference algorithm.

$$term \quad := \quad \_$$

Interpreter directives Another type of top-level expressions is a pragma, a direct command to the compiler. We will use these when evaluating the time it takes to normalize or elaborate an expression, or when enabling or disabling the use of "wired-in" types, e.g., to

[3]Proof assistants also use the concept of a metavariable, often with the syntax $?\alpha$.

compare the performance impact of using a Church encoding of numbers versus a natural type that uses hardware integers. We will once again use the syntax of Agda:

$$
\begin{array}{rcl}
\mathit{top} & := & \{-\#\ \mathit{BUILTIN}\ \mathit{id}\ \#-\} \\
 & | & \{-\#\ \mathit{ELABORATE}\ \mathit{term}\ \#-\} \\
 & | & \{-\#\ \mathit{NORMALIZE}\ \mathit{term}\ \#-\}
\end{array}
$$

The syntax and semantics presented here altogether comprise a working programming language. A complete listing of the syntax and semantics is included in Appendix A.

This syntax now needs to be translated into a recognizer (a parser and a lexer), and the semantics into a type-checker and an evaluator for the language. A simplified grammar, translated from the syntax, is included in Listing 2.2. Compared to the previous syntax specifications, the grammar also needs to encode the precedence and associativity of each construct.

With this, the language specification is complete, and we can move on to the next part, implementing a type-checker and an interpreter for this language.

```
FILE : STMT (STMTEND STMT)* ;
STMT : '{-#' ID+ '#-}'
     | ID ':' EXPR
     | ID (':' EXPR)? '=' EXPR
     ;
EXPR : 'let' ID ':' EXPR '=' EXPR 'in' EXPR
     | 'λ' LAM_BINDER '.' EXPR
     | PI_BINDER+ '→' EXPR
     | ATOM ARG*
     ;
LAM_BINDER
     : ID | '_' | '{' (ID | '_') '}' ;
PI_BINDER
     : ATOM ARG*
     | '(' ID+ ':' EXPR ')'
     | '{' ID+ ':' EXPR '}'
     ;
ARG
     : ATOM
     | '{' ID ('=' TERM)? '}'
     ;
ATOM : '(' ID ':' EXPR ')' '×' EXPR
     | EXPR '×' EXPR
     | '(' EXPR (',' EXPR)+ ')'
     | '(' EXPR ')'
     | ID '.' ID
     | ID | NAT | 'Unit' | | '_'
     ;
STMTEND : ('\n' | ';')+ ;
ID : [a-zA-Z] [a-zA-Z0-9] ;
SKIP : [ \t] | '--' [^\r\n]* | '{-' [^#] .* '-}' ;
```

Listing 2.2: A simplified version of the grammar (written using ANTLR syntax)

Chapter 3

Language implementation: Montuno

3.1 Introduction

Now with a complete language specification, we can move onto the next step: writing an interpreter. In this chapter, we will introduce the algorithms at the core of an interpreter and build a tree-based interpreter for the language, elaborating on key implementation decisions. The algorithms involved can be translated from specification to code quite naturally, at least in the style of interpreter we will create in this chapter. The second interpreter in Truffle will require a quite different programming paradigm and deciding on many low-level implementation details, e.g., how to implement actual function calls.

These algorithms presented here are state-of-the-art algorithms that are also used in other dependently-typed languages. In particular, normalization-by-evaluation as presented by Christiansen [10]; bidirectional typing as formalized by Dunfield and Krishnaswami [12]; and pattern unification presented in the thesis of Ulf Norell [39]. Several key implementation decisions: laziness, choice of meta-context, and the specifics of unification, were based on Kovács' SmallTT project [32].

The Kotlin implementation is fully my work. Most implementations of dependently-typed languages are in (purely) functional languages, with Haskell being the most common, so while it would simplify this part of the thesis, it would be impossible to extend an existing implementation. The reasoning behind picking Kotlin as the implementation language will be explained momentarily. The interpreter created in this chapter will be referred to using the working name Montuno[1].

Language The choice of a programming language is mostly decided by the eventual target platform Truffle, as we will be able to share parts of the implementation between the two interpreters. The language of GraalVM and Truffle is Java, although other languages that run on the Java Virtual Machine can be used[2]. My personal preference lies with more func-

[1]Montuno, as opposed to the project Cadenza, to which this project is a follow-up. Both are music terms, *cadenza* being a "long virtuosic solo section", whereas *montuno* is a "faster, semi-improvised instrumental part".

[2]Even though Kotlin seems not to be recommended by Truffle authors, there are several languages implemented in it, which suggests there are no severe problems. (from `https://github.com/oracle/graal/issues/1228`)

tional languages like Scala or Kotlin, as the code is often cleaner and more concise[3], so in the end, after comparing the languages, I have selected Kotlin due to its multi-paradigm nature: Truffle requires the use of annotated classes, but this first interpreter can be written in a more natural functional style.

Libraries Truffle authors recommend against using many external libraries in the internals of the interpreter, as the techniques the libraries use may not work well with Truffle: the JIT compiler relies on inlining and whole-function optimization, and any external call to, e.g., a logging service, might be inlined and cause compilation slow-downs.

Therefore, we will need to design our own supporting data structures based on the fundamental data structures provided directly by Kotlin. Only two external libraries would be too complicated to reimplement, and both of these were chosen because they are among the most widely used in their field:

- a parser generator, ANTLR, to process input into an abstract syntax tree,
- a terminal interface library, JLine, to implement the interactive interface.

For the build and test system, the recommended choices of Gradle and JUnit were used.

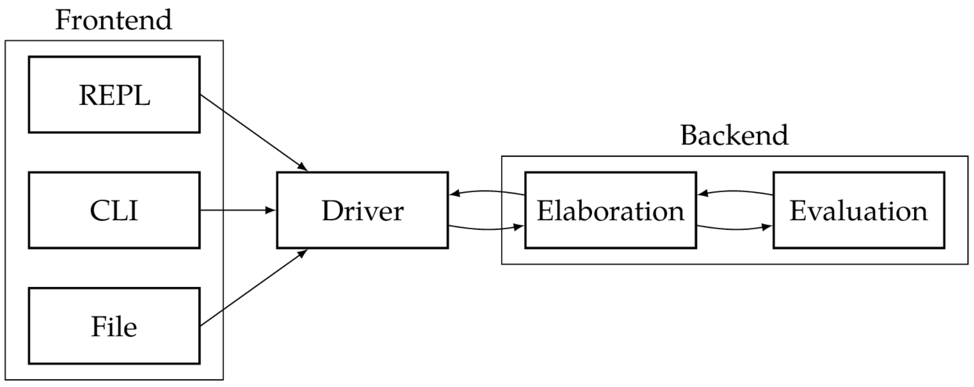


Figure 3.1: Interpreter component overview

3.1.1 Program flow

A typical interpreter takes in the user's input, processes it, and outputs a result. In this way, we can divide the interpreter into a frontend, a driver, and a backend, to reuse compiler terminology. A frontend handles user interaction, be it from a file or from an interactive environment, a backend implements the language semantics, and a driver connects them, illustrated in Figure 3.1.

Frontend The frontend is intended to be a simple way to execute the interpreter, offering two modes: a batch processing mode that reads from a file, and an interactive terminal

[3]Kotlin authors claim 40% reduction in the number of lines of code, compared to imperative code in Java (from `https://kotlinlang.org/docs/faq.html`)

environment that receives user input and prints out the result of the command. Proof assistants like Agda offer deeper integration with editors like tactics-based programming or others, similar to the refactoring tools offered in development environments for object-oriented languages, but that is unnecessary for the purposes of this thesis.

Backend The components of the backend, here represented as *elaboration* and *evaluation*, implement the data transformation algorithms that are further illustrated in Figure 3.2. In brief, the *elaboration* process turns user input in the form of partially-typed, well-formed *pre-terms* into fully-annotated well-typed *terms*. *Evaluation* converts between a *term* and a *value*: a term can be compared to program data, it can only be evaluated, whereas a value is the result of such evaluation and can be, e.g., compared for equality.

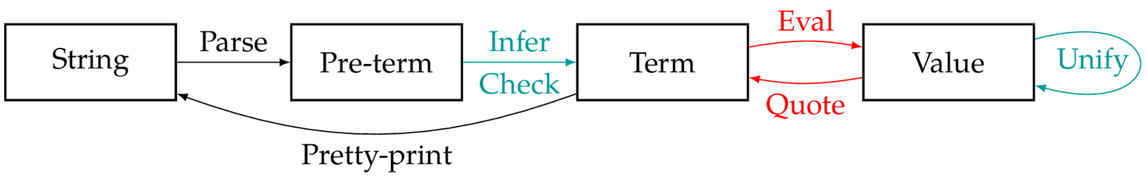


Figure 3.2: Data flow overview. Cyan is elaboration, red is normalization-by-evaluation

Data flow This interpreter can be called an AST (abstract syntax tree) interpreter, as the principal parts all consist of tree traversals and transformations, as all of the main data structures involved are trees: pre-terms, terms, and values are recursive data structures. The main algorithms to be discussed are: evaluation, normalization, and elaboration, all of them can be translated to tree traversals in a straightforward way.

In Figure 3.2, *Infer* and *Check* correspond to type checking and type inference, two parts of the *bidirectional typing* algorithm that we will use. *Unification* (*Unify*) forms a major part of the elaboration process, as that is how we check whether two values are equal. *Eval* corresponds to the previously described βδζι-reduction implemented using the *normalization-by-evaluation* style, whereas *Quote* builds a term back up from an evaluated value. To complete the description, *Parse* and *Pretty-print* convert between the user-readable, string representation of terms and the data structures of their internal representation. For the sake of clarity, the processes are illustrated using their simplified function signatures in Listing 3.1.

```
fun parse(input: String): PreTerm;
fun pretty(term: Term): String;
fun infer(pre: PreTerm): Pair<Term, Val>;
fun check(pre: PreTerm, wanted: Val): Term;
fun eval(term: Term): Val;
fun quote(value: Val): Term;
fun unify(left: Val, right: Val): Unit;
```

Listing 3.1: Simplified signatures of the principal functions

We will first define the data types in this chapter, especially focusing on closure representation. Then, we will specify and implement two algorithms: *normalization-by-evaluation*, and *bidirectional type elaboration*, and lastly, we finish the interpreter by creating its driver and frontend.

3.2 Data structures

In the previous chapter, we have specified the syntax of the language, which we first need to translate to concrete data structures before trying to implement the semantics. Sometimes, the semantics impose additional constraints on the design of the data structures, but in this case, the translation is quite straight-forward.

Properties Terms and values form recursive data structures. We will also need a separate data structure for pre-terms as the result of parsing user input. All of these structures represent only well-formed terms and in addition, terms and values represent the well-typed subset of well-formed terms. Well-formedness should be ensured by the parsing process, whereas type-checking will take care of the second property.

Pre-terms As pre-terms are mostly just an encoding of the parse tree without much further processing, the complete data type is only included in Appendix A.4. The `PreTerm` class hierarchy mostly reflects the `Term` classes with a few key differences, like the addition of compiler directives or variable representation, so in the rest of this section, we will discuss terms and values only.

Location A key feature that we will also disregard in this chapter is term location that maps the position of a term in the original source expression, mostly for the purpose of error reporting. As location is tracked in a field that occurs in all pre-terms, terms, and values, it will only be included in the final listing of classes in Appendix A.4.

$$
\begin{array}{rclclcl}
\mathit{term} & := & v & \mid & \mathit{constant} & & \\
 & \mid & a\ b & \mid & a\ \{b\} & & \\
 & \mid & a \to b & \mid & (a : A) \to b & \mid & \{a : A\} \to b \\
 & \mid & a \times b & \mid & (l : A) \times b & \mid & a.l \\
 & \mid & \text{let } x = v \text{ in } e & & & & \\
 & \mid & \_ & & & & \\
\mathit{value} & := & \mathit{constant} & & & & \\
 & \mid & \lambda x : A.b & \mid & \Pi x : A.b & & \\
 & \mid & (a_1, \cdots, a_n) & & & & \\
 & \mid & \_ & & & &
\end{array}
$$

Figure 3.3: Terms and values in Montuno (revisited)

The terms and values that were specified in Chapter 2 are revisited in Figure 3.3, there are two main classes of terms: those that represent computation (functions and function application), and those that represent data (pairs, records, constants).

Data classes Most *data* terms can be represented in a straight-forward way, as they map directly to features of the host language, Kotlin in our case. Kotlin recommends a standard way of representing primarily data-oriented structures using `data classes`[4]. These are

[4]`https://kotlinlang.org/docs/idioms.html`

classes whose primary purpose is to hold data, so-called Data Transfer Objects (DTOs). In Listing 3.2 we have the base classes for terms and values, and a few examples of structures mapped from the syntax to code.

```
sealed class Term
data class TLocal(val ix: Ix) : Term()
data class TPair(val left: Term, val right: Term) : Term()
data class TPi(val id: String?, val bound: Term, val body: Term) : Term()
data class TSg(val id: String?, val bound: Term, val body: Term) : Term()

sealed class Val
data class VLocal(val lvl: Lvl) : Val()
data class VPair(val left: Val, val right: Val) : Val()
data class VPi(val id: String?, val bound: Val, val cl: Closure) : Val()
data class VSg(val id: String?, val bound: Val, val cl: Closure) : Val()
```

Listing 3.2: Data classes representing some of the terms and values in the language

Some constructs are straightforward to map, but terms that encode computation, whether delayed (λ-abstraction) or not (application) are more involved. Variables *can* be represented in a straightforward way using the variable name, but a string-based representations is not the most optimal way. We will look at these three constructs in turn.

3.2.1 Functions

Closure Languages, in which functions are first-class values, all use the concept of a closure. A closure is, in brief, a function in combination with the environment in which it was created. The body of the function can refer to variables other than its immediate arguments, which means that the surrounding environment needs to be stored as well. The simplest example is the *const* function $\lambda x.\lambda y.x$, which, when partially applied to a single argument, e.g., let $plusFive = plus\ 5$, needs to store the value 5 until it is eventually applied to the remaining second argument: $plusFive\ 15 \longrightarrow 20$.

HOAS As Kotlin supports closures on its own, it would be possible to encode λ-terms directly as functions in the host language. This is possible, and it is one of the ways of encoding functions in interpreters. This encoding is called the higher-order abstract syntax (HOAS), which means that functions[5] in the language are equal to functions in the host language. Representing functions using HOAS produces very readable code, and in some cases, e.g., in the Haskell compiler GHC, it produces code an order of magnitude faster than using other representations [31]. An example of what it looks like is in Listing 3.3.

```
data class HOASClosure<T>(val body: (T) -> T)

val constFive = HOASClosure<Int> { (n) -> 5 }
```

Listing 3.3: Higher-order abstract syntax encoding of a closure

[5]In descriptions of the higher-order abstract syntax, the term *binders* is commonly used instead of function or λ-abstractions, as these constructs *bind* a value to a name.

Explicit closures However, we will need to perform some operations on the AST that need explicit access to environments and the arguments of a function. The alternative to reusing functions of the host language is a *defunctionalized* representation, also called *explicit closure* representation. We will need to use this representation later, when creating the Truffle version: function calls will need to be objects, nodes in the program graph, as we will see in Chapter 4. In this encoding, demonstrated in Listing 3.4, we store the term of the function body together with the state of the environment when the closure was created.

```
data class ExplicitClosure<T>(val env: Map<Name, Val>, val body: Term)

val constFive = ExplicitClosure<Int>(mapOf("x" to VNat(5)), TLocal("x"))
```

Listing 3.4: Defunctionalized function representation

3.2.2 Variables

Variable representation can be simple, as in Listing 3.4: a variable can be a simple string containing the name of the variable. This is also what our parser produces in the pre-term representation. Also, when describing reduction rules and substitution, we have also referred to variables by their names. That is not the best way of representing variables.

Named Often, when specifying a λ-calculus, the process of substitution $t[x := e]$ is kept vague, as a concern of the meta-theory in which the λ-calculus is encoded. When using variable names (strings), the terms themselves and the code that manipulates them are easily understandable. Function application, however, requires variable renaming (α-conversion), which involves traversing the entire argument term and replacing each variable occurrence with a fresh name that does not yet occur in the function body. However, this is a very slow process, and it is not used in any real implementation of dependent types or λ-calculus.

Nameless An alternative to string-based variable representation is a *nameless* representation, which uses numbers in place of variable names [28]. These numbers are indices that point to the current variable environment, offsets from either end of the environment stack. The numbers are assigned, informally, by *counting the lambdas*, as each λ-abstraction corresponds to one entry in the environment. The environment can be represented as a stack to which a variable is pushed with every function application, and popped when leaving a function. The numbers then point to these entries. These two approaches can be seen side-by-side in Figure 3.4.

	fix	$succ$
Named	$(\lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x)))\ g$	$\lambda x.x\ (\lambda y.x\ y)$
Indices	$(\lambda(\lambda 1\ (0\ 0)\ (\lambda 1\ (0\ 0))\ g$	$\lambda 0\ (\lambda 1\ 0)$
Levels	$(\lambda(\lambda 0\ (1\ 1)\ (\lambda 0\ (1\ 1))\ g$	$\lambda 0\ (\lambda 0\ 1)$

Figure 3.4: Named and nameless variable representations

de Bruijn indices The first way of addressing, de Bruijn indexing, is rather well-known. It is a way of counting from the top of the stack, meaning that the argument of the innermost (most recent) lambda has the lowest number. It is a "relative" way of counting, relative to the top of the stack, which is beneficial during, e.g., δ-reduction, in which a reference to a function is replaced by its definition: using indices, the variable references in the function body do not need to be adjusted after such substitution.

de Bruijn levels The second way is also called the "reversed de Bruijn indexing" [35], as it counts from the bottom of the stack. This means that the argument of the innermost lambda has the highest number. In the entire term, one variable is only ever addressed by one number, meaning that this is an "absolute" way of addressing, as opposed to the "relative" indices.

Locally nameless There is a third alternative that combines both named and nameless representations, and it has been used in e.g., the Lean proof assistant [13]. De Bruijn indices are used for bound variables and string-based names for free variables. This also avoids any need for bound variable substitution, but free variables still need to be resolved later during the evaluation of a term.

Our choice We will use a representation that has been used in recent type theory implementations [14, 21]: de Bruijn indices in terms, and de Bruijn levels in values. Such a representation avoids any need for substitution: "relative" indices do not need to be adjusted based on the size of the environment, whereas the "absolute" addressing of levels in values means that values can be directly compared. This combination of representations means that we can avoid doing any substitution at all, as any adjustment of variables is performed during the evaluation from term to value and back.

Implementation Kotlin makes it possible to construct type-safe wrappers over basic data types that are erased at runtime but that support custom operations. Representing indices and levels as `inline class`es means that we can increase and decrease them using the natural syntax e.g. `ix + 1`, which we will use when manipulating the environment in the next section. The final representation of variables in our interpreter is in Listing 3.5.

3.2.3 Class structure

Variables and λ-abstractions were the two non-trivial parts of the mapping between our syntax and Kotlin values. With these two pieces, we can fill out the remaining parts of the class hierarchy. The full class listing is in Appendix A.4, here only a direct comparison of the data structures is shown on the *const* function in Figure 3.5, and the most important differences between them are in Figure 3.6.

```
inline class Ix(val it: Int) {
    operator fun plus(i: Int) = Ix(it + i)
    operator fun minus(i: Int) = Ix(it - i)
    fun toLvl(depth: Lvl) = Lvl(depth.it - it - 1)
}

inline class Lvl(val it: Int) {
    operator fun plus(i: Int) = Lvl(it + i)
    operator fun minus(i: Int) = Lvl(it - i)
    fun toIx(depth: Lvl) = Ix(depth.it - it - 1)
}

data class VLocal(val it: Lvl) : Val()
data class TLocal(val it: Ix) : Term()
```

Listing 3.5: Variable representation

```
PLam("x", Expl,     TLam("x", Expl,     VLam("x", Expl,
  PLam("y", Expl,     TLam("y", Expl,     VCl([valX], VLam("y", Expl,
    PVar("x")))         TLocal(1)))         VCl([valX, valY], VLocal(0)))))
```

Figure 3.5: Direct comparison of `PreTerm`, `Term`, and `Value` objects

3.3 Normalization

Normalization is a series of βδζι-reductions, as defined in Chapter 2. While there are systems that implement normalization as an exact series of reduction rules, it is an inefficient approach that is not common in the internals of state-of-the-art proof assistants.

Normalization-by-evaluation An alternative way of bringing terms to normal form is the so-called *normalization-by-evaluation* (NbE) [42]. The main principle of this technique is interpretation from the syntactic domain of terms into a computational, semantic domain of values and back. In brief, we look at terms as an executable program that can be *evaluated*, the result of such evaluation is then a normal form of the original term. NbE is total and provably confluent [3] for any abstract machine or computational domain.

Neutral values If we consider only closed terms that reduce to a single constant, we could simply define an evaluation algorithm over the terms defined in the previous chapter. However, normalization-by-evaluation is an algorithm to bring any term into a full normal form, which means evaluating terms inside function bodies and constructors. NbE introduces the concept of "stuck" values that cannot be reduced further. In particular, free variables in a term cannot be reduced, and any terms applied to a stuck variable cannot be further

	Variables	Functions	Properties
`PreTerm`	String names	`PreTerm` AST	well-formed
`Term`	de Bruijn index	`Term` AST	well-typed
`Value`	de Bruijn level	Closure: `Term` + `Value` context	head-normal form

Figure 3.6: Important distinctions between `PreTerm`, `Term`, and `Value` objects

reduced and are "stuck" as well. These stuck values are called *neutral values*, as they are inert with regards to the evaluation algorithm.

Semantic domain Proof assistants use abstract machines like Zinc or STG; any way to evaluate a term into a final value is viable. This is also the reason to use Truffle, as we can translate a term into an executable program graph, which Truffle will later optimize as necessary. In this first interpreter, however, the computational domain will be a simple tree-traversal algorithm.

The set of neutral values in Montuno is rather small (Figure 3.7): an unknown variable, function application with a neutral *head* and arbitrary terms in the *spine*, and a projection eliminator.

$$neutral \quad := \quad var \quad | \quad neutral\, a_1 \ldots a_n \quad | \quad neutral.l_n$$

Figure 3.7: Neutral values

Specification The NbE algorithm is fully formally specifiable using four operations: the above-mentioned evaluation and quoting, reflection of a neutral value (*NeVal*) into a value, and reification of a value into a normal value (*NfVal*) that includes its type, schematically shown in Figure 3.8. In this thesis, though, we will only describe the relevant parts of the specification in words, and say that NbE (as we will implement it) is a pair of functions $nf = quote(eval(term))$.

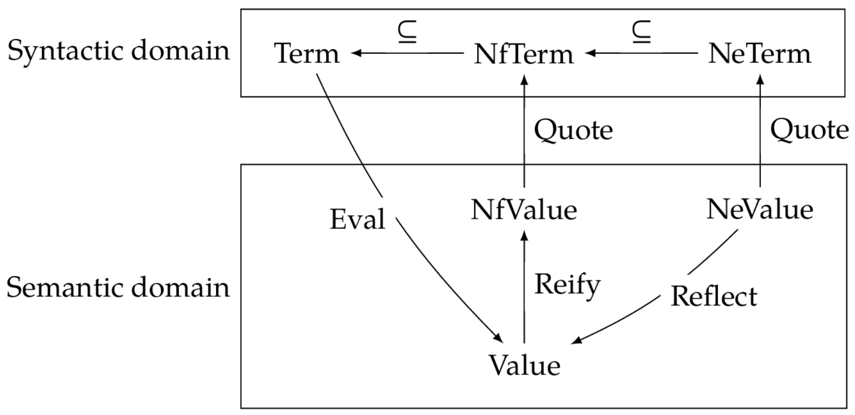


Figure 3.8: Syntactic and semantic domains in NbE [2]

3.3.1 Normalization strategies

Normalization-by-evaluation is, however, at its core inefficient for our purposes [29]. The primary reason to normalize terms in the interpreter is for type-checking and inference and that, in particular, needs normalized terms to check whether two terms are equivalent. NbE is an algorithm to get a full normal form of a term, whereas to compare values for equality, we only need the weak head-normal form. To illustrate: to compare whether a pair is equal to another term, we do not need to compare two fully-evaluated values, but only to find out whether that term is a pair of terms, which is given by the outermost constructor, the *head*.

In Chapter 2 we saw an overview of normal forms of λ-calculus. To briefly recapitulate, a normal form is a fully evaluated term with all sub-terms also fully evaluated. A weak head-normal form is a form where only the outermost construction is fully evaluated, be it a λ-abstraction or an application of a variable to a spine of arguments.

Reduction strategy Normal forms are associated with a reduction strategy, a set of small-step reduction rules that specify the order in which subexpressions are reduced. Each strategy brings an expression to their corresponding normal form. Common ones are *applicative order* in which we first reduce sub-terms left-to-right, and then apply functions to them; and *normal order* in which we first apply the leftmost function, and only then reduce its arguments. In Figure 3.9 there are two reduction strategies that we will emulate.

$$x \xrightarrow{name} x$$

$$\frac{}{(\lambda x.e) \xrightarrow{name} (\lambda x.e)}$$

$$\frac{e_1 \xrightarrow{name} (\lambda x.e) \qquad e[x := e_2] \xrightarrow{name} e'}{(e_1\ e_2) \xrightarrow{name} e'}$$

$$\frac{e_1 \xrightarrow{name} e_1' \not\equiv \lambda x.e}{(e_1\ e_2) \xrightarrow{name} (e_1'\ e_2)}$$

(a) Call-by-name to weak head normal form

$$x \xrightarrow{norm} x$$

$$\frac{e \xrightarrow{norm} e'}{(\lambda x.e) \xrightarrow{norm} (\lambda x.e')}$$

$$\frac{e_1 \xrightarrow{name} (\lambda x.e) \qquad e[x := e_2] \xrightarrow{norm} e'}{(e_1\ e_2) \xrightarrow{norm} e'}$$

$$\frac{e_1 \xrightarrow{name} e_1' \not\equiv \lambda x.e \qquad e_1' \xrightarrow{norm} e_1'' \qquad e_2 \xrightarrow{norm} e_2'}{(e_1\ e_2) \xrightarrow{norm} (e_1''\ e_2)}$$

(b) Normal order to normal form

Figure 3.9: Reduction strategies for λ-calculus [45]

In general programming language theory, a concept closely related to reduction strategies is an evaluation strategy. These also specify when an expression is evaluated into a value, but in our case, they apply to our host language Kotlin.

Call-by-value Call-by-value, otherwise called eager evaluation, corresponds to applicative order reduction strategy [4]. Specifically, when executing a statement, its sub-terms are evaluated inside-out and immediately reduced to a value. This leads to predictable program performance (the program will execute in the order that the programmer wrote it, evaluating all expressions in order), but this may lead to unnecessary computations performed: given an expression `const 5 (ackermann 4 2)`, the value of `ackermann 4 2` will be computed but immediately discarded, in effect wasting processor time.

Call-by-need Call-by-need, also lazy evaluation, is the opposite paradigm. An expression will be evaluated only when its result is first accessed, not when it is created or defined. Using call-by-need, the previous example will terminate immediately as the calculation `ackermann 4 2` will be deferred and then discarded. However, it also has some drawbacks, as the performance characteristics of programs may be less predictable or harder to debug.

Call-by-value is the prevailing paradigm, used in the majority of commonly used languages. However, it is sometimes necessary to defer the evaluation of an expression, however, and in such cases lazy evaluation is emulated using closures or zero-argument functions: e.g., in Kotlin a variable can be initialized using the syntax `val x by lazy { ackermann(4, 2) }`, and the value will only be evaluated if it is ever needed.

Call-by-push-value There is also an alternative paradigm, called call-by-push-value, that subsumes both call-by-need and call-by-value as they can be directly translated to CBPV– in the context of λ-calculus specifically. It defines additional operators *delay* and *force* to accomplish this, one to create a *thunk* that contains a deferred computation, one to evaluate the thunk. Also notable is that it distinguishes between values and computations: values can be passed around, but computations can only be executed, or deferred.

Emulation We can emulate normalization strategies by implementing the full normalization by evaluation algorithm, and varying the evaluation strategy. Kotlin is by default a call-by-value language, though, and evaluation strategy is an intrinsic property of a language so, in our case, this means that we need to insert `lazy` annotations in the correct places, so that no values are evaluated other than those that are actually used. In the case of the later Truffle implementation, we will need to implement explicit *delay* and *force* operations of call-by-push-value, which is why we introduced all three paradigms in one place.

3.3.2 Implementation

The basic outline of the implementation is based on Christiansen [10]. In essence, it implements the obvious evaluation algorithm: evaluating a function captures the current environment in a closure, evaluating a variable looks up its value in the environment, and function application inserts the argument into the environment and evaluates the body of the function.

Environments The brief algorithm description used a concept we have not yet translated into Kotlin: the environment, or evaluation context. When presenting the λ→-calculus, we have seen the typing context Γ, to which we add a value context.

$$\Gamma \;\; := \;\; \bullet \;\; | \;\; \Gamma, x : t$$

The environment, following the above definition, is a stack: defining a variable pushes a pair of a name and a type to the top, which is then popped off when the variable goes out of scope. An entry is pushed and popped whenever we enter and leave a function context, and the entire environment needs to be captured in its current state whenever we create a closure. When implementing closures in Truffle, we will also need to take care about which variables are actually used in a function. That way, we can capture only those that need to be captured and not the entire environment.

Linked list The natural translation of the environment definition is a linked list. It would also be the most efficient implementation in a functional language like Haskell, as appending to an immutable list is very cheap there. In Kotlin, however, we need to take care about not allocating too many objects and will need to consider mutable implementations as well.

Mutable/immutable In Kotlin and other JVM-based languages, an `ArrayDeque` is a fast data structure, a mutable implementation of the stack data structure. In general, array-backed data structures are faster than recursive ones on the JVM, which we will use in the Truffle implementation. In this first interpreter, however, we can use the easier-to-use immutable linked list implementation. It is shown in Listing 3.6, a linked list specialized for values; an equivalent structure is also implemented for types.

```
data class VEnv(val value: Val, val next: VEnv?)

fun VEnv?.len(): Int = if (this == null) 0 else 1 + next.len()
operator fun VEnv?.plus(v: Val): VEnv = VEnv(v, this)
operator fun VEnv?.get(n: Ix): Val
   = if (n.it == 0) this!!.value else this!!.next[n - 1]
```

Listing 3.6: Environment data structure as an immutable linked list

Environment operations We need three operations from an environment data structure: insert (bind) a value, look up a bound value by its level or index, and unbind a variable that leaves the scope. In Listing 3.6, we see two of them: the operator `plus`, used as `env + value`, binds a value, and operator `get`, used as `env[ix]`, looks a value up. Unbinding a value is implicit, because this is an immutable linked list: the reference to the list used in the outer scope is not changed by any operations in the inner scope. These operations are demonstrated in Listing 3.7, on the `eval` operations of a variable and a `let-in` binding.

There we also see the basic structure of the evaluation algorithm. Careful placement of `lazy` has been omitted, as it splits the algorithm into two: parts that need to be evaluated lazily and those that do not, but the basic structure should be apparent. The snippet uses the Kotlin `when-is` construct, which checks the class of the argument, in this case we check if `this` is a `TLocal`, `TLet`, etc.

```
fun eval(ctx: Context, term: Term, env: VEnv): Val = when (term) {
  is TLocal ->
    env[term.ix] ?: VLocal(Lvl(ctx.lvl - term.ix - 1), spineNil)
  is TLet -> eval(ctx, term.body, env + eval(ctx, term.defn, env))
  is TLam -> VLam(term.name, VClosure(env, term.body))
  is TApp -> when (fn := eval(ctx, term.lhs, env)) {
    is VLam -> eval(ctx, fn.cl.term, fn.cl.env + eval(ctx, term.rhs, env))
    is VLocal -> VLocal(fn.head, fn.spine + term.right)
  }
  // ...
}
```

Listing 3.7: Demonstration of the `eval` algorithm

Eval In Listing 3.7, a variable is looked up in the environment, and considered a neutral value if the index is bigger than the size of the current environment. In `TLet` we see how an environment is extended with a local value. A λ-abstraction is converted into a closure. Function application, if the left-hand side is a `VLam`, evaluates the body of this closure, and if the left-hand side is a neutral expression, then the result is also neutral value and its spine is extended with another argument. Other language constructs are handled in a similar way.

Quote In Listing 3.8, we see the second part of the algorithm. In the domain of values, we do not have plain variable terms, or `let-in` bindings, but unevaluated functions and "stuck" neutral terms. A λ-abstraction, in order to be in normal form, needs to have its body also in normal form, therefore we insert a neutral variable into the environment in place of the argument, and eval/quote the body. A neutral term, on the other hand, has at its head a neutral variable. This variable is converted into a term-level variable, and the spine reconstructed as a tree of nested `TApp` applications.

```
fun quote(ctx: Context, v: Val): Term = when (v) {
  is VLocal -> {
    x = TLocal(Ix(ctx.depth - v.head - 1))
    for (vSpine in v.spine.reversed()) {
        x = TApp(x, quote(ctx, vSpine))
    }
    x
  }
  is VLam -> TLam(v.name,
      quote(ctx, eval(ctx, v.cl.body, v.cl.env + VLocal(ctx.lvl)))
  )
  // ...
}
```

Listing 3.8: Demonstration of the `quote` algorithm

These two operations work together, to fully quote a value, we need to also lazily `eval` its sub-terms. The main innovation of the normalization-by-evaluation approach is the introduction of neutral terms, which have the role of a placeholder value in place of a value that has not yet been supplied. As a result, the expression $quote(eval(term, emptyEnv))$ produces a lazily evaluated normal form of a term in a weak head-normal form, with its sub-terms being evaluated whenever accessed. Printing out such a term would print out the fully normalized normal form.

Primitive operations Built-in language constructs like *Nat* or *false* that have not been shown in the snippet are mostly inserted into the initial context as values that can be looked up by their name. In general, though, constructs with separate syntax, e.g. Σ-types, consist of three parts:

- their type is bound in the initial context;
- the term constructor is added to the set of terms and values, and added in `eval()`;

- the eliminator is added as a term and as a spine constructor, i.e., an operation to be applied whenever the neutral value is provided.

The full listing is provided in the supplementary source code, as it is too long to be included in text.

3.4 Elaboration

The second part of the internals of the compiler is type elaboration. Elaboration is the transformation of a partially-specified, well-formed program submitted by a user into a fully-specified, well-typed internal representation [17]. In particular, we will use elaboration to infer types of untyped Curry-style λ-terms, and to infer implicit function arguments that were not provided by the user, demonstrated in Figure 3.10.

$$\begin{array}{rl} \text{function signature:} & id : \{A\} \to A \to A \\ \text{provided expression:} & id\ id\ 5 \\ \text{elaborated expression:} & (id\ \{Nat \to Nat\}\ id)\ \{Nat\}\ 5 \end{array}$$

Figure 3.10: Demonstration of type elaboration

Bidirectional typing Programmers familiar with statically-typed languages like Java are familiar with type checking, in which all types are provided by the user, and therefore are inputs to the type judgment $\Gamma \vdash e : t$. Omitting parts of the type specification means that the type system not only needs to check the types for correctness, but also infer (synthesize) types: the type t in $\Gamma \vdash e : t$ is produced as an output. In some systems, it is possible to omit all type annotations and rely only on the type constraints of built-in functions and literals. Bidirectional systems that combine both input and output modes of type judgment are now a standard approach [37], often used in combination with constraint solving.

Judgments The type system is composed of two additional type judgments we have not seen yet that describe the two directions of computation in the type system:

- $\Gamma \vdash e \Rightarrow t$ is "given the context Γ and term e, infer (synthesize) its type t", and
- $\Gamma \vdash e \Leftarrow t$ is "given the context Γ, term e and type t, check that t is a valid type for t".

The entire typing system described in Chapter 2 can be rewritten using these type judgments. The main principle is that language syntax is divided into two sets of constructs: those that constrain the type of a term and can be checked against an inferred term, and those that do not constrain the type and need to infer it entirely.

Bidirectional λ→-typing In Figure 3.11, this principle is demonstrated on the simply-typed λ-calculus with only variables, λ-abstractions and function application. The first four rules correspond to rules that we have introduced in Chapter 2, with the exception of the constant rule that we have not used there. The two new rules are (**ChangeDir**) and

$$\frac{a : t \in \Gamma}{\Gamma \vdash a \Rightarrow t}\ (\textbf{Var}) \qquad \frac{c \text{ is a constant of type } t}{\Gamma \vdash c \Rightarrow t}\ (\textbf{Const})$$

$$\frac{\Gamma, x : t \vdash e \Leftarrow u}{\Gamma \vdash \lambda x.e \Leftarrow t \to u}\ (\textbf{Abs}) \qquad \frac{\Gamma \vdash f \Rightarrow t \to u \qquad \Gamma \vdash a \Rightarrow t}{\Gamma \vdash f\ a \Rightarrow u}\ (\textbf{App})$$

$$\frac{\Gamma \vdash a \Rightarrow t \qquad \Gamma \vdash a = b}{\Gamma \vdash a \Leftarrow b}\ (\textbf{ChangeDir}) \qquad \frac{\Gamma \vdash a \Leftarrow t}{\Gamma \vdash (a : t) \Rightarrow t}\ (\textbf{Ann})$$

Figure 3.11: Bidirectional typing rules for the λ→-calculus

(**Ann**): (**ChangeDir**) says that if we know that a term has an already inferred type, then we can satisfy any rule that requires that the term checks against a type equivalent to this one. It is also sometimes called the "conversion rule", as it checks whether the terms can be converted into one another. (**Ann**) says that to synthesize the type of an annotated term $a : t$, the term first needs to check against that type.

Rules (**Var**) and (**Const**) produce an assumption, if a term is already in the context or a constant, then we can synthesize its type. In rule (**App**), if we have a function with an inferred type then we check the type of its argument, and if it holds then we can synthesize the type of the application $f\ a$. To check the type of a function in rule (**Abs**), we first need to check whether the body of a function checks against the type on the right-hand side of the arrow.

While slightly complicated to explain, this description produces a provably sound and complete type-checking system [17] that, as a side effect, synthesizes any types that have not been supplied by the user. Extending this system with other language constructs is not complex: the rules used in Montuno for local and global definitions are in Figure 3.12.

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t \qquad \Gamma, x : t \vdash b \Rightarrow u}{\Gamma \vdash \text{let}\, x : t = a \,\text{in}\, b \Rightarrow u}\ (\textbf{Let-In})$$

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t}{\Gamma \vdash x : t = a \Rightarrow t}\ (\textbf{Defn})$$

Figure 3.12: Bidirectional typing rules for `let-in` and top-level definitions

Meta-context One concern was not mentioned in the previous description: when inferring a type, we may not know all its component types: in rule (**Abs**), the type of the function we check may only be constrained by the way it is called. Implicit function arguments $\{A\ B\} \to A \to B \to A$ also only become specific when the function is actually called. The solution to this problem is a *meta-context* that contains *meta-variables*.

These stand for yet undetermined terms [41], either as placeholders to be filled in by the user in interactive proof assistants (written with a question mark, e.g. as $?\alpha$), or terms that can be inferred from other typing constraints using unification. These meta-variables can be either inserted directly by the user in the form of a hole "_", or implicitly, when inferring the type of a λ-abstraction or an implicit function argument [33].

There are several ways of implementing this context depending on the scope of meta-variables, or whether it should be ordered or the order of meta-variables does not matter. A simple-to-implement but sufficiently useful for our purposes is a globally-scoped meta-context divided into blocks placed between top-level definitions.

```
id : {A} → A → A = λx.x
?α = Nat
?β = ?α → ?α
five = (id ?β id) ?α 5
```

Listing 3.9: Meta-context for the expression `id id 5`

The meta-context implemented in Montuno is demonstrated in Listing 3.9. When processing a file, we process top-level expressions sequentially. The definition of the *id* function is processed, and in the course of processing *five*, we encounter two implicit arguments, which are inserted on the top-level as the meta-variables $?\alpha$ and $?\beta$.

3.4.1 Unification

Returning to the rule (**ChangeDir**) in Figure 3.11, a critical piece of the algorithm is how the equivalence of two types is checked. To check a term against a type $\Gamma \vdash a \Leftarrow t$, we first infer a type for the term $\Gamma \vdash a \Rightarrow u$, and then test its equivalence to the wanted type $t = u$.

The usual notion of equivalence in λ-calculus is *α-equivalence of β-normal forms*, that we discussed in Chapter 2, and it corresponds to structural equality of the two terms. *Conversion checking* is the algorithm that determines if two terms are convertible using a set of conversion rules.

As we also use meta-variables in the type elaboration process, these variables need to be solved in some way. This process of conversion checking together with solving meta-variables is called *unification* [26], and is a well-studied problem in the field of type theory.

Pattern unification In general, solving meta-variables is undecidable [1]. Given the constraint $?\alpha\ 5 = 5$, we can produce two solutions: $?\alpha = \lambda x.x$ and $?\alpha = \lambda x.5$. There are several possible approaches and heuristics: first-order unification solves for base types and cannot produce functions as a result; higher-order unification can produce functions but is undecidable; *pattern unification* is a middle ground and with some restrictions, it can produce functions as solutions.

In this thesis, I have chosen to reuse an existing algorithm [36] which, in brief, assumes that a meta-variable is a function whose arguments are all local variables in scope at the moment of its creation. Then, when unifying the meta-variable with another (non-variable) term, it builds up a list of variables the term uses, and stores such a solution as a *renaming* that maps the meta-variable arguments to the variables in the term which it was unified with. As the algorithm is rather involved but tangential to the goals of this thesis, I will omit a detailed description and instead point an interested reader at the original source [36].

3.4.2 Implementation

As with the implementation of normalization-by-evaluation, we will look at the most illustrative parts of the implementation. This time, the comparison can be made directly side-by-side, between the bidirectional typing algorithm and its implementation.

What was not mentioned explicitly so far is that the type elaboration algorithm has as its input `PreTerms`, and produces `Terms` in the case of type checking, and pairs of `Terms` and `Values` (the corresponding types) in the case of type inference. Unification (not demonstrated here) is implemented as parallel structural recursion over two `Value` objects.

In Figure 3.13, we see the previously described rule that connects the checking and synthesis parts of the algorithm and uses unification. Unification solves meta-variables as a side-effect, here it is only in the role of a guard as it does not produce a value. The code exactly follows the typing rule: the type of the pre-term is inferred, resulting in a well-typed term and its type. The type is unified with the "wanted" type and, if the unification successful, the rule produces the inferred term.

$$\frac{\Gamma \vdash a \Rightarrow t \qquad \Gamma \vdash a = b}{\Gamma \vdash a \Leftarrow b}\ (\textbf{ChangeDir})$$

```
fun LocalContext.check(pre: PreTerm, wanted: Value): Term = when (pre) {
  // ...
  else -> {
    val (t, actual) = infer(pre.term)
    unify(actual, wanted)
    t
  }
}
```

Figure 3.13: Side-by-side comparison of the **ChangeDir** rule

Figure 3.14 shows the exact correspondence between the rule and its implementation, one read left-to-right, the other top-to-bottom. Checking of the type and value are straightforward, translation of $\Gamma, x : t \vdash b \Rightarrow u$ binds a local variable in the environment, so the body of the `let-in` expression can be inferred, and the result is a term containing the inferred body and type, wrapped in a `TLet`.

$$\frac{\Gamma \vdash t \Leftarrow \star \qquad \Gamma \vdash a \Leftarrow t \qquad \Gamma, x : t \vdash b \Rightarrow u}{\Gamma \vdash \text{let } x : t = a \text{ in } b \Rightarrow u}\ (\textbf{Let-In})$$

```
fun LocalContext.infer(pre: PreTerm): Pair<Term, Value> = when (pre)
  is RLet -> {
    val t = check(pre.type, VStar)
    val a = check(pre.defn, t)
    val (b, u) = localDefine(pre.name, a, t).infer(pre.body)
    TLet(pre.name, t, a, b) to u
  } // ...
}
```

Figure 3.14: Side-by-side comparison of the **Let-in** rule

Lastly, the rule for a term-level λ-abstraction is demonstrated in Figure 3.15. This rule demonstrates the creation of a new meta-variable as without a placeholder, so we are not able to infer the type of the body of the function. This meta-variable might be solved in the course of inferring the body.

$$\frac{\Gamma, x : t \vdash e \Leftarrow u}{\Gamma \vdash \lambda x.e \Leftarrow \Pi t : *.u}\ (\textbf{Abs})$$

```
fun LocalContext.infer(pre: PreTerm): Pair<Term, Value> = when (pre)
  is RLam -> {
    val a = newMeta()
    val (b, t) = localBind(pre.name, a).infer(pre.body)
    TLam(pre.name, b) to VPi(pre.name, a, VCl(env, t.quote()))
  } // ...
}
```

Figure 3.15: Side-by-side comparison of the **Abs** rule

3.5 Driver

This concludes the complex part of the interpreter, what follows are rather routine concerns. Next part of the implementation is the driver that wraps the backend, and handles its interaction with the surrounding world. In particular, this includes the parser, pretty-printer, and state management.

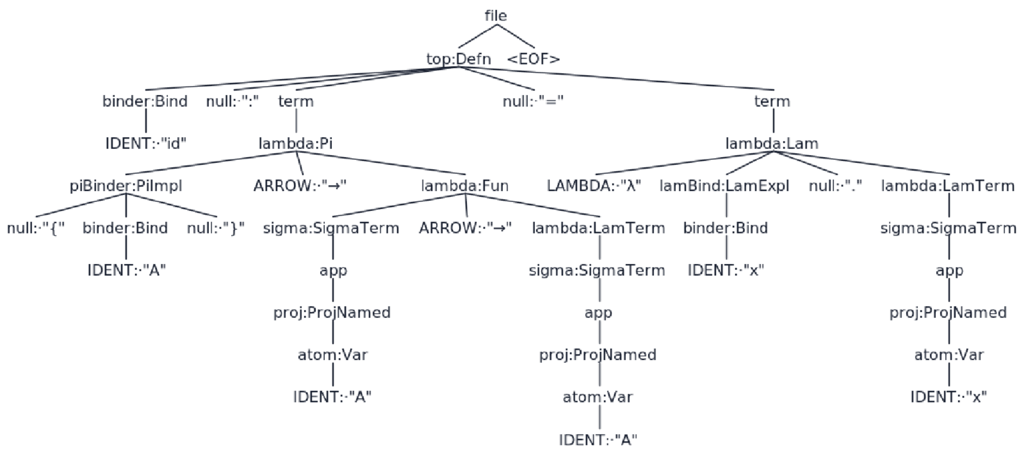


Figure 3.16: Parse tree of the `id` function

Parser Lexical and syntactic analysis is not the focus of this work, so simply I chose the most prevalent parsing library in Java-based languages, which seems to be ANTLR[6]. It comes with a large library of languages and protocols from which to take inspiration[7], so creating the parser was a rather simple matter.

[6]`https://www.antlr.org/`
[7]`https://github.com/antlr/grammars-v4/`

The grammar from the previous chapter is translated into ANTLR grammar almost directly, as it also uses syntax that reminds the Backus-Naur form. The result of parsing the `id` function $id : A \to A \to A = \lambda x.x$ is shown in Figure 3.16. This tree includes many redundant contexts, and needs to be translated to a simpler representation of a `PreTerm`, which can then be directly used as input for elaboration.

ANTLR provides two recommended ways of consuming the result of parsing using classical object-oriented design patterns: a listener and a visitor. I used neither as they were needlessly verbose or limiting[8]. Instead of these, a custom recursive-descent AST transformation was used that is demonstrated in Listing 3.10. This directly transforms the `ParseContext` objects created by ANTLR into our `PreTerm` data type.

```
fun TermContext.toAst(): PreTerm = when (this) {
  is Let -> RLet(id.toAst(), type.toAst(), defn.toAst(), body.toAst())
  is Lam -> rest.foldRight(binder.toAst()) { l, r -> RLam(l.toAst(), r) }
  is Pi -> rest.foldRight(binder.toAst()) { l, r -> l.toAst()(r) }
  is App -> spine.fold(head.toAst()) { l, r -> r.toAst()(l) }
  else -> throw UnsupportedOperationException(javaClass.canonicalName)
}
```

Listing 3.10: Parser to `PreTerm` transformation as a depth-first traversal

The data type itself is shown in Listing 3.11. As with terms and values, it is a recursive data structure, presented here in a slightly simplified manner compared to the actual implementation, as it omits the part that tracks the position of a term in the original source. The grammar that is used as the source for the parser generator ANTLR was already presented once in the conclusion of Chapter 2, so the full listing is only included in Appendix A.

```
sealed class TopLevel
class RDecl(val n: String, val type: Pre) : TopLevel()
class RDefn(val n: String, val type: Pre?, val term: Pre) : TopLevel()
class RTerm(val cmd: Pragma?, val term: Pre) : TopLevel()

sealed class Pre
object RU : Pre()
class RVar(val n: String) : Pre()
class RApp(val lhs: Pre, val rhs: Pre) : Pre()
class RLam(val n: String?, val body: Pre) : Pre()
class RPi(val n: String?, val type: Pre, val body: Pre) : Pre()
```

Listing 3.11: Snippet of the data type `PreTerm` (abbreviated to `Pre` for type-setting)

Pretty-printer A so-called pretty-printer is a transformation from an internal representation of a data structure to a user-readable string representation. The implementation of such a transformation is mostly straight-forward, complicated only by the need to correctly handle operator precedence and therefore parentheses.

This part is implemented using the Kotlin library `kotlin-pretty`, which is itself inspired by the Haskell library `prettyprinter` which, among other things, handles correct block

[8]In particular, ANTLR-provided visitors require that all return values share a common super-class. Listeners do not allow return values and would require explicit parse tree manipulation.

indentation and ANSI text coloring: that functionality is also used in error reporting in the terminal interface.

An excerpt from this part of the implementation is included in Listing 3.12, which demonstrates the pretty-printing of function application, and some constructions of the `kotlin-pretty` library.

```
fun Term.pretty(ns: NameEnv?, parens: Boolean): Doc = when (this) {
  is TVar -> ns[ix].text()
  is TApp -> par(parens, lhs.pretty(ns, true) + " ".text() + when (icit) {
      Icit.Impl -> "{".text() + rhs.pretty(ns, false) + "}".text()
      Icit.Expl -> rhs.pretty(ns, true)
  })
  is TLet -> {
    val d = listOf(
      ":".text() spaced ty.pretty(ns, false),
      "=".text() spaced bind.pretty(ns, false),
    ).vCat().align()
    val r = listOf(
      "let $n".text() spaced d,
      "in".text() spaced body.pretty(ns + n, false)
    ).vCat().align()
    par(parens, r)
  } // ...
}
```

Listing 3.12: Pretty-printer written using `kotlin-pretty`

State management Last component of the driver code is global interpreter state, which consists mainly of a table of global names. This table is required for handling incremental interpretation or suggestions (tab-completion) in the interactive environment. The global context also contains the meta-context, and tracks the position of the currently evaluated term in the original source file for error reporting.

Overall, the driver receives user input in the form of a string, parses it, supplies it expression by expression to the backend, receiving back a global name, or an evaluated value, which it pretty-prints and returns back to the user-facing frontend code.

3.6 Frontend

We will consider only two forms of user interaction: batch processing of a file via a command-line interface, and a terminal environment for interactive use. Later, with the Truffle interpreter, we can also add an option to compile a source file into an executable using Truffle's capability to produce *Native Images*.

CLI We will reuse the entry point of Truffle languages, a `Launcher` class, so that integration of the Truffle interpreter is easier later, and then we are able to create single executable that is able to use both interpreters.

`Launcher` handles pre-processing command-line arguments for us, a feature for which we would otherwise use an external library like `JCommander`. In the Truffle interpreter, we will also use the *execution context* it prepares using various JVM options but for now, we will only use `Launcher` for argument processing.

Two modes of execution are implemented, one mode that processes a single expression provided on the command line and `--normalize`s it, `--elaborate`s it, or find its `--type`. The second mode is sequential batch processing mode that reads source code either from a file or from standard input, and processes all statements and commands in it sequentially.

As we need to interact with the user we encounter another problem, that of error reporting. It has been mentioned in passing several times, and in this implementation of the interpreter, it is handled only partially. To report an error well, we need its cause and location. Did the user forget to close a parenthesis, or is there a type error and what can they do to fix it? Syntactic errors are reported well in this interpreter, but elaboration errors only sometimes.

Error tracking pervades the entire interpreter, position records are stored in all data structures, location of the current expression is tracked in all evaluation and elaboration contexts, and requires careful placement of update commands and throwing and catching of exceptions. As error handling is implemented only passably and is not the focus of this thesis, it is only mentioned briefly here.

In Listing 3.13, a demonstration of the command-line interface is provided: normalization of an expression, batch processing of a file, and finally, starting up of the REPL.

```
$> cat demo.mt
id : {A} -> A -> A = \x. x
const : {A B} -> A -> B -> A = \x y. x
{-# TYPE id #-}
{-# NORMALIZE id const #-}

$> montuno demo.mt
{A} → A → A
λ x _. x
$> montuno --type id
{A} → A → A
```

Listing 3.13: Example usage of the CLI interface

REPL Read-Eval-Print Loop is the standard way of implementing interactive terminal interfaces to programming languages. The interpreter receives a string input, processes it, and writes out the result. There are other concerns, e.g., implementing name completion, different REPL-specific commands or, in our case, switching the backend of the REPL at runtime.

From my research, JLine is the library of choice for interactive command-line applications in Java, so that is what I used. Its usage is simple, and implementing a basic interface takes only 10s of lines. The commands reflect the capabilities of the command-line interface: (re)loading a file, printing out an expression in normalized or fully elaborated forms, and printing out the type of an expression. These are demonstrated in a simple way in Listing 3.14.

```
$> montuno
Mt> :load demo.mt
Mt> <TAB><TAB>
id const true false cond
Mt> :normalize id const
λ x _. x
Mt> :type id
{A} -> A -> A
Mt> :quit
```

Listing 3.14: REPL session example

Chapter 4

Adding JIT compilation to Montuno: MontunoTruffle

4.1 Introduction

In the first part of this thesis, we introduced the theory of dependent types, specified a small, dependently typed language, and introduced some of the specifics of creating an interpreter for this language, under the name Montuno. The second part is concerned with the Truffle language implementation framework: we will introduce the framework itself and the features it provides to language designers, and use it to build a second interpreter.

To reiterate the goal of this thesis, the intent is to create a vehicle for evaluating whether adding just-in-time compilation produces visible improvements in the performance of dependently typed languages. Type elaboration is often a performance bottleneck [24], and because it involves evaluation of terms, it should be possible to improve it using JIT compilation; as optimizing AST evaluation is a good candidate for JIT compilation. We have designed a language that uses features and constructs that are representative of state-of-the-art proof assistants and dependently typed languages, so that such evaluation may be used as a guideline for further work.

This chapter is concerned with building a second interpreter based on Truffle. First, however, we need to introduce the idea of just-in-time compilation in general, and see how Truffle implements the concept.

4.2 Just-in-time compilation

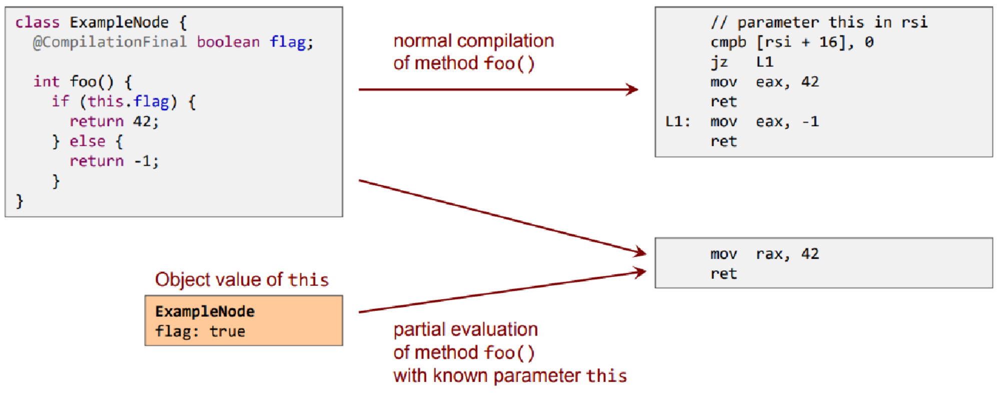
Just-in-time (JIT) compilation, in general, is a technique that combines an interpreter and a compiler into a single runtime. A program is first interpreted, and later compiled during its runtime. The JIT compiler often observes the behavior of the interpreted program, so that it can compile it more efficiently. While a program is running, the JIT compiler optimizes the parts that run often; using an electrical engineering metaphor, such parts are sometimes called *"hot loops"*.

The optimizations often rely on assumption that, when executing a program, its functions (and the functions in the libraries it uses) are only called in a specific pattern, configuration, or with a specific type of data. When talking about specific optimizations, the terms *slow path* and *fast path* are often used. The fast path is the one for which the program is currently optimized, whereas the slow paths are all the other ones, e.g., function calls or branches that were not used during the specific program execution.

There are several approaches to JIT compilation: *meta-tracing* and *partial evaluation* are the two most common ones.

Meta-tracing A JIT compiler based on meta-tracing records a *trace* of the path taken during program execution. Often used paths are then optimized: either rewritten, or directly compiled to machine code. Tracing, however, adds some overhead to the runtime of the program, so only some paths are traced. While the programmer can provide hints to the compiler, meta-tracing may result in unpredictable peak performance. This technique has been successfully used in projects like PyPy, that is built using the RPython JIT compiler [7], or on GHC with mixed results [44].

Partial evaluation The second approach to JIT compilation is called *partial evaluation*, also called the *Futamura projection*. The main principle is as follows: (fully) evaluating a program using an interpreter produces some output, whereas partially evaluating a program using an interpreter produces a specialized executable. The specializer assumes that the program is constant and can, e.g., eliminate parts of the interpreter that will not be used by the program. This is the approach taken by Truffle [34].



```
class ExampleNode {
  @CompilationFinal boolean flag;

  int foo() {
    if (this.flag) {
      return 42;
    } else {
      return -1;
    }
}
```

```
      // parameter this in rsi
      cmpb [rsi + 16], 0
      jz   L1
      mov  eax, 42
      ret
L1:   mov  eax, -1
      ret
```

```
ExampleNode
flag: true
```

```
      mov  rax, 42
      ret
```

Figure 4.1: Partial evaluation with constant folding[1]

The basic principle is demonstrated in Figure 4.1, on actual code produced by Truffle. In its vocabulary, a `CompilationFinal` value is assumed to be unchanging for a single instance of the program graph node (the field `flag` in the figure), and so the JIT compiler can transform a conditional `if` statement into an unconditional one, eliminating the second branch.

[1]Source: Graal: High Performance Compilation for Managed Languages [52]

There are, in fact, three Futamura projections, referred to by their ordinals: the *first Futamura projection* specializes an interpreter with regards to a program, producing an executable. The *second Futamura projection* combines the specializer itself with an interpreter, producing a compiler. The third projection uses the specializer on itself, producing a compiler maker. As we will see in later sections, Truffle and GraalVM implement both the first and second projections [34].

4.3 Truffle and GraalVM

I have mentioned Truffle several times already in previous chapters. To introduce it properly, we first need to take a look at the Java Virtual machine (JVM). The JVM is a complex platform that consists of several components: a number of compilers, a memory manager, a garbage collector, etc., and the entire purpose of this machinery is to execute `.class` files that contain the bytecode representation of Java, or other languages that run on the JVM platform. During the execution of a program, code is first translated into generic executable code using a fast C1 compiler. When a specific piece of code is executed enough times, it is further compiled by a slower C2 compiler that performs more expensive optimizations, but also produces more performant code.

The HotSpotVM is one such implementation of this virtual machine. The GraalVM project, of which Truffle is a part, consists of several components and the main one is the Graal compiler. It is an Oracle research project that replaces the C2 compiler inside HotSpotVM, to modernize an aging code base written in C++, and replace it with a modern one built with Java [11]. The Graal compiler is used in other ways, though, some of which are illustrated in Figure 4.2. We will now look at the main ones.

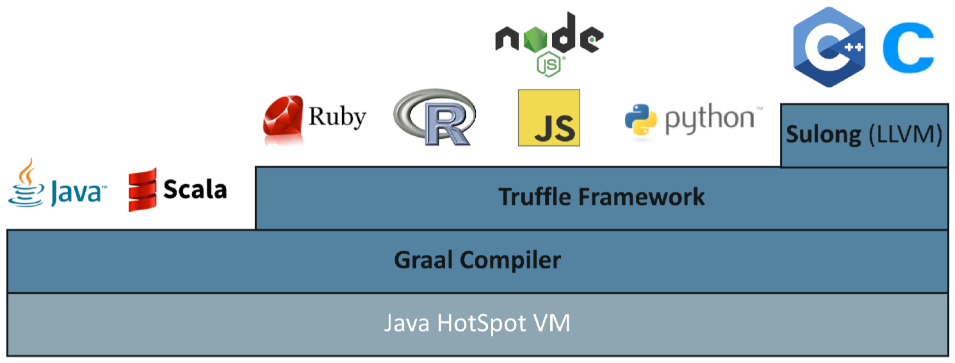


Figure 4.2: GraalVM components and Truffle[2]

Graal Graal itself is at its core a graph optimizer applied to program graphs. It processes Java bytecode into a graph of the entire program, spanning across function calls, and reorders, simplifies, and overall optimizes it.

[2]Source: `https://www.graalvm.org/community/assets`

It actually builds two graphs in one: a data-flow graph, and an instruction-flow graph. Data-flow describes what data is required for which operation, which can be reordered or optimized away, whereas the instruction-flow graph stores the actual order of instructions that will happen on the processor. Figure 4.3 shows the output of a graph visualization tool provided by Graal. This specific graph is the execution path of a `CounterNode` that simply reads its internal field, adds a one to it, and stores the result; more complex program graphs are often very large and hard to read.

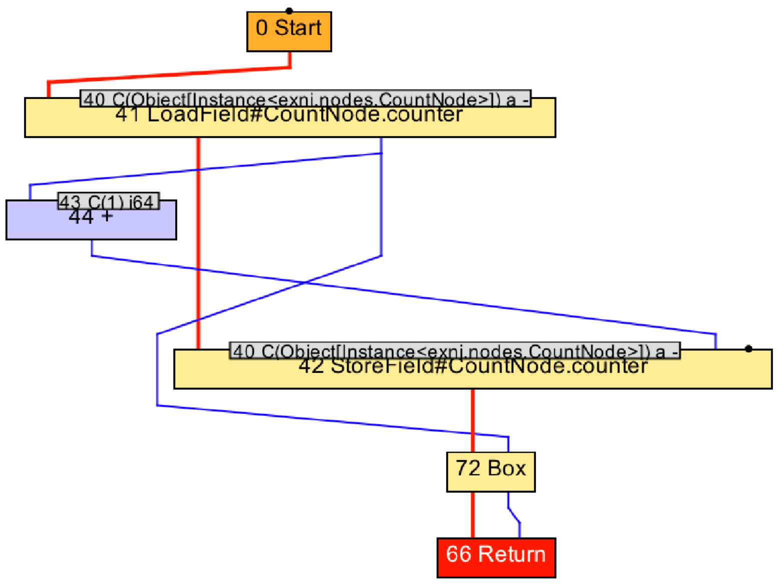

0 Start
40 C(Object[Instance<exni.nodes.CountNode>]) a -
41 LoadField#CountNode.counter
43 C(1) i64
44 +
40 C(Object[Instance<exni.nodes.CountNode>]) a -
42 StoreField#CountNode.counter
72 Box
66 Return

Figure 4.3: Graal program graph, visualized using IGV[3]

SubstrateVM As Graal is a standalone Java library, it can also be used in contexts other than the HotSpotVM. SubstrateVM is an alternative virtual machine that executes Graal-optimized code. It does not perform just-in-time optimizations, though, but uses Graal as an ahead-of-time compiler. The result is a small stand-alone executable file that does not depend on a JVM being installed on a machine, called a *Native Image*. By replacing JIT compilation with ahead-of-time, these binaries start an order-of-magnitude faster than regular Java programs, and can be freely copied between machines, similar to Go or Rust binaries [55].

Truffle The Graal program graph, Graal IR, is a directed graph structure in static single assignment form. As it is implemented in Java itself, the graph structure is extensible [11]. Truffle exposes this extensibility of the program to developers. In essence, it is a graph manipulation library and a set of utilities for creating these graphs. These graphs are the abstract syntax tree of a language: each node has an `execute` method; calling the method returns the result of evaluating the expression it represents.

[3]Source: Graal: High Performance Compilation for Managed Languages [52]

Interpreter/compiler When creating a programming language, there is a trade-off between writing an interpreter and a compiler. An interpreter is usually simpler to implement and each function in the host language directly encodes the semantics of a language construct, but the result can be rather slow: compared to the language in which the interpreter is written, it can often be slower by a factor to 10x to 100x [55]. A compiler, on the other hand, does not execute a program directly, but instead translates its semantics onto the semantics of a different virtual machine, be it the JVM, LLVM, or x86 assembly.

Truffle attempts to side-step this trade-off by making it possible to create an interpreter that can be compiled on-demand via JIT when interpreted or ahead-of-time into a Native Image; the result should be an interpreter-based language implementation with the performance of a compiled language and access to all JVM capabilities (e.g. memory management). Instead of running an interpreter inside a host language like Java, the interpreter is embedded one layer lower, into a program graph that runs directly on the JVM and is manipulated by the Truffle runtime that runs next to it.

Polyglot Truffle languages can all run next to one another on the JVM. As a side-effect, communication between languages is possible without the need for usual FFI (foreign function interface) complications. As all values are JVM objects, access to object properties uses the same mechanisms across languages, as does function invocation. In effect, any language from Figure 4.2 can access libraries and values from any other such language.

TruffleDSL Truffle is a runtime library that manages the program graph and a number of other concerns like variable scoping, or the object storage model that allows objects from different languages to share the same layout. TruffleDSL is a user-facing library in the form of a domain-specific language (DSL) that aids in simplified construction specialized Truffle node classes, inline caches, language type systems, and other specifics. This DSL is in the form of Java *annotations* that give additional information to classes, methods, or fields, so that a DSL processor can later use them to generate the actual implementation details.

Instrumentation The fact that all Truffle languages share the same basis, the program graph, means that a shared suite of tooling could be built on top of it: a profiler (VisualVM), a stepping debugger (Chrome Debugger), program graph inspector (IGV), a language server (Graal LSP). We will use some of these tools in further sections.

4.4 Truffle in detail

Concluding the general introduction to Truffle and GraalVM, we will now look at the specifics of how a Truffle language differs from the type of interpreter we created previously.

The general concept is very similar to the previously created AST interpreter: there is again a tree data structure at the core, where each node corresponds to one expression that can be evaluated. The main differences are in a number of details that were previously implicit, though, like the simple action of "calling a function", which in Truffle involves the interplay of, at a minimum, five different classes.

Figure 4.4 shows the components involved in the execution of a Truffle language. Most of our work will be in the parts labeled "AST", "AST interpreter", and "AST rewriting". All of these involve the contents of the classes that form the abstract syntax tree, as individual graph nodes contain their data, but also their interpretation and rewriting specifics.

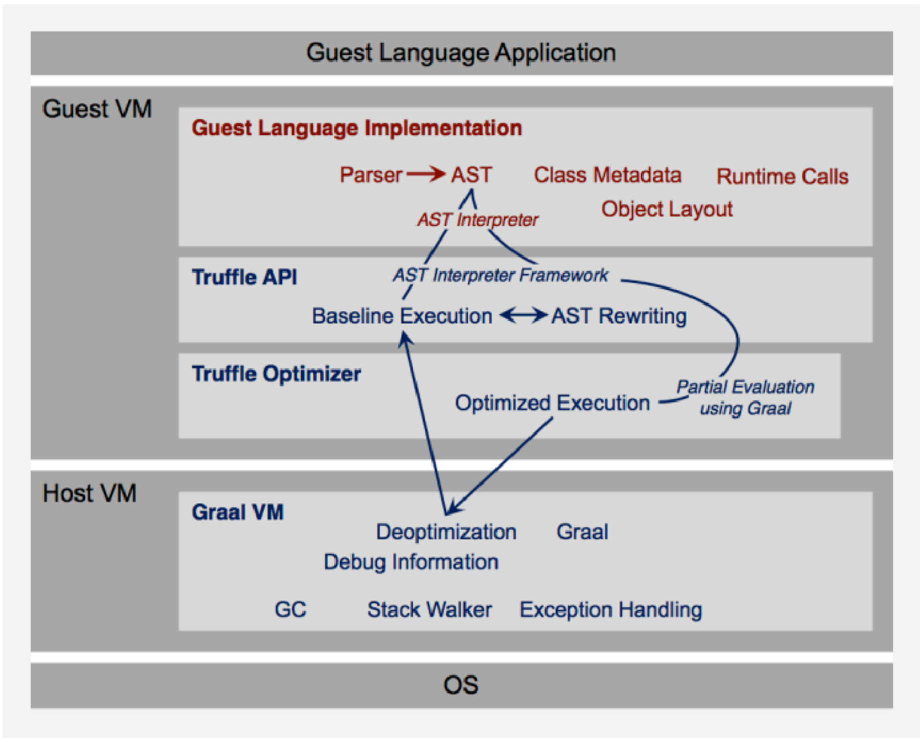


Figure 4.4: Architecture of a Truffle language, arrows denote program execution flow[4]

Overall, the implementation of a Truffle language can be divided into a few categories. Some of the classes to be sub-classed and methods to be implemented are included in parentheses to give a brief idea of the terminology we will use, although we will expand on each one momentarily. These blocks are:

- language execution (`Launcher`),
- language registration (`Language`, `Context`, `ParsingRequest`),
- program entry point (`RootNode`, `CallTarget`),
- node execution (`VirtualFrame`, `execute`, `call`),
- node specialization (`Specialization`, `Profile`, `Assumption`),
- value types (`TypeSystem`, `ValueType`),
- compiler directives (`transferToInterpreter`, `TruffleBoundary`),

[4]Source: Graal: High Performance Compilation for Managed Languages [52]

- function calls (`InvokeNode`, `DispatchNode`, `CallNode`),
- object model (`Layout`, `Shape`, `Object`), and
- others (instrumentation, `TruffleLibrary` interfaces, threads).

Launcher The entry point to a Truffle language is a `Launcher` (Listing 4.1). This component handles processing command-line arguments, and uses them to build a language execution context. A language can be executed from Java directly without a `Launcher`, but it handles all GraalVM-specific options and switches, many of which we will use later, and correctly builds a language execution environment, including all debugging and other tools that the user may decide to use.

```
class MontunoLauncher : AbstractLanguageLauncher() {
  companion object {
    @JvmStatic fun main(args: Array<String>) = Launcher().launch(args)
  }
  override fun getDefaultLanguages() = arrayOf("montuno");
  override fun launch(contextBuilder: Context.Builder) {
    contextBuilder.arguments(getLanguageId(), programArgs)
    Context context = contextBuilder.build()
    Source src = Source.newBuilder(getLanguageId(), file).build()
    Value returnVal = context.eval(src)
    return returnVal.execute().asInt()
  }
}
```

Listing 4.1: A minimal language `Launcher`

Language registration The programming language is represented by a `Language` object, whose primary purpose is to answer `ParsingRequest`s with the corresponding program graphs, and to manage execution `Context`s that contain global state of a single language process. It also specifies general language properties like support for multi-threading, or the MIME type and file extension, and decides which functions and objects are exposed to other Truffle languages.

```
@TruffleLanguage.Registration(
  id = "montuno", defaultMimeType = "application/x-montuno"
)
class Language : TruffleLanguage<MontunoContext>() {
  override fun createContext(env: Env) = MontunoContext(this)
  override fun parse(request: ParsingRequest): CallTarget {
    CompilerAsserts.neverPartOfCompilation()
    val root = ProgramRootNode(parse(request.source))
    return Truffle.getRuntime().createCallTarget(root)
  }
}
```

Listing 4.2: A minimal `Language` registration

Program entry point Listing 4.2 demonstrates both a language registration and the creation of a `CallTarget`. A call target represents the general concept of a *callable object*, be it a function or a program, and as we will see later, a single call to a call target corresponds to a single stack `VirtualFrame`. It points to the `RootNode` at the entry point of a program graph, as shown in Figure 4.5.

A `CallTarget` is also the basic optimization unit of Truffle: the runtime tracks how many times a `CallTarget` was entered (called), and triggers optimization (partial evaluation) of the program graph as soon as a threshold is reached.

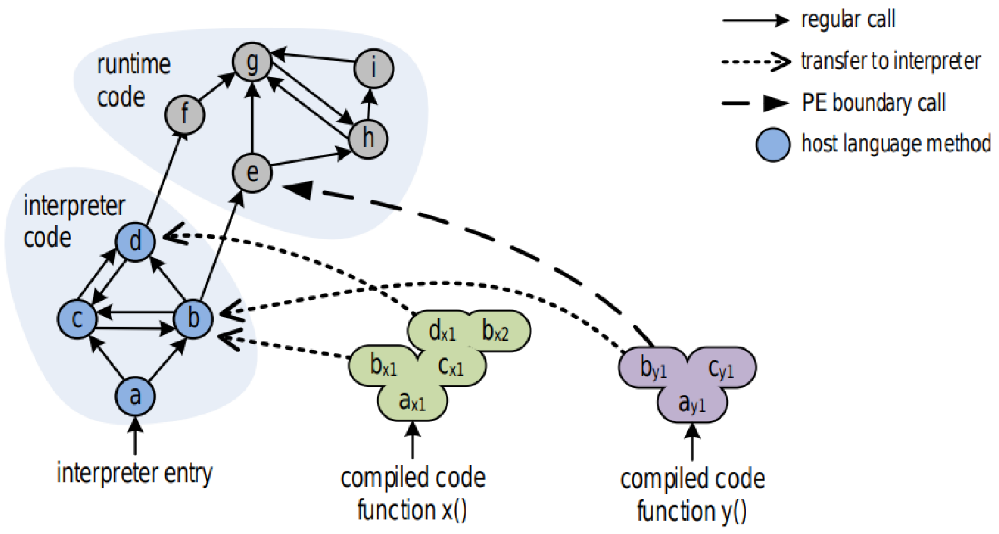


Figure 4.5: Combination of regular and partially-evaluated code[5]

Node execution A `RootNode` is a special case of a Truffle `Node`, the basic building block of the program graph. Each node has a single way of evaluating the expression it represents, the `execute` method. We may see nodes with multiple `execute` methods later, but they are all ultimately translated by the Truffle DSL processor into a single method: Truffle will pick the most appropriate one based on the methods' return type, arguments types, or user-provided *guard* expressions.

Listing 4.3 contains an example of two nodes. They share a parent class, `LanguageNode`, whose only method is the most general version of `execute`: one that takes a virtual frame and returns anything. An `IntLiteralNode` has only one way of providing a result, it returns the literal value it contains. `AddNode`, on the other hand, can add either integers or strings, so it uses another Truffle DSL option, a `@Specialization` annotation, which then generates the appropriate logic for choosing between the methods `addInt`, `addString`, and `typeError`.

Specialization Node specialization is one of the main optimization capabilities of Truffle. The `AddNode` in Listing 4.3 can handle strings and integers both, but if it only ever receives integers, it does not need to check whether its arguments are strings on the *fast path* (the

[5]Source: Graal: High Performance Compilation for Managed Languages [52]

```
abstract class LanguageNode : Node() {
  abstract fun execute(frame: VirtualFrame): Any
}
class IntLiteralNode(private val value: Long) : LanguageNode() {
  override fun execute(frame: VirtualFrame): Any = value
}
abstract class AddNode(
  @Child val left: LanguageNode, @Child val right: LanguageNode,
) : LanguageNode() {
  @Specialization fun addInt(l: Int, r: Int) = l + r
  @Specialization fun addString(l: String, r: String) = l + r
  @Fallback fun typeError(l: Any?, r: Any?): Unit
    = throw TruffleException("type error")
}
```

Listing 4.3: Addition with type specialization

currently optimized path). Using node specialization, the `AddNode` can be in one of four states: uninitialized, integers-only, strings-only, and both generic. Whenever it encounters a different combination of arguments, a specialization is *activated*. Overall, the states of a node form a directed acyclic graph: a node can only ever become more general, as the Truffle documentation emphasizes.

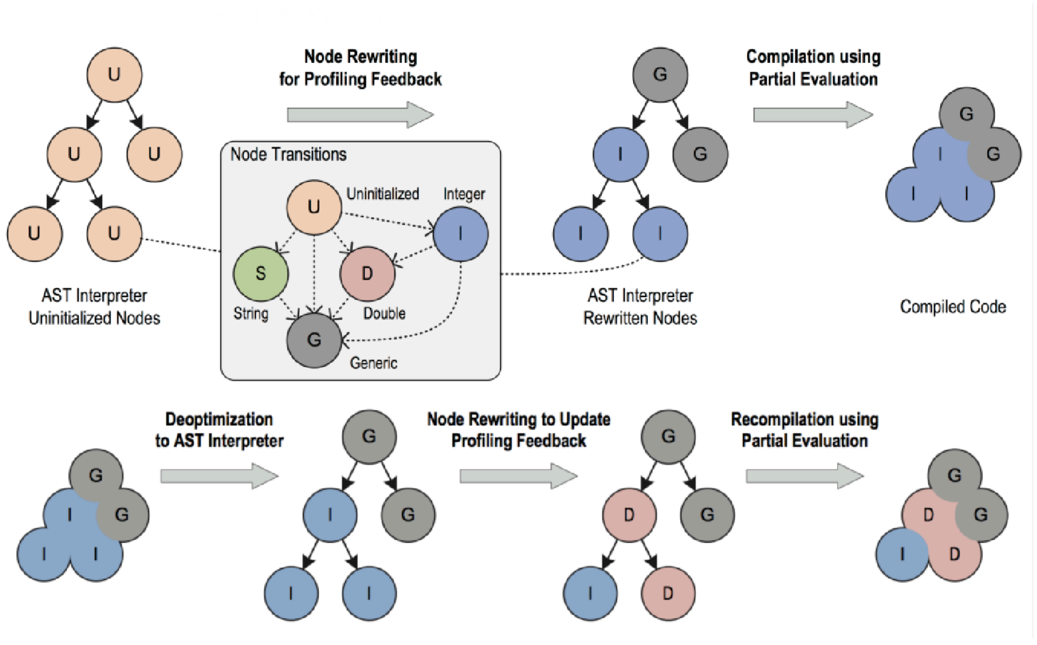

Node Rewriting for Profiling Feedback
Compilation using Partial Evaluation
Node Transitions
Uninitialized
Integer
String
Double
Generic
AST Interpreter Uninitialized Nodes
AST Interpreter Rewritten Nodes
Compiled Code
Deoptimization to AST Interpreter
Node Rewriting to Update Profiling Feedback
Recompilation using Partial Evaluation
U
I
G
S
D

Figure 4.6: Node optimization and deoptimization in Truffle[6]

(De)optimization Node specialization combined with the optimization of a `CallTarget` when called enough times are sufficient to demonstrate the process of JIT compilation in Truffle. Figure 4.6 demonstrates this process on a node type with several more state transitions. When all nodes in a program graph reach a stable state where no more special-

[6]Source: One VM to Rule Them All [55]

izations take place, it is may be partially evaluated. This produces efficient machine code instead of slow interpreter-based code, specialized for the nodes' current states.

However, this compilation is *speculative*, it assumes that nodes will not encounter different values, and this is encoded in explicit *assumption* objects. When these assumptions are invalidated, the compiled machine code is discarded, and the nodes revert back to their non-optimized form. This process is called *deoptimization* [53], and can be explicitly invoked using the Truffle method `transferToInterpreter`.

After a deoptimization, the states of nodes should again stabilize, so that they may be partially evaluated into efficient machine code once more. Often, this (de)optimization process repeats multiple times during the execution of a single program: the period from the start of a program until a stable state is called the *warm-up* phase.

Value types Nodes can be specialized based on various criteria, but the above-mentioned specialization with regards to the type of arguments requires that these types are all declared and aggregated into a `TypeSystem` object and annotation. These are again processed by Truffle DSL into a class that can check the type of a value (`isUnit`), and perform implicit conversion between them (`asBoolean`, `castLong`). Listing 4.4 demonstrates a `TypeSystem` with a custom type `Unit` and the corresponding required `TypeCheck`, and with an implicit type-cast in which an integer is implicitly convertible into a long integer.

```
@CompilerDirectives.ValueType
object Unit

@TypeSystem(Unit::class, Boolean::class, Int::class, Long::class)
open class Types {
  companion object {
    @ImplicitCast
    fun castLong(value: Int): Long = value.toLong()
    @TypeCheck(Unit::class)
    fun isUnit(value: Any): Boolean = value === Unit
  }
}
```

Listing 4.4: A `TypeSystem` with an implicit cast and a custom type

Function invocation An important part of the implementation of any Truffle language consists of handling function calls. A common approach in multiple Truffle is as follows: Given an expression like `fibonacci(5)`. This expression is evaluated in multiple steps: an `InvokeNode` resolves the function that the expression refers to (`fibonacci`) into a `CallTarget`, and evaluates its arguments (`5`). A `DispatchNode` creates a `CallNode` for the specific `CallTarget` and stores it in a cache. Finally, a `CallNode` is what actually performs the switch from one part of the program graph to another, building a stack `Frame` with the function's arguments, and entering the `RootNode` referred to by the `CallTarget`.

Stack frames `Frames` were mentioned several times already: they are Truffle's abstraction of a stack frame. In general, stack frames contain variables and values in the local scope of a function, those that were passed as its arguments and those declared in its body.

In Truffle, this is encoded as a `Frame` object, and passed as an argument to all `execute` functions. Frame layout is set by a `FrameDescriptor` object, which contains `FrameSlot`s that refer to parts of the frame. Listing 4.5 demonstrates two nodes that interact with a `Frame`: a reference to a local variable, and a local variable declaration.

```
class ReadLocalVarNode(val name: String) : Node {
  fun execute(frame: VirtualFrame): Any {
    val slot: FrameSlot = frame.getFrameDescriptor().findFrameSlot(name)
    return frame.getValue(slot ?: throw TruffleException("not found"));
} }
class WriteLocalVarNode(val name: String, val body: Node) : Node {
  fun execute(frame: VirtualFrame): Unit {
    val slot: FrameSlot = frame.getFrameDescriptor().addFrameSlot(name)
    frame.setObject(slot, body.execute(frame));
} }
```

Listing 4.5: Basic operations with a `Frame`

There are two kinds of a `Frame`, virtual and materialized frames. A `VirtualFrame` is, as its name suggests, virtual, and its values can be freely optimized by Truffle, reorganized, or even passed directly in registers without being allocated on the heap (using a technique called Partial Escape Analysis). A `MaterializedFrame` is not virtual, it is an object at the runtime of a program, and it can be stored in program's values or nodes. A virtual frame is preferable in almost all cases, but e.g., implementing closures requires a materialized frame, as it needs to be stored in a `Closure` object. This is shown in Listing 4.6, where `frame.materialize()` captures a virtual frame and stores it in a closure.

```
@CompilerDirectives.ValueType
data class Closure(
  val callTarget: CallTarget,
  val frame: MaterializedFrame,
)
class ClosureNode(val ct: CallTarget) : Node {
  fun executeClosure(frame: VirtualFrame) {
    return Closure(ct, frame.materialize())
  }
}
```

Listing 4.6: A closure value with a `MaterializedFrame`

Caching These were the main features required for writing a Truffle language, but there are several more tools for their optimization, the first one being *inline caching*. This is an old concept that originated in dynamic languages, where it is impossible to statically determine the call target in a function invocation, so it is looked up at runtime. Most function call sites use only a limited number of call targets, so these can be cached. As the cache is a local one, placed at the call site itself, it is called an *inline cache*. This concept is used for a number of other purposes, e.g., caching the `FrameSlot` in an assignment operator, or the `Property` slot in an object access operation.

In the case of function dispatch, a `DispatchNode` goes through the following stages: *uninitialized*; *monomorphic*, when it is specialized to a single call target; *polymorphic*, when it stores a number of call targets small enough that the cost of searching the cache is smaller than the cost of function lookup; and *megamorphic*, when the number of call targets exceeds the size of the cache, and every function call is looked up again. Listing 4.7 demonstrates this on a DispatchNode, adding a polymorphic cache with size 3, and also demonstrates the Truffle DSL annotations Cached. The cache key is the provided CallTarget, based on which a DirectCallNode is created and cached as well. The megamorphic case uses an IndirectCallNode: in a DirectCallNode, the call target can be inlined by the JIT compiler, whereas in the indirect version it can not.

```
abstract class DispatchNode : Node {
  abstract fun executeDispatch(
    frame: VirtualFrame, callTarget: CallTarget, args: Array<Any>): Any

  @Specialization(limit="3", guards="callTarget == cachedCallTarget")
  fun doDirect(
    frame: VirtualFrame, callTarget: CallTarget, args: Array<Any>,
    @Cached("callTarget") cachedCallTarget: CallTarget,
    @Cached("create(cachedCallTarget)") callNode: DirectCallNode
  ) = callNode.call(args)

  @Specialization(replaces="doDirect")
  fun doIndirect(
    frame: VirtualFrame, callTarget: CallTarget, args: Array<Any>,
    @Cached("create()") callNode: IndirectCallNode
  ) = callNode.call(callTarget, args)
}
```

Listing 4.7: Polymorphic and megamorphic inline cache on a DispatchNode

Guards Listing 4.7 also demonstrates another optimization feature, a generalization of nodes specializing themselves based on types or arguments. A Specialization annotation can have arbitrary user-provided *guards*. These are often used in tandem with a cache, or with complex type specializations. In general, using a Specialization makes it possible to choose the most optimal node implementation based on its situation or configuration.

Profiles Another tool for optimization are *profiles*. These are objects that the developer can use to track whether a conditional statement was executed: in the implementation of an if statement, or when handling an exception. The compiler will use the information collected during optimization, e.g., when a ConditionProfile tracks that the condition in an if statement was true every time, the compiler will omit the else branch during compilation.

Assumptions *Assumptions* are the last tool that a developer can use to provide more information to the compiler. Unlike profiles and specializations that are local to a node, assumptions are global objects whose value can be changed from any part of a program graph. An assumption is *valid* when created, and it can be *invalidated*, which triggers deoptimization of any code that relies on it. A typical use of assumptions is shown in Listing 4.8 [46], where TruffleRuby relies on the fact that global variables are only seldom changed and can be cached. A ReadGlobalVarNode reads the value of the global variable only the first time, and relies on two assumptions afterwards, which are invalidated whenever the value of the variable changes, and the cached value is discarded.

```
@Specialization(assumptions = [
  "storage.getUnchangedAssumption()",
  "storage.getValidAssumption()"
])
fun readConstant(
  @Cached("getStorage()") storage: GlobalVariableStorage,
  @Cached("storage.getValue()") value: Any
) = value
```

Listing 4.8: Cached reading of a global variable using assumptions [46]

Inlining During optimization, the Graal compiler replaces `DirectCallNode`s with the contents of the call target they refer to, performing function *inlining* [54]. Often, this is the optimization with the most impact, as replacing a function call with the body of the callee means that many other optimizations can be applied. For example, if a `for` loop contains only a function call and the function is inlined, then the optimizer could further analyze the data flow, and potentially either reduce the loop to a constant, or to a vector instruction.

There are potential drawbacks, and Truffle documentation warns developers to place `TruffleBoundary` annotations on functions that would be expanded to large program graphs, like `printf`, as Graal will not ever inline a function through a `TruffleBoundary`.

Splitting Related to inlining, a call target can also be *split* into a number of *monomorphic* call targets. Previously, we saw an `AddNode` that could add either integers or strings. If this was a global or built-in function that was called from different places with different configurations of arguments, then this node could be split into two: one that only handles integers and one for strings. Only the monomorphic version would then be inlined at a call site, leading to even better possibility of optimizations.

Both of these two techniques, inlining and splitting, are guided by Graal heuristics, and they are generally one of the last optimization techniques to be checked when there are no more gains to be gained from caching or specializations.

Object model Truffle has a standard way of structuring data with fixed layout, called the Object Storage Model [22]. It is primarily intended for class instances that have a user-defined data layout, but e.g., the meta-interpreter project DynSem [51] uses it for variable scopes, and TruffleRuby uses it to make C `struct`s accessible from Ruby as if they were objects. Similar to `Frame`s, an empty `DynamicObject` is instantiated from a `Shape` (corresponds to a `FrameDescriptor`) that contains several instances of a `Property` (corresponds to a `FrameSlot`). Listing 4.9 shows the main method of a node that accesses an object property, also utilizing a polymorphic cache.

Interop As previously mentioned, it is possible to evaluate *foreign* code from other languages using functions like *eval*, referred to as *polyglot*. However, Truffle also makes it

```
@Specialization(guards=[
  "addr.key() == keyCached",
  "shapeCached.check(addr.frame())"
], limit="20")
fun doSetCached(
  addr: FrameAddr, value: Any,
  @Cached("addr.key()") keyCached: Occurrence,
  @Cached("addr.frame().getShape()") shapeCached: Shape,
  @Cached("shapeCached.getProperty(keyCached)") slotProperty: Property
): Unit {
  slotProperty.set(addr.frame(), value, shapeCached)
}
```

Listing 4.9: Accessing an object property using a `Shape` and a `Property` [51]

possible to use other languages' *values*: to define a foreign function and use it in the original language, to import a library from a different language and use it as if it was native. This is referred to as an interoperability message protocol or *interop*, for short.

Truffle uses *libraries* to accomplish this. They play a role similar to *interfaces* in object-oriented languages [22], and describe capabilities of `ValueTypes`. A library *message* is an operation that a value type can support, and it is implemented as a special node in the program graph, as a nested class inside the value type. The `ValueTypes` of a foreign language then need to be mapped based on these libraries into a language: a value that implements an `ArrayLibrary` can be accessed using array syntax, see Listing 4.10. Libraries are also used for polymorphic operations inside a language if there is a large amount of value types, to remove duplicate code that would otherwise be spread over multiple `Specializations`.

```
class ArrayReadNode : Node {
  @Specialization(guards="arrays.isArray(array)", limit="2")
  fun doDefault(
    array: Object, index: Int,
    @CachedLibrary("array") arrays: ArrayLibrary
  ): Int = arrays.read(array, index)
}
```

Listing 4.10: Array access using a `Library` interface[7]

4.5 Mapping concepts to Truffle

Where to use Truffle? Truffle uses JIT compilation, and optimizes repeatedly executed parts of a program. Many parts of the previously implemented interpreter are only one-off computations, though, e.g., the elaboration process itself that processes a pre-term once and produces a corresponding term, while discarding the pre-term. Only the evaluation of terms to values runs multiple times, as (top-level) functions are stored in the form of terms. It is possible that the elaboration process might benefit as well, by implementing *infer*, *check*, and *unify* as Truffle nodes and using those in place of functions.

[7]Source: `https://www.graalvm.org/graalvm-as-a-platform/language-implementation-framework/TruffleLibraries/`

Inspiration For inspiration, I have looked at a number of other functional languages that use Truffle: several theses (TruffleClojure [16], TrufflePascal [18], Mozart-Oz [27]), two Oracle projects (FastR [49], TruffleRuby [46]), and other projects (Cadenza [30], DynSem [51], Mumbler [15], Truffled PureScript [48]).

In the last phases of writing this thesis, I have encountered the project Enso [38] that was released April 13th, a month before my thesis deadline. It is a visual programming language that uses dependent types at its core. In particular, compared to my previous approach, it strictly delineates between elaboration, compilation, and evaluation, introducing a special *compiler* component. This allows them to apply optimizations in the spirit of a compiled language, gradually performing optimization *passes*, and refining the code that is transformed into a Truffle graph in the end.

After comparing their approach to mine, I have found theirs significantly more viable with regards to optimization: my original approach was to transform the entire elaboration algorithm into a Truffle graph containing nodes like *QuoteNode*, *InferNode*, etc. The entire system contained a large amount of components, and the program graph did not usually manage to stabilize enough that it would be compiled.

I have adopted this separation of elaboration and evaluation, meaning that only *closures* are compiled into machine code. This significantly simplifies both the implementation and any optimization efforts (profiling and reading of Truffle graphs), and this is the approach that I will describe in this section. This approach is very similar to the implementations of other proof assistants that also use an evaluation platform other than the host language for evaluation, e.g., Coq supports several backends (`native_compute`, `vm_compute`),

4.5.1 Approach

Out of the many changes that are required, the largest one is in the encoding of functions and closures, and replacing closure implementation with call targets. Environments and variable references need to be rewritten to use explicit stack frames, and lazy evaluation cannot use Kotlin's `lazy` abstraction, but instead needs to be encoded as an explicit `Thunk` object.

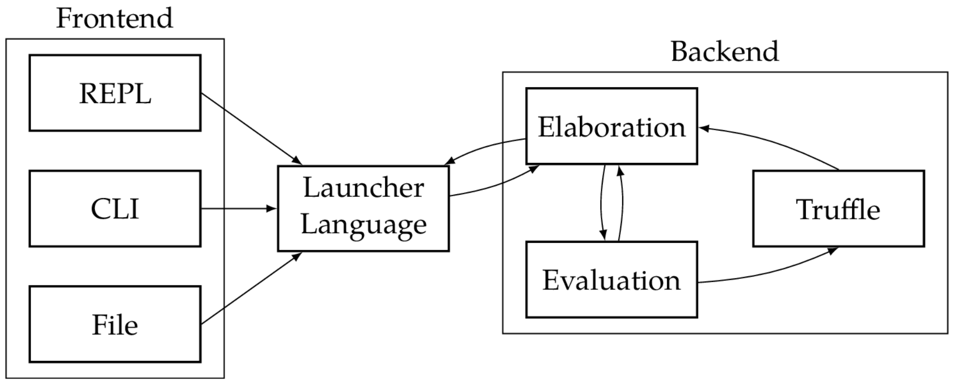


Figure 4.7: Component overview of the Truffle interpreter

Figure 4.7 demonstrates the components of the new interpreter. The `Launcher` is the same as in the previous interpreter, only now we use the `Context` that it prepares based on user-provided options. The `Language` object initializes a different `Context` object, a `MontunoContext`, which is an internal object containing the top-level variable scope, the meta-variable scope, and other global state variables. `Language` then dispatches parsing requests to the parser, and the pre-terms it produces are then wrapped into a `Program-RootNode`.

Executing the `ProgramRootNode` starts the elaboration process, calling into the code of the original interpreter. During evaluation of a term into a value, any closures produced are compiled into a Truffle graph, saving the current environment in the form of an array. The closure is then evaluated when supplied with an argument into a value, which can again be compared, unified, or built back up into a `Term` using `quote`.

Elaboration and evaluation both access the `MontunoContext` object to resolve top-level variables and meta-variables into the corresponding `Term`s. The REPL needs to obtain lists of bound variables to produce suggestions and process other REPL commands from the context as well, but all interaction between the host code of the `Launcher` and the internals of the language need to happen via the polyglot interface. This means that any REPL commands that need to access the language context now need to be implemented as language pragmas: e.g., to reset the language state, we now need the pragma `{-# RESET #-}`, which is then also accessible in user-provided code as well.

The data flow in Figure 4.8, if compared with the previous data flow diagram, only adds the data representation *Code*, and the operation *Close*, which represents the construction of a closure that *closes* over the current environment. Otherwise, other parts of the system can stay the same, at least on first glance.

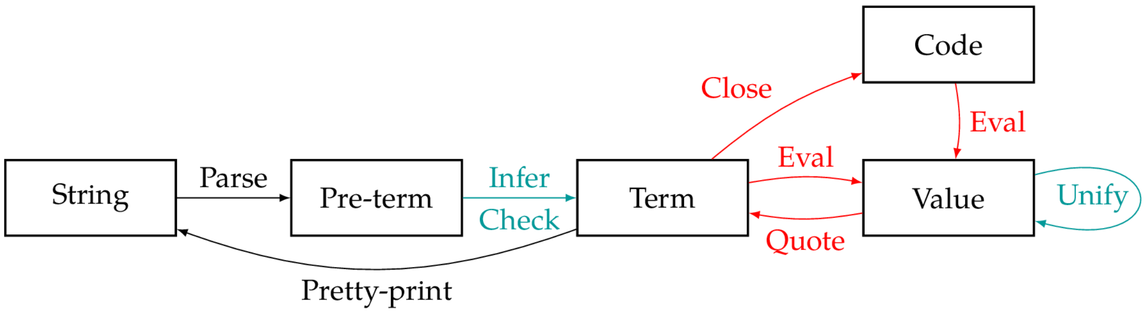


Figure 4.8: Data flow overview. Cyan is elaboration, red normalization-by-evaluation

4.5.2 Implementation notes

While I have sketched the changes that would be required when creating a new Truffle interpreter based on the previous, non-Truffle one, the actual implementation process was slightly different. Instead of creating a entirely separate interpreter that would share some library code, as with my first attempts at creating a Truffle interpreter for Montuno, the effort to incorporate the improvements of Enso resulted in the two codebases merging.

Unlike my previous implementations, the non-Truffle and Truffle versions share the entire elaboration component. The only difference is in a single pluggable component, a `Com-piler`. This is an interface with a single method, `buildClosure`, that produces the original VClosure value in the non-Truffle version, and in the Truffle version it transforms the body of a closure into a Truffle graph.

4.5.3 Values

```
@TypeSystem(
  VUnit::class, VThunk::class,
  VLam::class, VPi::class,
  VPair::class, VSg::class,
  VMeta::class, VLocal::class, VTop::class,
  VNat::class, VBool::class,
)
class Types {
  @TypeCheck(VUnit::class)
  fun isVUnit(value: Any) = value === VUnit
}

@ValueType
object VUnit : TruffleObject
@ValueType
class VPair(val left: Any, val right: Any) : TruffleObject
```

Listing 4.11: A TypeSystem and two simple ValueTypes from the Truffle interpreter

Not including constants, we have only two main value types: a Π-type (equivalent to a λ-abstraction), and a Σ-type. A Π-type maps onto a closure and will be discussed momentarily. A Σ-type can be expressed as a pair, or a linked list of nested pairs, to use the simplest representation. Then there are neutral terms, unresolved variables that accumulate a spine of unapplied operands and projections: these are expressed as a head containing a variable reference, and a spine with an array of spine values.

Each of these values needs to be a separate ValueType class, and an entry in the Truffle type system. A snippet in Listing 4.11 shows the TypeSystem and two simple value types. Other than the above-mentioned types of values, there is a number of literal types, and a type VThunk. This type needs to be explicitly mentioned here, so that we can implement lazy evaluation in Truffle; its interface is exactly the same as a Kotlin lazy, but a VThunk inherits from the class TruffleObject so that it can be used inside compiled code.

4.5.4 Closures

A closure needs to store the function to execute, which was a Term in the non-Truffle implementation, together with the execution environment. The Truffle version replaces the Term with a CallTarget that points to a Truffle graph. A CallTarget can only be called with an array of objects, so this is what a Closure stores in the place of an environment. The CallTarget points to a ClosureRootNode, which first copies the array of arguments it was given into the local scope, and then executes the body. Reports from the Cadenza project [30] show that this is more efficient than simply accessing the environment as an array, as the virtual frame can be optimized by Truffle. The closure object itself can be seen in Listing 4.12. Notably, it also uses the InteropLibrary mentioned in the previous chapter, meaning that it can also be invoked from outside of the interpreter internals.

```
@CompilerDirectives.ValueType
@ExportLibrary(InteropLibrary::class)
data class TruffleClosure(
    val env: VEnv, val callTarget: CallTarget
) : TruffleObject {
  override val arity: Int = 1
  @ExportMessage fun isExecutable() = true
  @ExportMessage override fun execute(vararg args: Any?): Val {
    return callTarget.call(*concat(env, args))
  }
}
```

Listing 4.12: Sketch of the closure implementation

Changing the definition of a closure also means that all `VPi` or `VLam` values now contain a closure that was processed by the previously mentioned `Compiler` component. My original intent was to compile only top-level definitions (globally defined functions or constants) and meta-variables, but the processes of meta-variable solving and elaborating a top-level definition both produce a value, so it is simpler to produce closures globally, across all the entire interpreter.

A `Term` is processed by the `Compiler` into a `Code` object, which is then wrapped into the `ClosureRootNode`, and converted into a `CallTarget`. The compilation process produces the Truffle graph as another tree structure, only this one will be interpreted by Truffle and not our algorithms. A snippet of the compilation code can be seen in Listing 4.13. It converts the `Term` type into a version of the `eval` function: a Π-type produces a closure, therefore we need to start a new compilation process with its body. a local variable reads from the `Frame`, therefore we find the `FrameSlot` that corresponds to the de Bruijn index, and compile it into a *read* operation. Finally, a unit term is compiled into a constant expression.

```
class TruffleCompiler(val ctx: MontunoContext) : Compiler() {
  override fun buildClosure(t: Term, env: VEnv): Closure {
    val fd = FrameDescriptor()
    val code = compileTerm(bodyTerm, Lvl(env.size + 1))
    val root = TruffleRootNode(code, ctx.top.lang, fd)
    return TruffleClosure(ctx, env, root.callTarget)
  }
  private fun compileTerm(
    t: Term, depth: Lvl, fd: FrameDescriptor
  ): Code = when (t) {
    is TPi -> CClosure(t.name, t.type,
                       compileTerm(t.bound, depth, fd),
                       compileClosure(body, depth + 1))
    is TApp -> CApp(compileTerm(t.l, depth, fd),
                    compileTerm(t.r, depth, fd))
    is TLocal -> CReadLocal(fd.findFrameSlot(t.ix.toLvl(depth).it))
    is TMeta -> CDerefMeta(t.slot)
    TUnit -> CConstant(VUnit)
  } //...
}
```

Listing 4.13: Snippet from the `Compiler`

4.5.5 Elaboration

Given the previously mentioned approach, changing the elaboration would not be necessary at all. However, in the course of building the second version, I had attempted to perform type-directed optimization, meaning that to optimize a term well, I needed its type. In the course of performing this change, I had changed the meta-context from untyped to typed, meaning that each meta-variable now has a type that the solution needs to conform to. This means that all binders in terms and values can now store the type of their argument, which allows the optimization process to use this information.

The actual implementation of meta-variables, as used by the compiler is shown in Listing 4.14. The Truffle graph node stores a reference to the meta-variable. If it has been solved between the time of the compilation and execution, this node is replaced with the value of the solution. If it has not, then this node produces a neutral value with an empty spine.

```
open class CDerefMeta(val slot: MetaEntry) : Code() {
  override fun execute(frame: VirtualFrame): Val = when {
    !slot.solved -> VMeta(slot.meta, VSpine(), slot)
    else -> {
      replace(CConstant(slot.value!!, loc))
      slot.value!!
    }
  }
}
```

Listing 4.14: Meta-variable reference node in Truffle

4.5.6 Normalization

Given the approach taken, there are no actual changes to the normalization algorithm, only the references to closures in Π- and λ-terms have been changed to be an opaque interface, which offers only a single operation `instantiate`. The non-Truffle implementation needed to slightly change as well to conform to this interface. The interface and the non-Truffle implementation of a closure are demonstrated in Listing 4.15.

```
interface Closure {
  fun inst(v: Val): Val
}
@CompilerDirectives.ValueType
@ExportLibrary(InteropLibrary::class)
data class PureClosure(val env: VEnv, val body: Term) : Closure {
  override val arity: Int = 1
  @ExportMessage fun isExecutable() = true
  @ExportMessage override fun execute(vararg args: Any?): Val
    = body.eval(concat(env, args))
}
```

Listing 4.15: Non-Truffle closure implementation

4.5.7 Built-ins

Built-in constants and types need to be implemented as special nodes. The resolution of a built-in name to its corresponding node happens during compilation. Each built-in term has its arity, the number of expected arguments. The compiler produces the correct number of closures (`VLam`) nodes around the built-in node invocation that corresponds to the arity of the operation. These are passed to a `BuiltinRootNode` that uses them directly, unlike the `ClosureRootNode`, that first copies them to the local scope

This is shown on the example of a `Succ` node in Listing 4.16. This node has the arity 1, it expects a single argument, which can be either an already evaluated integer or a `Thunk` that will produce an integer, which is then *forced*, and coerced to a integer using a function generated by the `TypeSystem`.

```
class Succ : BuiltinTerm(1) {
  @Specialization
  fun doInt(n: Int) = n + 1
  @Specialization
  fun doThunk(t: Thunk) = TypesGen.asVNat(t.force()) + 1
}
```

Listing 4.16: Example of a built-in node, a `Succ` node

4.5.8 Driver

The previously-mentioned decision to share elaboration code was also partly motivated by the experience of working on the original Truffle code base: while it is possible to pause the execution of a Truffle program, and inspect it inside a debugger, it is impossible to access the language context from outside the language code. This made any attempts at incremental development using unit tests impossible, as I was only able to test the complete process by providing input from the outside.

While this also means that the polyglot interface of the language is well-tested, it discouraged me from trying to adapt the existing Truffle codebase when attempting to integrate the concepts from Enso. However, this limitation places additional demands on the driver code that handles user interaction. Each user command that is impossible to express using the polyglot interface needs to be implemented as an interpreter directive, a pragma. As processing of parsed input happens in the course of elaboration, this means that the function `checkTopLevel` not only handles elaboration of top-level definitions, but also these commands. They are parsed into `RTerm` nodes, as the same mechanism is used for pragmas like `NORMALIZE`, that expect a well-formed expression. An snippet of the code that processes these commands is shown in Listing 4.17.

One other benefit of the two interpreters sharing their code base is that switching between backends is possible by simply issuing a command in the REPL interface, as demonstrated in Listing 4.18. Extending the compiler backend with different or better optimizations passes, in the manner of Montuno, would be possible by simply extending the `Compiler` interface. Any algorithmic improvements to the elaboration process would then be shared between all implementations.

```
fun checkTopLevel(top: MontunoContext, e: TopLevel): Any? {
  top.metas.newMetaBlock()
  val ctx = LocalContext(top, LocalEnv(top.ntbl))
  return when (e) {
    // ...
    is RTerm -> when (e.cmd) {
      Pragma.NOTHING -> top.pretty(ctx.infer(e.tm))
      Pragma.RESET -> { top.reset(); null }
      Pragma.SYMBOLS -> top.getSymbols()
      Pragma.BUILTIN -> { top.registerBuiltins(e.loc, e.tm); null }
      Pragma.PRINT -> top.printElaborated()
    }
  }
}
```

Listing 4.17: Processing REPL commands inside elaboration

```
$> montuno --truffle
Mt> :engine
montuno-truffle
Mt> id : {A} -> A -> A = \x.x
Mt> :normalize id id
λ x. x
Mt> :engine montuno-pure
Mt> id : {A} -> A -> A = \x.x
Mt> :normalize id id
λ x. x
```

Listing 4.18: Switching compilation engines in REPL

Chapter 5

Evaluation

We now have two interpreters, one written in pure Kotlin that uses the general JIT compilation provided by JVM, the other delegating term evaluation to Truffle. It is now time to evaluate their performance on a number of both elaboration and evaluation tasks. We will also briefly compare their performance with the performance of state-of-the-art proof assistants on the same tasks.

The primary goal is to evaluate the general time and memory characteristics of the systems and how they vary with the size of the term and on the number of binders (λ- and Π-abstractions). For this purpose, we will construct a set of benchmarks involving simple expressions, whose size we can easily vary.

A secondary goal is to investigate the costs associated with a runtime system based on the JVM, and how they may be eliminated. We will also prepare a suite of benchmarks by translating a number of test cases from common performance test suites, and compare this system's performance with other functional languages.

5.1 Workload

The evaluation workload will be split into two parts: elaboration tasks, that test the combined performance of our infer/check bidirectional typing algorithm, normalization-by-evaluation, and unification; and normalization tasks, that only test the performance of normalization-by-evaluation. These benchmark tasks were partially adapted from the work of András Kovács [32], and partially extrapolated from the evaluation part of Jason Gross's doctoral thesis [24].

5.1.1 Normalization

Both normalization and elaboration tasks need to involve terms that can be easily made larger or smaller. A typical workload involves Church-encoded terms, naturals in particular, as these involve λ-abstraction and application. They will be tested in the first set of tasks: evaluation of a large natural number to a unit value. These will be first evaluated on Church-encoded naturals, and then on the built-in type *Nat* that is backed by a Java integer.

```
Nat : Unit = (N : Unit) → (N → N) → N → N
zero : Nat = λ N s z. z
succ : Nat → Nat = λ a N s z. s (a N s z)
mul : Nat → Nat → Nat = λ a b N s z. a N (b N s) z
forceNat : Nat → Unit = λn. n _ (λx.x) Unit
```

(a) Church-encoded naturals and `forceNat`

```
n10    = mul n2 n5
n20    = mul n2 n10
-- ...
n20M   = mul n2 n10M
```

(b) Large Church numbers

Figure 5.1: Benchmark tasks: large Church naturals

5.1.2 Elaboration

Elaboration will test not only the NbE part of our system, but also type inference and checking. We will use not only deeply nested function application of Church-encoded terms, but also terms with large types: those that contain many or deeply nested function arrows or Π-types. The first task is the elaboration of a polymorphic *id* function repeatedly applied to itself, as this produces meta-variables whose solution doubles in size with each application. The second task especially tests unification: the task is to unify two large Church-encoded natural numbers, to check them for equivalence.

```
idTy : {A} → A → A
id : idTy = λ x. x
test : idTy = id id id [...] id
```

(a) Church-encoded naturals and `forceNat`

```
Eq : Nat -> Nat -> Unit
   = λ x y. (P : Nat → Unit) → P x → P y
x : Eq n20Mb n20M = \_ x.x
```

(b) Large Church numbers

Figure 5.2: Benchmark tasks: elaboration

5.2 Methodology

There are many ways how we can measure each language's performance on these tasks. The main concern is that Montuno and MontunoTruffle are JIT-compiled languages that need a significant amount of warm-up: the first iterations will take significantly longer than the iterations that happen after warm-up, after all code is JIT-optimized.

There are several options for eliminating the influence of JIT warm-up on performance measurements: the first is to measure an "empty run", and simply subtract the times from benchmarking runs. This eliminates JIT start-up times, but does not eliminate the time required for warming-up of the user system itself. For this reason, there are various tools for either statistical analysis of the results that discard all but the stable-state performance, or that simply measure a large number of iterations.

This, however, measures the performance of the system on a single computational task. The user-visible delay between starting the program and seeing results is what interests the users of a program, especially in an interpreter for a programming language, that may need to run often, for a tight feedback loop during program development.

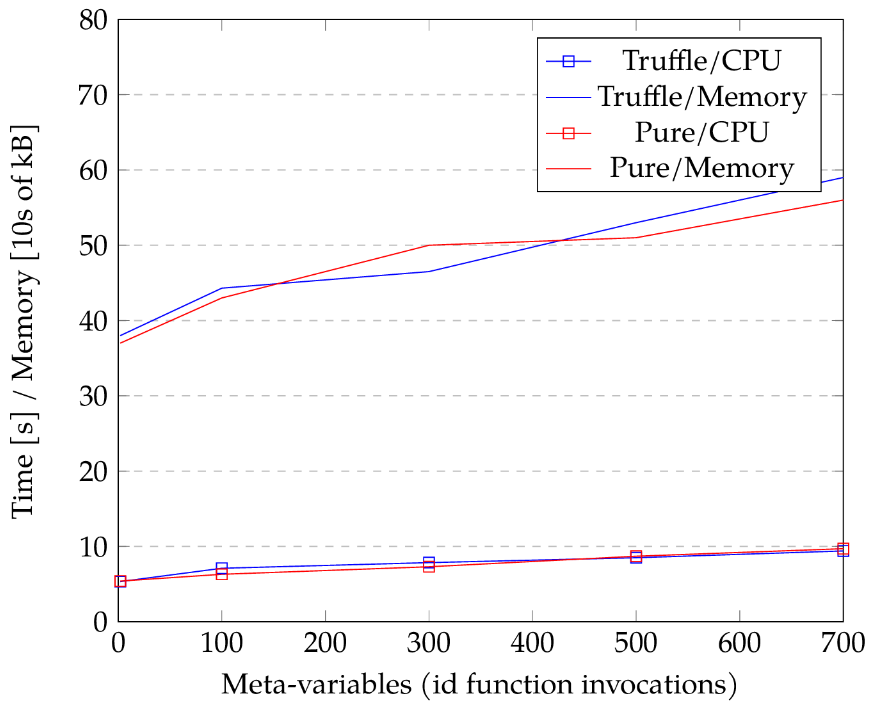


Figure 5.3: Results of evaluation on the *id* stress test

For this reason, I have selected the option of measuring whole-system performance including start-up and warm-up times, but with a large number of iterations of the benchmark tasks, so their influence on the overall runtime is also visible. Aside from measuring the time it takes to normalize or elaborate an expression, we will also measure the peak memory usage using the system tool `time -v`.

These benchmarks were run on my personal computer, a laptop with 32GBs of RAM and a 16 core processor running at 1.4GHz (AMD Ryzen 7 PRO 4750U). No other programs were running on the machine at the time of benchmarking, to eliminate external influences.

5.3 Results

The measurements were performed using ten iterations of the program (accomplished by adding a loop to the program code), as an average of five measurements.

Figure 5.3 shows the results of evaluating the system on the *idStress* benchmark, measuring the performance of solving meta-variables. The results are disappointing. Unfortunately, I have only started stress-testing the system in the last parts of the implementation, otherwise the project might have taken a different direction.

The original benchmarks in the SmallTT project [32] included comparison of up to one million binders and not only 700, but the reason for not including more than 700 is that the elaboration overflowed the stack, using too deep recursive function calls. Overall, this shows how recursive descent tree transformation algorithms are unsuitable for the Java Virtual Machine, and does not say too much about the performance of the system overall.

Program warm-up (a trivial run that only checks the base *id* definition) takes over five seconds, going up to ten seconds for the largest run. The long warm-up is to be expected from a JVM-based system. However, even compared to the two years old benchmarks of systems like Agda or Coq from the SmallTT project, the performance of these systems are extremely disappointing.

Rewriting recursive descent algorithms into stack based ones is not a small undertaking, so instead of trying to push the deadline even further, I have searched for possible causes. I have found another project of Kovács', a recent one that compares the suitability of different platforms for elaboration [31], and which has added JVM-based systems to its comparison two months ago. In that evaluation, JVM-based platforms have just as disappointing performance as my solution.

Instead of the many planned measurements and evaluations, I have instead attempted to analyze possible performance bottlenecks: largely, they come down to the capability of the JVM to handle recursive functions, which is a long-standing issue with the JVM capabilities. Trivial optimizations did not help, and non-trivial optimizations were out of my time range.

5.4 Discussion

Instead of general programming platforms, languages with dependent types usually use the platforms of functional languages, e.g., GHC Haskell and its Spineless Tagless G-Machine (STG), OCaml and the Zinc abstract machine, or Idris 2 that uses Chez Scheme.

These platforms are optimized for fast function calls, currying, tail-calls, and non-eager (lazy) evaluation strategy, all of which needs to be emulated manually on the JVM. I will follow with a brief list of optimizations that I have attempted to apply on the Montuno interpreters:

- Replacing functions for tree transformations with object methods on the term and value nodes themselves. This halved the stack usage.
- Applying λ-merging to closures, introducing the concept of the arity of a closure: the body of a closure is not called until all arguments it expects are supplied, which applies mostly to nested `VLam` or `VPi` terms. This helped in some test cases, but not in the general case.
- Removing lazy evaluation. Surprisingly, this improved the performance of the system the most, as the overhead caused by wrapping and unwrapping values in closures was removed. This is an optimization that helps in general system performance, but its effects are often unpredictable.
- Unfolding recursive helper functions for processing linked lists of data into iterative ones, using `for` cycles. This helped slightly with stack usage, but not as much as I'd expected.

After looking for more optimization opportunities, I have found an analysis in the Enso [38] system, where they have encountered the same problem, and used an interesting workaround. Each thread in the JVM has a separate stack space limit, so instead of working around the stack limit, they introduce a heuristic that tracks stack usage [1], and spawns a new thread with the currently running computation.

Although in my research, I have primarily focused on the optimization opportunities of Truffle and GraalVM, I have disregarded the optimizations required for the system itself that is written Kotlin, and often directly used the same style that I am used to from working in Haskell and similar languages.

5.5 Next work

From my research, it seems that Truffle can bring interesting benefits to programming language implementers, andn perhaps even for implementing the runtime systems of dependent types, but not for the style of algorithms that are used in dependent type elaboration without significantly more effort put into their optimization and adaptation for the JVM platform.

I have started this thesis as a follow up to the Cadenza project [30] that asked whether Truffle can be useful to a simply-typed λ-calculus implementation. Its results were slightly disappointing, but the project's author suggested a follow-up project to me as a thesis topic likely to produce a positive result. I will have to conclude from the performance of this project that *cannot* benefit a runtime system for dependent types, at least not using the approach that I have taken.

When finishing my thesis, I have discovered that parallel to my work on this thesis, the author has started a new project called TruffleSTG[2], that attempts to apply the knowledge from these two projects to another domain, that is even more likely to benefit from Truffle. The STG is the abstract machine used by Haskell and other languages that compile to Haskell. Instead of implementing the entire elaboration system in Kotlin or another JVM-based language, TruffleSTG uses the GHC Haskell machinery to process dependent types, and only uses Truffle as a compiler. This takes the approach from my thesis to the extreme. Where I have used Truffle as a backend that directly communicates with the elaboration process, TruffleSTG communicates with GHC using GHC-WPC interface files, meaning that it only attempts to act as a runtime system for Haskell and nothing more.

The language implemented in the course of this thesis has its limitations, the user interface is incomplete, and the system likely has some bugs that would be discovered when used on a larger scale, all of which would be good to fix and publish this project as inspiration for future endeavors in this area.

However, I believe a very useful follow-up to this project would be a rigorous evaluation of the different platforms that can be used for both functional languages, and for language with dependent types, extending the benchmarks by Kovacs [31], and standardizing them as a set of common elaboration tasks, which can then be used to compare the performance of proof assistants and systems with dependent types.

[1]Source: `https://enso.org/docs/developer/enso/runtime/unbounded-recursion.html`
[2]TruffleSTG, `https://github.com/acertain/trufflestg`

Chapter 6

Conclusion

The first part of this thesis presents the concepts necessary for understanding and specifying type systems based on the systems of the λ-cube. I have used these concepts to specify a small, dependently-typed language. The second part contains an overview of state-of-the-art algorithms involved in creating an interpreter for a dependently-typed language, and presents my implementation of such an interpreter in Kotlin.

In the third part of the thesis, I have presented GraalVM, the Truffle language implementation framework, and the optimization possibilities they provide. I have implemented a second interpreter for the language using the Truffle framework, also written in Kotlin. The fourth part contains the compilation of a small set of benchmarks for investigating the elaboration performance of dependently-typed languages, and uses them to evaluate the effect of JIT compilation on the performance of the interpreters, and to coarsely compare their performance with the performance of state-of-the-art languages.

The results are, however, unsatisfactory. The benefits brought by JIT compilation are outweighed by the overhead of the implementation on the JVM platform. Its performance is lacking, compared to platforms like GHC Haskell, or Chez Scheme that are used in other dependently-typed languages. I believe further investigation would manage to eliminate most causes of inefficiency, but compared to the initial expectations, such conclusions disprove the entire premise of my thesis.

Overall, despite the negative conclusion, I believe this thesis has fulfilled a large part of its goals. It presents a concise introduction to the concepts required for implementing a dependently-typed language, and an overview of the optimization opportunities offered by Truffle, which can both form the starting point for other projects in this area.

Bibliography

[1] Abel, A. and Pientka, B. Higher-order dynamic pattern unification for dependent types and records. In: Springer. *International Conference on Typed Lambda Calculi and Applications*. 2011, p. 10–26.

[2] Abel, A., Vezzosi, A. and Winterhalter, T. Normalization by evaluation for sized dependent types. *ICFP '17*. ACM. 2017, vol. 1, p. 1–30.

[3] Altenkirch, T. and Kaposi, A. Normalisation by evaluation for dependent types. In: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. 2016.

[4] Ariola, Z. M. and Felleisen, M. The call-by-need lambda calculus. *Journal of functional programming*. Cambridge University Press. 1997, vol. 7, no. 3, p. 265–301.

[5] Baez, J. and Stay, M. Physics, topology, logic and computation: a Rosetta Stone. In: *New structures for physics*. Springer, 2010, p. 95–172.

[6] Barendregt, H. P. Lambda calculi with types. Oxford: Clarendon Press. 1992.

[7] Bolz, C. F. *Meta-tracing just-in-time compilation for RPython*. 2014. Dissertation. Universitäts-und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf.

[8] Bove, A. and Dybjer, P. Dependent types at work. In: Springer. *International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development*. 2008, p. 57–99.

[9] Brady, E. Idris 2: Quantitative Type Theory in Practice. *ArXiv preprint arXiv:2104.00480*. 2021.

[10] Christiansen, D. T. *Checking Dependent Types with Normalization by Evaluation: A Tutorial (Haskell Version)*. 2019.

[11] Duboscq, G., Stadler, L., Würthinger, T., Simon, D., Wimmer, C. et al. Graal IR: An extensible declarative intermediate representation. In: *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*. 2013.

[12] Dunfield, J. and Krishnaswami, N. Bidirectional typing. *ArXiv preprint arXiv:1908.05839*. 2019.

[13] Ebner, G., Ullrich, S., Roesch, J., Avigad, J. and Moura, L. de. A metaprogramming framework for formal verification. *ICFP '17*. ACM. 2017, vol. 1, p. 1–29.

[14] Eisenberg, R. A. Stitch: the sound type-indexed type checker (functional pearl). In: *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell*. 2020, p. 39–53.

[15] Esquivias, C. *cesquivias/mumbler - GitHub*. [Online; accessed May 09, 2021]. Available at: `https://github.com/cesquivias/mumbler/`.

[16] Feichtinger, T. *TruffleClojure: A self-optimizing AST-Interpreter for Clojure/submitted by: Thomas Feichtinger*. 2015. Master's thesis. Johannes Kepler Universität, Linz.

[17] Ferreira, F. and Pientka, B. Bidirectional elaboration of dependently typed programs. In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. 2014, p. 161–174.

[18] Flimmel, J. *Pascal with Truffle*. 2017. Master's thesis. Univerzita Karlova, Matematicko-fyzikální fakulta.

[19] Fumero, J., Steuwer, M., Stadler, L. and Dubach, C. Just-in-time gpu compilation for interpreted languages with partial evaluation. In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2017, p. 60–73.

[20] Gonthier, G. Formal proof–the four-color theorem. *Notices of the AMS*. 2008, vol. 55, no. 11, p. 1382–1393.

[21] Gratzer, D., Sterling, J. and Birkedal, L. Implementing a modal dependent type theory. *ICFP '19*. ACM. 2019, vol. 3, p. 1–29.

[22] Grimmer, M., Seaton, C., Schatz, R., Würthinger, T. and Mössenböck, H. High-performance cross-language interoperability in a multi-language runtime. In: *Proceedings of the 11th Symposium on Dynamic Languages*. 2015, p. 78–90.

[23] Gross, J., Chlipala, A. and Spivak, D. I. Experience implementing a performant category-theory library in Coq. In: Springer. *International Conference on Interactive Theorem Proving*. 2014, p. 275–291.

[24] Gross, J. S. *Performance Engineering of Proof-Based Software Systems at Scale*. 2021. Dissertation. Massachusetts Institute of Technology.

[25] Guallart, N. An overview of type theories. *Axiomathes*. Springer. 2015, vol. 25, no. 1, p. 61–77.

[26] Gundry, A. and McBride, C. A tutorial implementation of dynamic pattern unification. *Unpublished draft*. 2013.

[27] Istasse, M. *An Oz implementation using Truffle and Graal*. 2017. Master's thesis. Universidad de Granada.

[28] Kamareddine, F. Reviewing the Classical and the de Bruijn Notation for λ-calculus and Pure Type Systems. *Journal of Logic and Computation*. Oxford University Press. 2001, vol. 11, no. 3, p. 363–394.

[29] Kleeblatt, D. *On a Strongly Normalizing STG Machine with an Application to Dependent Type Checking*. 2011. Master's thesis. Technischen Universitat Berlin.

[30] Kmett, E. *ekmett/cadenza - GitHub*. [Online; accessed May 09, 2021]. Available at: `https://github.com/ekmett/cadenza/`.

[31] Kovács, A. *AndrasKovacs/normalization-bench - GitHub*. [Online; accessed Apr 24, 2021]. Available at: `https://github.com/AndrasKovacs/normalization-bench/`.

[32] Kovács, A. *AndrasKovacs/smalltt - GitHub*. [Online; accessed May 09, 2021]. Available at: `https://github.com/AndrasKovacs/smalltt/`.

[33] Kovács, A. Elaboration with First-Class Implicit Function Types. *ICFP '20*. ACM. vol. 4.

[34] Latifi, F. Practical Second Futamura Projection: Partial Evaluation for High-Performance Language Interpreters. In: *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 2019, p. 29–31.

[35] Lescanne, P. and Rouyer Degli, J. Explicit substitutions with de Bruijn's levels. In: Springer. *International Conference on Rewriting Techniques and Applications*. 1995, p. 294–308.

[36] Mazzoli, F. and Abel, A. Type checking through unification. *ArXiv preprint arXiv:1609.09709*. 2016.

[37] Nawaz, M. S., Malik, M., Li, Y., Sun, M. and Lali, M. I. U. A Survey on Theorem Provers in Formal Methods. *CoRR*. 2019, abs/1912.03028.

[38] New Byte Order Inc.. *enso-org/enso - GitHub*. [Online; accessed Apr 24, 2021]. Available at: `https://github.com/enso-org/enso/`.

[39] Norell, U. *Towards a practical programming language based on dependent type theory*. 2007. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology.

[40] Norell, U. Dependently typed programming in Agda. In: Springer. *International school on advanced functional programming*. 2008, p. 230–266.

[41] Norell, U. and Coquand, C. Type checking in the presence of meta-variables. *Submitted to Typed Lambda Calculi and Applications*. Citeseer. 2007.

[42] Pagano, M. M. *Type-Checking and Normalisation By Evaluation for Dependent Type Systems*. 2012. Dissertation. Universidad Nacional de Córdoba.

[43] Pierce, B. C. and Benjamin, C. *Types and programming languages*. MIT Press, 2002.

[44] Schilling, T. *Trace-based just-in-time compilation for lazy functional programming languages*. 2013. Dissertation. University of Kent.

[45] Sestoft, P. Demonstrating lambda calculus reduction. In: *The essence of computation*. Springer, 2002, p. 420–435.

[46] Shopify. *Optimizing Ruby Lazy Initialization in TruffleRuby with Deoptimization*. [Online; accessed Mar 15, 2021]. Available at: `https://shopify.engineering/optimizing-ruby-lazy-initialization-in-truffleruby-with-deoptimization`.

[47] Šipek, M., Mihaljević, B. and Radovan, A. Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM. In: IEEE. *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2019, p. 1671–1676.

[48] SlamData Inc.. *slamdata/truffled-purescript - GitHub*. [Online; accessed Apr 24, 2021]. Available at: `https://github.com/slamdata/truffled-purescript/`.

[49] Stadler, L., Welc, A., Humer, C. and Jordan, M. Optimizing R language execution via aggressive speculation. *DLS '16*. ACM. 2016, vol. 52, no. 2, p. 84–95.

[50] Stolpe, D., Felgentreff, T., Humer, C., Niephaus, F. and Hirschfeld, R. Language-Independent Development Environment Support for Dynamic Runtimes. In:. ACM, 2019, p. 80–90.

[51] Vergu, V., Tolmach, A. and Visser, E. Scopes and frames improve meta-interpreter specialization. In: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. 2019.

[52] Wimmer, C. *Graal: High Performance Compilation for Managed Languages*. [Online; accessed Mar 15, 2021, presented at PLDI '17]. Available at: `http://lafo.ssw.uni-linz.ac.at/papers/2017_PLDI_GraalTutorial.pdf`.

[53] Wimmer, C., Jovanovic, V., Eckstein, E. and Würthinger, T. One Compiler: Deoptimization to Optimized Code. In: *Proceedings of the 26th International Conference on Compiler Construction*. Association for Computing Machinery, 2017, p. 55–64. CC 2017.

[54] Würthinger, T., Wimmer, C., Humer, C., Wöss, A., Stadler, L. et al. Practical partial evaluation for high-performance dynamic language runtimes. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, p. 662–676.

[55] Würthinger, T., Wimmer, C., Wöss, A., Stadler, L., Duboscq, G. et al. One VM to rule them all. In: *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. 2013, p. 187–204.

Appendices

Appendix A

Language specification

A.1 Syntax

$$
\begin{array}{rcllllll}
\textit{term} & := & v & \mid & \textit{constant} & & \\
 & \mid & a\ b & \mid & a\ \{b\} & & \\
 & \mid & a \to b & \mid & (a : A) \to b & \mid & \{a : A\} \to b \\
 & \mid & a \times b & \mid & (l : A) \times b & \mid & a.l \\
 & \mid & \text{let } x = v \text{ in } e & & & & \\
 & \mid & \_ & & & & \\
\textit{value} & := & \textit{constant} & \mid & \textit{neutral} & & \\
 & \mid & \lambda x : A.b & \mid & \Pi x : A.b & & \\
 & \mid & (a_1, \cdots, a_n) & & & & \\
 & \mid & \_ & & & & \\
\textit{neutral} & := & \textit{var} & \mid & \textit{neutral}\ a_1\ ...a_n & \mid & \textit{neutral}.l_n
\end{array}
$$

Figure A.1: Terms and values in Montuno (revisited)

A.2 Grammar

```
grammar Montuno;
@header { package montuno; }
file : END* decls+=top? (END+ decls+=top)* END* EOF ;
top : id=IDENT ':' type=term                     #Decl
    | id=binder (':' type=term)? '=' defn=term #Defn
    | '{-#' cmd=IDENT (target=term)? '#-}'   #Pragma
    | term                                     #Expr
    ;
term : lambda (',' tuple+=term)* ;
lambda
    : LAMBDA (rands+=lamBind)+ '.' body=lambda #Lam
    | 'let' IDENT ':' type=term '=' defn=term 'in' body=lambda #LetType
    | 'let' IDENT '=' defn=term 'in' body=lambda #Let
    | (spine+=piBinder)+ ARROW body=lambda      #Pi
    | sigma ARROW body=lambda                   #Fun
    | sigma                                    #LamTerm
    ;
sigma
    : '(' binder ':' type=term ')' TIMES body=term #SgNamed
    | type=app TIMES body=term            #SgAnon
    | app                                 #SigmaTerm
    ;
app : proj (args+=arg)* ;
proj: proj '.' IDENT #ProjNamed
    | proj '.1'      #ProjFst
    | proj '.2'      #ProjSnd
    | atom           #ProjTerm
    ;
arg : '{' (IDENT '=')? term '}' #ArgImpl
    | proj                      #ArgExpl
    ;
piBinder
    : '(' (ids+=binder)+ ':' type=term ')'    #PiExpl
    | '{' (ids+=binder)+ (':' type=term)? '}' #PiImpl
    ;
lamBind
    : binder                    #LamExpl
    | '{' binder '}'            #LamImpl
    | '{' IDENT '=' binder '}' #LamName
    ;
atom: '(' term ')'              #Rec
    | IDENT                     #Var
    | '_'                       #Hole
    | ('()' | 'Unit' | 'Type') #Star
    | NAT                       #Nat
    | '[' IDENT '|' FOREIGN? '|' term ']' #Foreign
    ;
binder : IDENT #Bind | '_' #Irrel ;
IDENT : [a-zA-Z] [a-zA-Z0-9'_]*;
NAT : [0-9]+;
LAMBDA : '\\' | 'λ';
ARROW : '->' | '→';
TIMES : '×' | '*';
```

A.3 Built-in constructs

- Unit : Type
- unit : Unit
- Nat : Type
- zero : Nat
- succ : Nat → Nat
- natElim : {A} → Nat → A → (Nat → A) → A
- Bool : Type
- true : Bool
- false : Bool
- cond : {A} → Bool → A → A → A
- fix : {A} → (A→A) → A
- the : (A) → A → A

A.4 Pre-terms

```
package montuno.syntax

typealias Pre = PreTerm
sealed class TopLevel : WithLoc
sealed class PreTerm : WithLoc

data class RDecl(val n: String, val ty: Pre) : TopLevel()
data class RDefn(val n: String, val ty: Pre?, val tm: Pre) : TopLevel()
data class RTerm(val cmd: Pragma, val tm: Pre?) : TopLevel()

data class RVar (val n: String) : Pre()
data class RPair(val lhs: Pre, val rhs: Pre) : Pre()
data class RApp (
    val arg: ArgInfo, val rator: Pre, val rand: Pre
) : Pre()
data class RLam (
    val arg: ArgInfo, val bind: Binding, val body: Pre
) : Pre()
data class RLet (
    val n: String, val type: Pre?, val defn: Pre, val body: Pre
) : Pre()
data class RPi  (
    val bind: Binding, val icit: Icit, val type: Pre, val body: Pre
) : Pre()
data class RSg  (
    val bind: Binding, val type: Pre, val body: Pre
) : Pre()
data class RProjF(val body: Pre, val field: String) : Pre()
data class RProj1(val body: Pre) : Pre()
data class RProj2(val body: Pre) : Pre()

data class RU   (override val loc: Loc) : Pre()
data class RHole(override val loc: Loc) : Pre()
data class RNat(val n: Int) : Pre()
```