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 **Bc. JAKUB ZÁRYBNICKÝ**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

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 Just-in-time (JIT) optimizing compilers like GraalVM or RPython. This thesis evaluates whether such a transition would also benefit dependently-typed programming languages by implementing a minimal language based on the $\lambda\star$-calculus using the Truffle language implementation framework on the GraalVM platform, a partial evaluation-based JIT compiler based on the Java Virtual Machine.

This thesis introduces the type theoretic notion of dependent types, specifies a minimal dependently-typed language, and implements two interpreters for this language: a simple AST-based interpreter, and a Truffle-based interpreter. A number of optimization techniques that use the capabilities of a JIT compiler are then applied to the Truffle-based interpreter. The performance of these interpreters is then evaluated on a number of normalization and elaboration tasks designed to be comparable with other system, and the performance is then compared with a number of state-of-the-art dependent languages and proof assistants.

[...specific numbers]

Abstrakt

[...česky]

Keywords

Truffle, Java Virtual Machine, just-in-time compilation, compiler construction, dependent types, lambda calculus

Klíčová slova

Truffle, Virtuální stroj JVM, just-in-time kompilace, tvorba překladačů, závislé typy, lambda kalkul

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

[...česky]

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 22, 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

Todo list

Chapter 1

Introduction

Proof assistants like Coq, F*, Agda or Idris, or other languages with dependent types like Cayenne or Epigram, allow programmers to write provably correct-by-construction code in a manner similar to a dialog with the compiler [42]. They also face serious performance issues when applied to problems or systems on a large-enough scale [23] [24]. Their performance grows exponentially with the number of lines of code in the worst case [41], which is a significant barrier to their use. While many of the performance issues are fundamentally algorithmic, a better runtime system would improve the rest. However, custom runtime systems or more 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 [8], has improved the performance of several dynamic languages: project like TruffleRuby, FastR, or PyPy. It has allowed these languages to reuse the optimization machinery provided by these platforms, improve their performance, and simplify their runtime systems.

As there are no standard benchmarks for dependently typed languages, we design a small, dependently-typed core language to see if using specific just-in-time (JIT) compilation techniques produces asymptotic runtime improvements in the performance of β-normalization and βη-conversion checking, which are among the main computational tasks in the elaboration process and is also the part that can most likely benefit from JIT compilation. The explicit non-goals of this thesis are language completeness and interoperability, as neither are required to evaluate runtime performance.

State-of-the-art proof assistants like Coq, Agda, Idris, or others is what we can compare our results with. There are also numerous actively developed research projects in this area; Lean is a notable one that I found too late in my thesis to incorporate its ideas. However, the primary evaluation will be against the most well established proof assistants.

Reformulate

As for the languages that use Truffle, the language implementation framework that allows interpreters to use the JIT optimization capabilities of GraalVM, an alternative implementation of the Java Virtual Machine: there are numerous general-purpose functional languages, the most prominent of which are TruffleRuby and FastR. Both were reimplemented on the

Truffle platform, resulting in significant performance improvements[1, 2]. We will investigate the optimization techniques they used, and reuse those that are applicable to our language.

There is also a number of functional languages on the Java Virtual Platform that do not use the Truffle platform, like Clojure, Scala or Kotlin, as well as purely functional languages like Eta or Frege. All of these languages compile directly to JVM byte code: we may compare our performance against their implementation, but we would not be able to use their optimization techniques. To the best of my knowledge, neither meta-tracing nor partial evaluation have been applied to the dependently-typed lambda calculus.

The closest project to this one is Cadenza [34], which served as the main inspiration for 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 project applies similar ideas to the dependently-typed lambda calculus, where the presence of type-level computation should lead to larger gains.

Rephrase

In this thesis, I will use the Truffle framework to evaluate how well are the optimizations provided by the just-in-time compiler GraalVM suitable to the domain of dependently-typed languages. GraalVM helps to turn slow interpreter code into efficient machine code by means of *partial evaluation* [58]. During partial evaluation, specifically the second Futamura projection [38], 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. Depending on the quality of the specializer, this may result in performance gains of several orders of magnitude.

Truffle makes this available to language creators, they only need to create an interpreter for their language. It also allows such interpreters to take advantage of GraalVM's *polyglot* capabilities, and directly interoperate with other JVM-based languages, their code and values [52]. Development tooling can also be derived for Truffle languages quite easily [54]. Regardless of whether Truffle can improve their performance, both of these features would benefit dependently-typed or experimental languages.

While this project was originally intended just as a $\lambda\Pi$ calculus compiler and an efficient runtime, it has ended up much larger due to a badly specified assignment. I also needed to study type theory and type checking and elaboration algorithms that I have used in this thesis, and which form a large part of chapters 2 and 3.

Starting from basic λ-calculus theory and building up to the systems of the lambda cube, we specify the syntax and semantics of a small language that I refer to as Montuno (Chapter 2). We go through the principles of λ-calculus evaluation, type checking and elaboration, 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 an interpreter for Montuno using the Truffle framework (Chapter 4), and apply various JIT optimizations to it (Chapter 6). After designing and using a set of benchmarks to evaluate the language's performance, 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. [51]

[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	$a \times b$ sum type
$p \vee q$ disjunction	$a + b$ product type
$p \Rightarrow q$ implication	$a \rightarrow b$ 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

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

Natural deduction	λ→ calculus
$\dfrac{}{\Gamma_1, \alpha, \Gamma_2 \vdash \alpha}$ axiom	$\dfrac{}{\Gamma_1, x : \alpha, \Gamma_2 \vdash x : \alpha}$ variable
$\dfrac{\Gamma, \alpha \vdash \beta}{\Gamma \vdash \alpha \to \beta}$ implication introduction	$\dfrac{\Gamma, x : \alpha \vdash t : \beta}{\Gamma \vdash \lambda x.t : \alpha \to \beta}$ abstraction
$\dfrac{\Gamma \vdash \alpha \to \beta \qquad \Gamma \vdash \alpha}{\Gamma \vdash \beta}$ 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 natural deduction and λ→-calculus

length in Listing 2.1: the type `Vect` has two parameters, one is the length of the list (a Peano number), 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 given as a finite number between zero and the length of 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, source: the Idris base library

On the other hand, these languages are often restricted in some ways. General Turing-complete languages allow non-terminating programs: non-termination leads to a 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.

2.2 Languages

2.2.1 λ-calculus

We will start from the untyped lambda calculus, as it is the language that all following ones will build upon. Introduced in the 1930s by Alonzo Church as a model of computation, it is a very simple language that consists of only three constructions: abstraction, application, and variables, written as in Figure 2.3.

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

(a) Standard (Church) notation

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

(b) De Bruijn notation

Figure 2.3: λ-calculus written in Church and de Bruijn notation

β-reduction The λ-abstraction $\lambda x.\ t$ represents a program that, when applied to the expression x, returns the term t. For example, the expression $(\lambda x.x\ x)\ t$ produces the expression $t\ t$. This step, applying a λ-abstraction to a term, is called *β-reduction*, and it is the basic *rewrite rule* of λ-calculus. Another way of saying that is that the x is assigned/replaced with the expression T, and it is written as the substitution $M[x := T]$

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

α-conversion We however need to ensure that the variables in the substituted terms do not overlap and if they do, we need to rename them. This is called *α-conversion* or α renaming. In general, the variables that are not bound in λ-abstractions,*free variables.* may need to be replaced before every β-reduction so that they do not become *bound* after substitution.

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

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, it is also 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.

		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

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

ζ-reduction For local variables, equivalent process is called the *ζ-reduction*.

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

ι-reduction 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 projection that extracts the first value of the pair.

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

Normal form By repeatedly βδιζ-reducing an expression–applying functions to their arguments, replacing constants and local variables with their definitions, evaluating objects, and α-renaming variables if necessary, we get a β-normal form, or just *normal form* for short. This normal form is unique up to α-conversion, according to the Church-Rossier theorem.

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

Other normal forms There are also other normal forms, they all have something to do with unapplied functions. If we have an expression and repeatedly use only 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 [50]. 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 the 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).

2.2.2 λ→-calculus

It is often useful, though, to describe the kinds of objects we work with. 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, also written λ→ as "→" is the connector used in types, introduces the concept of types. We have a set of basic types that are connected into terms

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

(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

using the arrow →, and type annotation or assignment $x : A$. We now have two 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].

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 we have terms that include type information, 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 type 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*, or terms of *pre-syntax*.

There are some basis algorithms of type theory, 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 [17].

Types and context The complete syntax of the λ→-calculus is in Figure 2.6. 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. This language 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.

$$
\begin{array}{llll}
e & & & (\mathit{terms}) \\
 & := & v & \text{variable} \\
 & | & M\ N & \text{application} \\
 & | & \lambda x.\ t & \text{abstraction} \\
 & | & x : \tau & \text{annotation} \\
\tau & & & (\mathit{types}) \\
 & := & \beta & \text{base types} \\
 & | & \tau \to \tau' & \text{composite type} \\
\Gamma & & & (\mathit{typingcontext}) \\
 & := & \emptyset & \text{empty context} \\
 & | & \Gamma, x : \tau & \text{type judgement} \\
v & & & (\mathit{values}) \\
 & := & \lambda x.\ t & \text{closure}
\end{array}
$$

Figure 2.6: λ→-calculus syntax

Typing rules The simply-typed λ-calculus can be completely specified by the typing rules in Figure 2.7 [46]. 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.

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

We briefly mentioned the problem of termination in the previous section; the simply-typed λ-calculus is strongly normalizing, meaning that all well-typed terms have a unique normal form. In other words, there is no way of writing a well-typed divergent term; the Y combinator is impossible to type in λ→ and any of the systems in the next chapter [9].

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

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

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

Figure 2.7: λ→-calculus typing rules

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; terms depending on types $(\square, \star)$, 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:

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

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

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

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. This system is also called System F, and it is equivalent to propositional logic [6].

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

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].

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

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 $R = \{(\star, \square)\}$[1] means that a kind can depend on a type using the rule **Product**.

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

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

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 $\lambda\Pi\omega$-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.

$$\begin{array}{ll}
\dfrac{}{\vdash s_1 : s_2}\ (s_1, s_2) \in A & (\textsc{Start}) \\[2ex]
\dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}\ s \in S & (\textsc{Var}) \\[2ex]
\dfrac{\Gamma \vdash x : A \qquad \Gamma \vdash B : s}{\Gamma, y : B \vdash x : A}\ s \in S & (\textsc{Weaken}) \\[2ex]
\dfrac{\Gamma \vdash f : \Pi_{x:A} B(x) \qquad \Gamma \vdash a : A}{\Gamma \vdash f a : B[x := a]} & (\textsc{App}) \\[2ex]
\dfrac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash \Pi_{x:A} B(x) : s}{\Gamma \vdash (\lambda x : A.b) : \Pi_{x:A} B(x)}\ s \in S & (\textsc{Abs}) \\[2ex]
\dfrac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi_{x:A} B(x) : s_3}\ (s_1, s_2, s_3) \in R & (\textsc{Product}) \\[2ex]
\dfrac{\Gamma \vdash a : A \qquad \Gamma \vdash A' : s \qquad A \longrightarrow_\beta A'}{\Gamma \vdash a : A'}\ s \in S & (\textsc{Conv})
\end{array}$$

Figure 2.9: Typing rules of a pure type system

Universes This can be generalized even more. Instantiating this system with an infinite set of sorts $S = \{Type_0, Type_1, ...\}$ instead of the set $\{\star, \square\}$ and setting A to $\{(Type_0, Type_1), (Type_1, Type_2), ...\}$ leads to an infinite hierarchy of *type universes*, and is in fact an interesting topic in the field of type theory. Proof assistants commonly use such a hierarchy [9].

Type in Type Going the other way around, simplifying S to $S = \{\star\}$ and setting A to $\{(\star, \star)\}$, leads 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.

Show Girard's paradox?

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 λΠω-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

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 haven't seen so far; we will define specific *"wired-in"* types instead, 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 $Nat \to Nat \to String$ when called as `printf "%d%d"`.

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

Implicit arguments The type-checker can infer many type arguments. Agda adds the concept of implicit function arguments [9] 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 [37]. 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.

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

Figure 2.11: Π-type syntax

The plain *function type* $A \to B$ 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*.

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

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.

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

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

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

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

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

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

Figure 2.12: Σ-type semantics

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:

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

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 $obj.a$.

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

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 $a : A \vdash \mathit{inl}\ A : A + B$, 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:

$$\mathit{data\ Maybe\ a} = \mathit{Nothing} \mid \mathit{Just\ a}$$

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.

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

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 $\mathit{unit} : \mathit{Unit}$.

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 (constructors) *true* and *false*, and a simple eliminator *if* that returns one of two values based on the truth value of its argument.

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

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

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

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

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 [9]. 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*.

Dependent eliminator too? ncatlab

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

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

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

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

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

Figure 2.16: `Nat` semantics

2.3.4 μ-types

There are multiple ways of encoding recursion in λ-calculi with types, based on whether a recursive expression is delimited using types, or whether it is also reflected in the type of a recursive expression. Recursion must be defined carefully if the type system needs to be consistent, as non-restricted general recursion leads to non-termination and inconsistency. *Iso-recursive types* use explicit folding and unfolding operations, that convert between the recursive type $\mu a.T$ and $T[a := \mu a.T]$, whereas in *equi-recursive types* these operations are implicit and inserted by the type-checker.

As both complicate the type-checker, we will use a simpler value-level recursive combinator fix. While this does compromise the consistency of the type system, it is sufficient for the purposes of runtime system characterization.

The semantics of the function fix are described in Figure 2.17. This definition is sufficient to define e.g., the recursive computation of a Fibonacci number or a local recursive binding as below.

$$\frac{\Gamma \vdash f : A \to A}{\Gamma \vdash \mathit{fix} f : A}\ (\textbf{Type-Fix})$$

$$\frac{\Gamma, x : A \vdash t : A}{\Gamma \vdash \mathit{fix}\ (\lambda x.t) \longrightarrow_{\beta} t[x := (\lambda x.t)] : A}\ (\textbf{Eval-Fix})$$

Figure 2.17: `fix` semantics

```
fib = fix (λf. λn. if (isLess n 2) n (add (f (n - 1)) (f (n-2))))

evenOdd
  : (isEven : Nat → Bool) × (isOdd : Nat → Bool) × Top
  = fix (λf. ( if isZero x then true else f.isOdd (pred x)
             , if isZero x then false else f.isEven (pred x)
             , Top
             ))
```

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 2.18.

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

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

Figure 2.18: `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.

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

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[2]. 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 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:

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

Polyglot Lastly, one language feature that will only be described and implemented in Chapter 4: a *"polyglot"* construct that offers a way to execute code in a different language, which is a feature of the Truffle framework. The selected syntax is a three-part expression that contains the name of language to be used, the foreign code, and the type of the result of evaluating this foreign code:

$$term \quad := \quad [|\ id\ |\ foreign\ |\ term\ |]$$

The syntax and semantics presented here altogether comprise a working programming language. A complete listing of the semantics is included in Appendix B. The syntax, written using the notation of the ANTLR parser generator is in Listing 2.2. The syntax does not mention constants like *true* or *Nat*, as they will be implemented as global definitions bound in the initial type-checking context and do not need to be recognized during parsing.

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.

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

```
FILE : STMT (STMTEND STMT)* ;
STMT : "{-#" PRAGMA "#-}"
     | 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 "|" FOREIGN "|" TERM "]"
     | EXPR "×" EXPR
     | "(" EXPR ("," EXPR)+ ")"
     | "(" EXPR ")"
     | ID "." ID
     | ID
     | NAT
     | "*"
     | "_"
     ;
 STMTEND : ("\n" | ";")+ ;
 ID : [a-zA-Z] [a-zA-Z0-9] ;
 SKIP : [ \t] | "--" [^\r\n]* | "{-" [^#] .* "-}" ;
// pragma discussed in text
```

Listing 2.2: The complete grammar, written using simplified ANTLR

Chapter 3

Language implementation: Montuno

3.1 Introduction

Now with a complete language specification, we can move onto the next step: writing an interpreter. The algorithms involved can be translated from specification to code quite naturally. at least in the style of interpreter we will create at first. 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.

In this chapter we will introduce the algorithms at the core of an interpreter and build a tree-based implementation for the language, elaborating on key implementation decisions. This interpreter will be referred to using the working name Montuno[1].

This interpreter can be called an AST (abstract syntax tree) interpreter, as the principal parts all consist of tree traversals, due to the fact that all 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.

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 functional languages like Scala or Kotlin, as the code often is 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.

[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. "[...] and Kotlin might use abstractions that don't properly partially evaluate." (from https://github.com/oracle/graal/issues/1228)

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

Source

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 . 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.

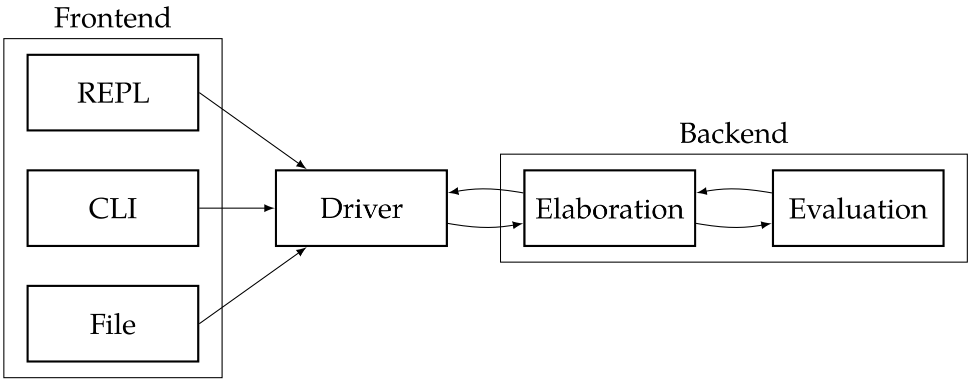


Figure 3.1: Overview of interpreter components

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 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.

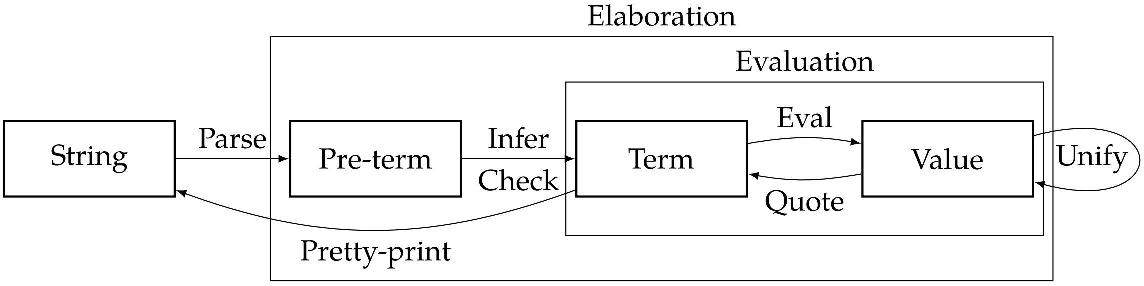


Figure 3.2: Data flow overview

Data flow 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 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 pprint(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

In this chapter, we will first define the data types, 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

We have specified the syntax of the language in the previous chapter, 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 value 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 C. 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 C.

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

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 a 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 has a standard way of representing primarily data-oriented structures using `data classes`. These are classes whose primary purpose is to hold data, so-called Data Transfer Objects (DTOs), and are the recommended approach in Kotlin[4]. In Listing 3.2 we have the base classes for terms and values, and a few examples of structures that map directly from the syntax to a data object.

```kotlin
sealed class Term
sealed class Value

data class TLet(val id: String, val bind: Term, val body: Term) : Term()
data class TSigma(val id: String, val type: Term, val body: Term) : Term()
data class TPair(val left: Term, val right: Term) : Term()

data class VPair(val left: Value, val right: Value) : Value()
```

Listing 3.2: Pair and `let-in` representations

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

Terms that encode computation, whether delayed (λ-abstraction) or not (application) will be more involved. Variables *can* be represented in a straight-forward way, 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 it was created. The body of the function can refer to variables other than its immediate arguments, which means the surrounding environment needs to be stored as well. The simplest example is the *const* function $λx.λy.x, which, when partially applied to a single argument, e.g., let $five = const5$, needs to store the value 5 until it is eventually applied to the remaining second argument: $five15 \longrightarrow 5$.

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., on GHC produces code an order of magnitude faster than using other representations [36]. An example of what it looks like is in Listing 3.3.

```
data class Closure<T>(val fun: (T) -> T)

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

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

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 Closure<T>(val fun: Term, val environment: Map<Name,Term>)

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

Listing 3.4: Defunctionalized function representation

[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.

3.2.2 Variables

Representing variables can be as straight-forward 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 [31]. These numbers are indices that point to the current variable environment, offsets from the top or the top 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 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" [39], as it counts from the start 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 as terms are that substituted *into* an existing value do not need to have the "relative" indices adjusted based on the size of the current environment, whereas the "absolute" addressing of levels in values means that values can be directly compared. This combination of representations means that we can doing avoid 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 perform add and subtract 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.

```
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) : Val()
```

Listing 3.5: Variable representation

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 C, 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.

```
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

	Variables	Functions	Properties
`PreTerm`	String names	`PreTerm` AST	well-formed
`Term`	de Bruijn index	`Term` AST	well-typed
`Value`	de Bruijn level	Closure (`Term` AST + `Values` in context)	normal form

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

3.3 Normalization

3.3.1 Approach

Normalization-by-evaluation 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. An alternative way of bringing terms to normal form is the so-called *normalization-by-evaluation* (NbE) [44]. 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 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, 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 $nf = quote(eval(term))$,

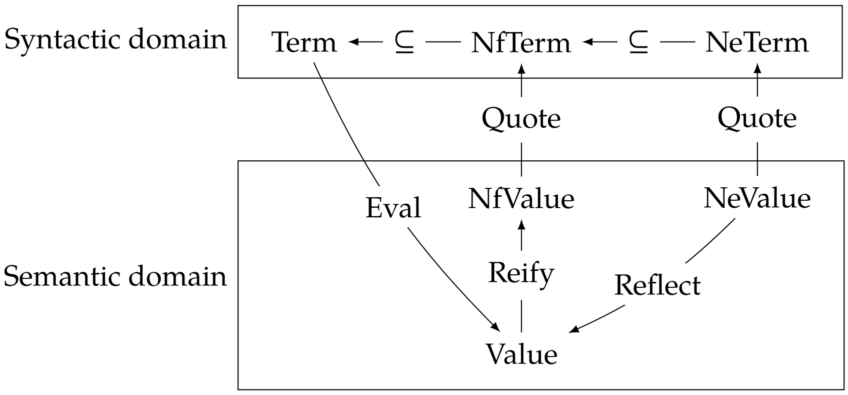


Figure 3.8: Syntactic and semantic domains in NbE [2]

3.3.2 Normalization strategies

Normalization-by-evaluation is, however, at its core inefficient for our purposes [33]. 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 λ-term and a pair are equal, we do not need to compare two fully-evaluated values, but only to find out whether the term is a pair of a λ-term, 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 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 sub-expressions are reduced. Each strategy brings an expression to their corresponding normal form. Common ones are *applicative order* in which we first reduce sub-expressions 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 [50]

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-expressions 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 all commonly used languages with the exception of Haskell. 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, which 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.

3.3.3 Implementation

The basic outline of the implementation is based on Christiansen's [11]. 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 $\lambda\rightarrow$-calculus, we have seen the typing context Γ, to which we add a value context.

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

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] ?: VNeutral(HLocal(Lvl(ctx.lvl - term.ix - 1), spineNil))
  is TLet -> eval(ctx, term.body, env + eval(ctx, term.defn, env))
  is TLam -> VLam(term.name, VCl(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 VNeutral -> VNeutral(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 VNeutral -> {
    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 + VNeutral(HLocal(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

3.4.1 Approach

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 [41], often used in combination with constraint solving.

Judgments The type system is composed of two additional type judgments we haven't 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.

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

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

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

Figure 3.11: Bidirectional typing rules for the $\lambda\to$-calculus

Bidirectional $\lambda\to$ 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 (**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. (**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 [43], 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 [37].

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 $?\beta$.

3.4.2 Unification

Returning to the rule (**ChangeDir**) in Figure 3.12, a critical piece of the algorithm is the equality of two types that this rule uses. 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 $t = u$.

Conversion checking The usual notion of equivalence in λ-calculus is *α-equivalence of β-normal forms*, that we discussed in Chapter 2, which 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.

Unification 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 can produce functions as solutions, with some restrictions.

Renaming In this thesis, I have chosen to reuse the algorithm from [40] 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 arguments to a meta-variable to the variables with which it was unified. 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 [40].

3.4.3 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 is that the type elaboration algorithm has `PreTerm`s as its input, and produces `Term`s in the case of type checking, and pairs of `Term`s and `Value`s (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 pre-term is inferred, resulting in a pair of a well-typed term and its type, the type is unified with the wanted type and, if successful, the produced term is the return value.

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

```
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 that 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)
    Pair(TLet(pre.name, t, a, b), 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. The type produced on the last line of the snippet is a `VPi` unlike the rule, as the rule was written for the λ→-calculus; it is semantically equivalent, however. This rule demonstrates the creation of a new meta-variable as without a placeholder, we are not able to infer the type of the body of the function. This meta-variable might or might not be solved in the course of inferring the body: either way, both the term and the type only contain a reference to a globally-scoped meta-variable and not the solution.

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

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

```
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)
    Pair(TLam(pre.name, b), VPi(pre.name, a, VCl(env, t.quote())))
  }
  // ...
}
```

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

its interaction with the surrounding world. In particular, the parser, pretty-printer, and state management.

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. 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 LetContext -> RLet(id.toAst(), type.toAst(), defn.toAst(), body.toAst())
  is LamContext -> rands.foldRight(body.toAst()) { l, r -> RLam(l.toAst(), r) }
  is PiContext -> spine.foldRight(body.toAst()) { l, r -> l.toAst()(r) }
  is AppContext -> operands.fold(oprator.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 B.

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

[6] `https://www.antlr.org/`

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

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

```
sealed class PreTerm
typealias Pre = PreTerm
typealias N = String

sealed class TopLevel
data class RDecl(val n: N, val type: Pre) : TopLevel()
data class RDefn(val n: N, val type: Pre?, val term: Pre) : TopLevel()
data class RTerm(val cmd: Command, val term: Pre) : TopLevel()

object RU : Pre()
object RHole : Pre()
data class RVar(val n: N) : Pre()
data class RNat(val n: Int) : Pre()
data class RApp(val lhs: Pre, val rhs: Pre) : Pre()
data class RLam(val n: N, val body: Pre) : Pre()
data class RPi(val n: N, val type: Pre, val body: Pre) : Pre()
data class RLet(val n: N, val type: Pre, val defn: Pre, val body: Pre) : Pre()
data class RForeign(val lang: N, val eval: N, val type: Pre) : Pre()
```

Listing 3.11: Data type `PreTerm`

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 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 precedence enumeration `Prec`, the optionally parenthesizing operation `par`, and other constructions of the `kotlin-pretty` library.

```
enum class Prec { Atom, App, Pi, Let }
fun Term.pretty(ns: NameEnv?, p: Prec = Prec.Atom): Doc<Nothing> = when (this) {
  is TVar -> ns[ix].text()
  is TApp -> par(p, Prec.App,
    arg.pretty(ns, Prec.App) spaced body.pretty(ns, Prec.Atom))
  is TLet -> {
    val d = listOf(
      ":".text() spaced ty.pretty(ns, Prec.Let),
      "=".text() spaced bind.pretty(ns, Prec.Let),
    ).vCat().align()
    val r = listOf(
      "let $n".text() spaced d,
      "in".text() spaced body.pretty(ns + n, Prec.Let)
    ).vCat().align()
    par(p, Prec.Let, 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, which is required for handling incremental interpretation or suggestions (tab-completion) in the interactive environment. It also 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, expression by expression supplies it 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*.

```
> cat demo.mt
id : {A} -> A -> A = \x. x
{-# TYPE id #-}
{-# ELABORATE id 5 #-}
{-# NORMALIZE id 5 #-}

> montuno demo.mt
{A} -> A -> A
id {Nat} 5
5
> montuno --type id
{A} -> A -> A
```

Listing 3.13: Example usage of the CLI interface

CLI We will reuse the entry point of Truffle languages, a `Launcher` class, so that integration of the Truffle interpreter is easier later, and so that 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.

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>
Nat Bool zero succ true false if natElim id const
Mt> :normalize id 5
5
Mt> :elaborate id 5
id {Nat} 5
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 bottleneck in their performance [24], and because it involves evaluation of terms, it should be possible to improve 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 the Truffle implements the concept.

4.2 Just-in-time compilation

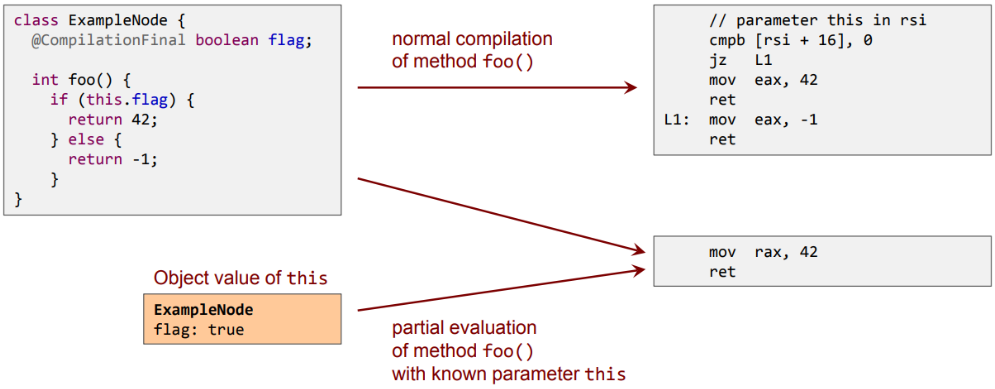
Just-in-time compilation (JIT) is an optimization technique that is based on the assumption that, when executing a program, its functions (and the functions in the libraries it uses) are only called in a specific pattern, configuration, or with a specific type of data. While a program is running, the JIT compiler optimizes the parts of it that run often; using an electrical engineering metaphor, such parts are sometimes called *"hot loops"*.

Often, when talking about specific optimizations, we will use the terms *slow path* and *fast path*. 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 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 [8], or on GHC with mixed results [49].

Partial evaluation The second approach to JIT compilation is called *partial evaluation*, also called the *Futamura projection*. The main principle is as follows: where evaluating (running) an interpreter on a program produces some output, partially evaluating (specializing) the interpreter with regards to a program produces an 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 [38].



```
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
```

```
     mov  rax, 42
     ret
```

```
ExampleNode
flag: true
```

Figure 4.1: Partial evaluation with constant folding (source: oracle.com)

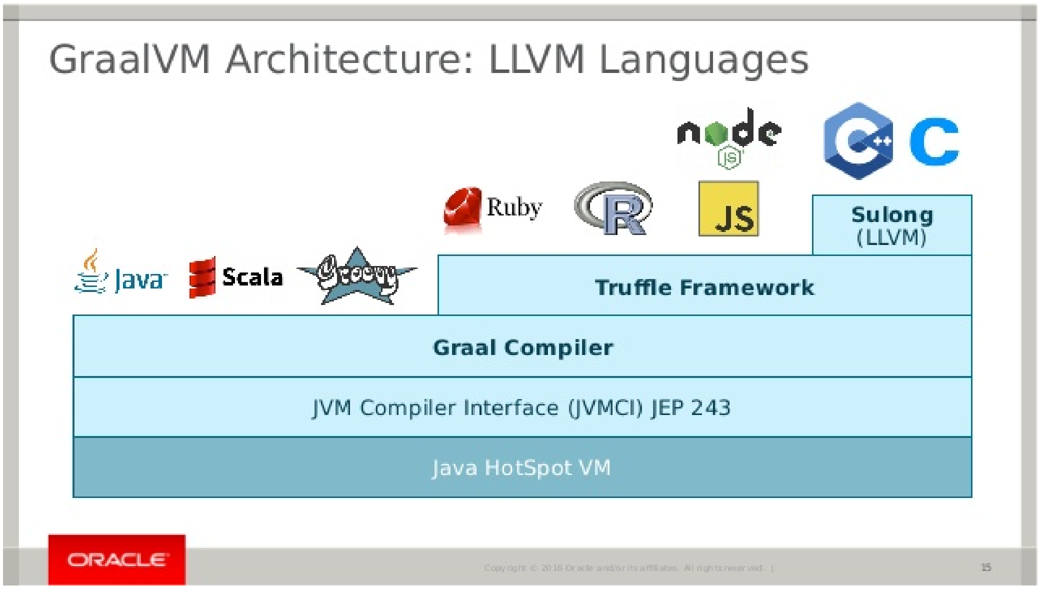
The basic principle is demonstrated in Figure 4.1, on actual code produced by Truffle. In its vocabulary, a `CompilationFinal` value is assumed to be unchanging for a single instance of the program graph node (the field `flag` in the figure), and so the JIT compiler can transform a conditional `if` statement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

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 [38].

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 [12]. 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.



GraalVM Architecture: LLVM Languages

Ruby
JS
Sulong
(LLVM)
Java
Scala
Truffle Framework
Graal Compiler
JVM Compiler Interface (JVMCI) JEP 243
Java HotSpot VM
ORACLE
Copyright © 2016 Oracle and/or its affiliates. All rights reserved. |
15

Figure 4.2: GraalVM and Truffle (source: oracle.com)

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.

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 as the will happen on the processor: see Figure 4.3.

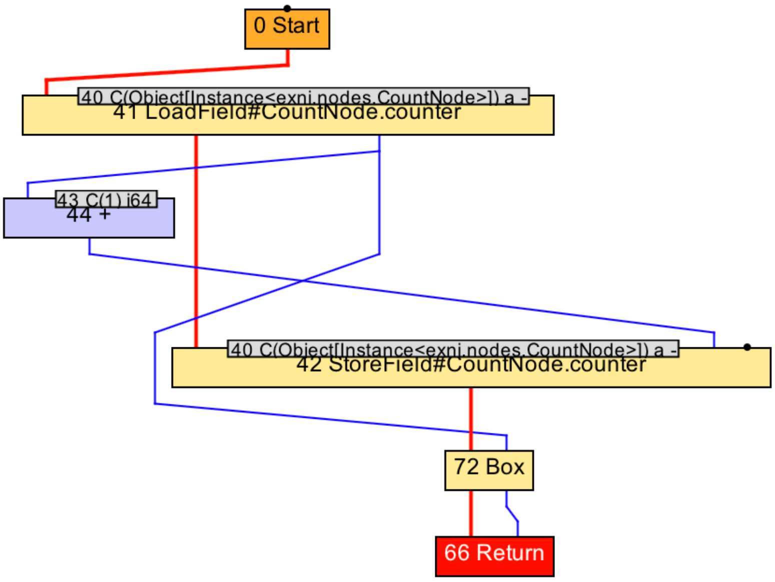

0 Start
40 C(Object[Instance<exni.nodes.CountNode>]) a -
41 LoadField#CountNode.counter
43 C(1) i64
44 +
40 C(Object[Instance<exni.nodes.CountNode>]) a -
42 StoreField#CountNode.counter
72 Box
66 Return

Figure 4.3: Graal program graph, visualized using IGV (source: norswap.com)

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 [58].

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 [12], and it is this capability that makes Truffle possible. Truffle is, in essence, 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 it returns the result of evaluating the expression it represents.

Interpreter/compiler When creating a programming language, There is a trade-off between writing a interpreter and a compiler. An interpreter is straight-forward 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, in can be slower often by a factor to 10x to 100x [58]. A compiler, on the other hand, does not execute a program directly, but instead maps 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 has 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 simplifies 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 then 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

This concludes the general introduction to Truffle and GraalVM. Now we will 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.

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 paren-

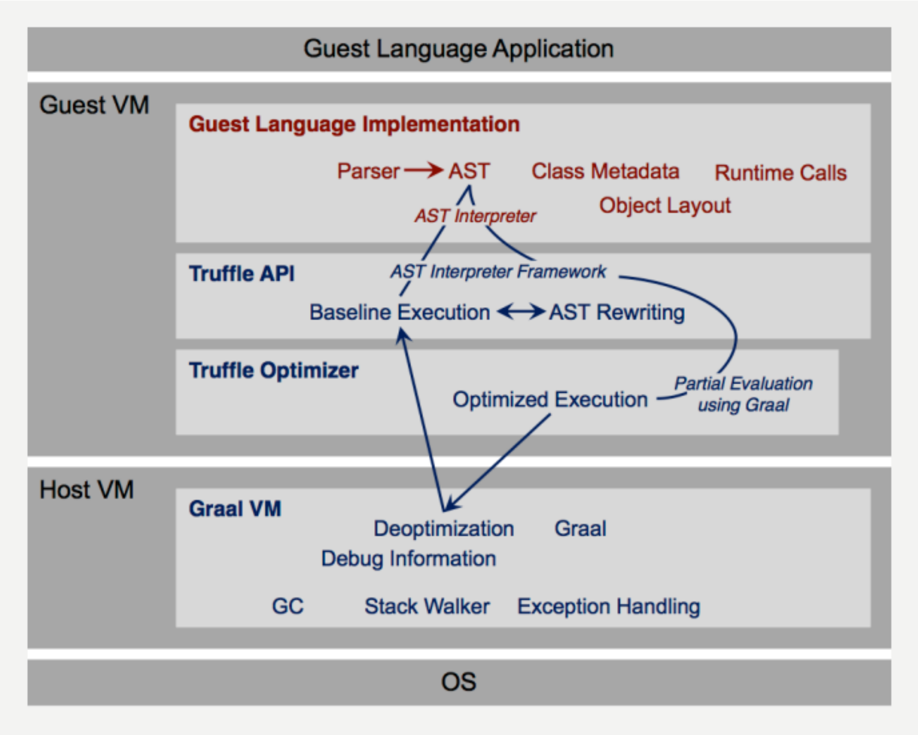


Figure 4.4: Architecture of a Truffle language (source: oracle.com)

theses 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 (`Specialize`, `Profile`, `Assumption`),
- value types (`TypeSystem`, `ValueType`),
- compiler directives (`transferToInterpreter`, `TruffleBoundary`),
- 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 the 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(): Array<String> = 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 A language's primary object is a `Language`, whose primary purpose is to answer `ParsingRequests` with the corresponding program graphs, and to manage execution `Contexts` 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 parseAST = parse(request.source)
        val nodes = parseAST.map { toNode(it, this) }.toTypedArray()
        return Truffle.getRuntime().createCallTarget(ProgramRootNode(nodes))
    }
}
```

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 a single call to a call target corresponds to a single stack `VirtualFrame`, as we will see later. 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.

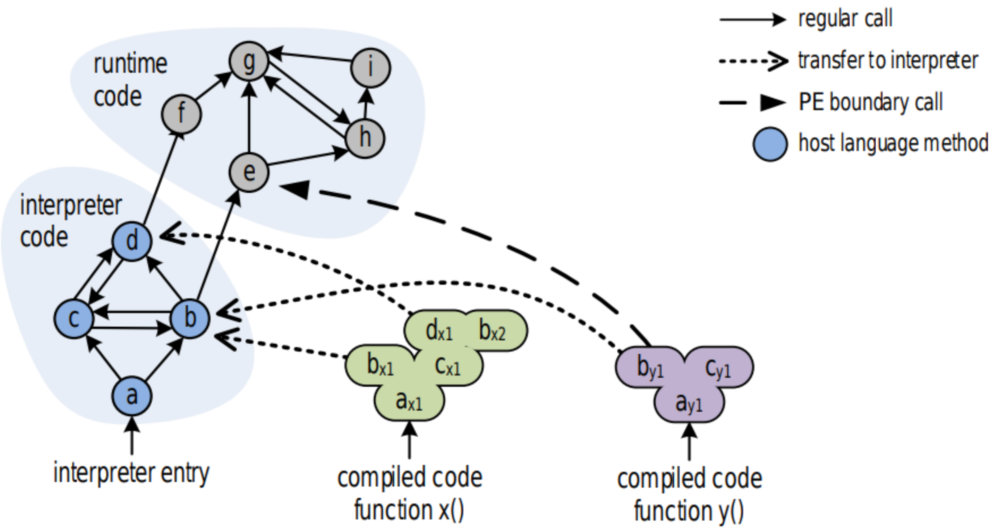


Figure 4.5: Combination of regular and partially-evaluated code (source: oracle.com)

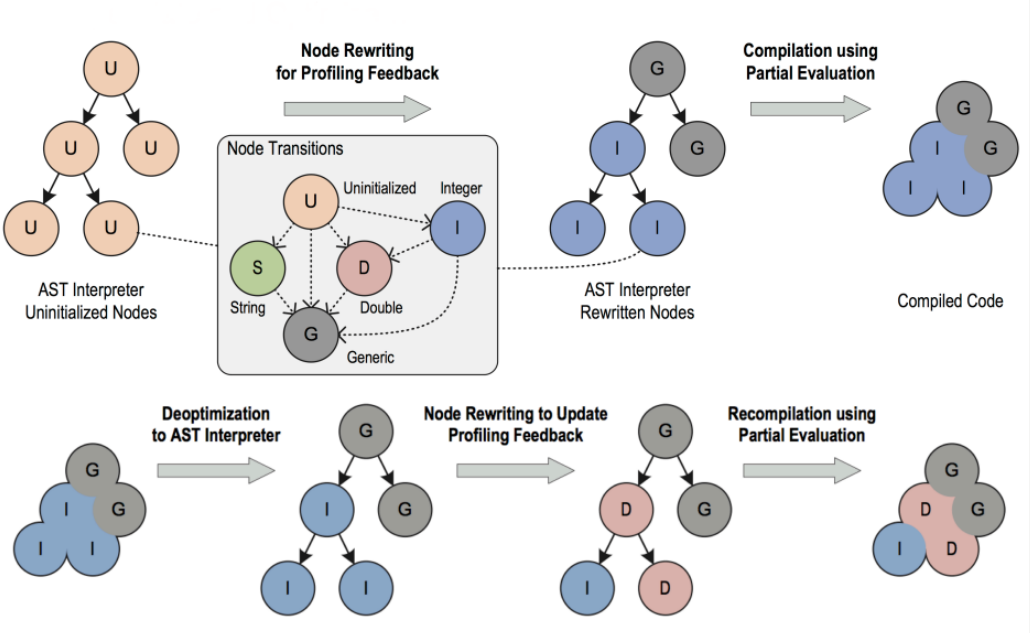
Node execution A `RootNode` is a special case of a Truffle `Node`, the basic building block of the program graph. Each node has a single way of obtaining the result of evaluating the expression it represents, the `execute` method. We may see nodes with multiple `execute` methods later, but they are all 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 with two nodes. They share a parent class, `LanguageNode`, whose only method is the most general version of `execute`: one that takes a `VirtualFrame` 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`.

```
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(left: Int, right: Int) = left + right
  @Specialization fun addString(left: String, right: String) = left + right
  @Fallback fun typeError(left: Any?, right: Any?): Unit
    = throw TruffleException("type error")
}
```

Listing 4.3: Addition with type specialization

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 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 emphasize.



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

Figure 4.6: Node optimization and deoptimization in Truffle (source: oracle.com)

(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 a node reaches a stable state where no more specializations take place, that part of the program graph may be partially evaluated. This produces efficient machine code instead of slow interpreter-based code, specialized for the nodes' current state.

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* [56], 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, asBoolean), and perform implicit conversion between them (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 RootNode and a CallTarget, and evaluates its arguments (5). A DispatchNode creates a CallNode for the specific CallTarget and stores it in a cache, and finally a CallNode 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.

```
class ReadLocalVarNode(val name: String) : Node {
  fun execute(frame: VirtualFrame): Any {
    val slot: FrameSlot = frame.getFrameDescriptor().findFrameSlot(name)
    return frame.getValue(slot ?: throw TruffleException("$name 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

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 it is passed as an argument to all execute functions. Frame layout is set by a FrameDescriptor object, which contains FrameS-

`lots` 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.

There are two kinds of a `Frame`, virtual and materialized frames. A `VirtualFrame` is, as its name suggests, virtual, and the values in it 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 4.12, where `frame.materialize()` captures a virtual frame and stores it in a closure.

```
@CompilerDirectives.ValueType
data class Closure(
  val callTarget: RootCallTarget,
  val frame: MaterializedFrame,
)
class ClosureNode(val root: FunctionRootNode) : Node {
  fun executeClosure(frame: VirtualFrame): Closure = Closure(
    Truffle.getRuntime().createCallTarget(root),
    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 only use 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 stages: *uninitialized;* /*monomorphic*, specialized to a single call target; *polymorphic*, 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. Figure 4.7 demonstrates this on a DispatchNode, adding a polymorphic cache with size 3, and also demonstrates the Truffle DSL annotations `Cached` and `guards`. 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.

Guards Figure 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,

```
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`

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 which branch did code execution take: in the implementation of an `if` statement, or when handling an exception. The compiler will use the information collected during optimization, e.g., when the condition in an `if` statement was true every time, and it is tracked in a `ConditionProfile`, 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 which triggers deoptimization of any code that relies on it. A typical use of assumptions is shown in Figure 4.8 [51], in which 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. These 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 [51]

Inlining During optimization, the Graal compiler replaces `DirectCallNode`s with the contents of the call target they refer to, performing function *inlining* [57]. 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 on the last optimization techniques to be checked when there are no more gains to be gained from caching or specializations.

```
@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` [55]

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 [55] 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`). Figure 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 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.

This is made possible by Truffle *libraries*, that 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[1]

4.5 Mapping concepts to Truffle

We can now move on to the implementation of the second interpreter itself. Many of the features presented will mostly be used only in Chapter 6, as this chapter only aims to create a Truffle interpreter that works, as even Truffle documentation recommends to "First, make it work, then make it fast".

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, 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, but this chapter will only implement the simpler solution and keep elaboration outside of Truffle evaluation, as many changes will be required nonetheless. This optimization will be evaluated in Chapter 6.

Inspiration For inspiration, I have looked at a number of other functional languages that use Truffle: a number of theses (TruffleClojure [16], TrufflePascal [18], Mozart-Oz [30]), two Oracle projects (FastR [53], TruffleRuby [51]), and other projects (Cadenza [34], DynSem [55], Mumbler [15], Truffled PureScript [29]).

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

In the last phases of this thesis, the project Enso [28] was released, that also aims to implement a dependently-typed language using Truffle. While time constraints did not allow me to improve on their approach, I have attempted to incorporate and evaluate several of their innovations, especially in Chapter 6.

4.5.1 Approach

Out of the many changes that are required, the largest is the encoding of functions and closures, replacing data objects with `CallTargets`. Environments and variable references need to be rewritten to use `Frames`, and lazy evaluation cannot use Kotlin's `lazy` abstraction, but instead needs to be encoded as an explicit `Thunk` object.

The representation of the evaluation algorithm will also be different, we need to replace a tree transformation algorithm that processes an inert data structure with object-oriented nodes, where each implements its logic in the `execute` method.

(launcher, language, root, elab, eval, unify, context)

Figure 4.7: Program flow 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 language `Context` that it prepares based on user-provided options. The `Language` object initializes a different `Context` object, a `MontunoContext`, which is an internal object that contains 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 `ProgramRootNode`.

Executing the `ProgramRootNode` starts the elaboration process, where `infer` and `check` build up terms as executable nodes. Any `eval` invocations in the process are then handled by Truffle, producing a `ValueType`. These can 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 `Terms`. The REPL accesses the context as well in order to produce lists of bound variables, and process REPL commands.

FillIn

The data flow in 4.8 makes the data transformations clear, especially the parts where Truffle is involved .

preterm AST, Term Nodes, Value AST + Nodes

Figure 4.8: Data flow inside the Truffle interpreter

4.5.2 Values

Disregarding constants, there are only two main value types per Figure 4.9: 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, that we will attempt to optimize in Chapter 6. Then, there

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

Figure 4.9: Terms and values in Montuno (revisited)

are neutral terms, unresolved variables that accumulate a spine of unapplied operands and projections: these will be 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 class, a `ValueType`, 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 `Thunk`. We need to have this type explicitly mentioned here, to implement lazy evaluation in Truffle later.

We may need to perform a common set of operations on these values, to have them implement a shared interface: the Truffle way is to use a `Library`, which will be mentioned later, as relevant.

```
@TypeSystem(
  Constant::class, Neutral::class,
  Unit::class, Pi::class, Func::class, Pair::class,
  Thunk::class,
  Boolean::class, Int::class, BigInt::class,
)
class Types {
  @TypeCheck(Unit::class)
  fun isUnit(value: Any) = value === Unit
}

@ValueType
object Unit : TruffleObject
@ValueType
class Pair(val left: Any, val right: Any) : TruffleObject
```

Listing 4.11: A `TypeSystem` and two simple `ValueType`s from the Truffle interpreter

Closures There are many possible representations of a closure on Truffle. We will explore some later in Chapter 6, but for now, one simple representation is a simply wrapping a `Term` in a `RootNode`, and storing it alongside the current scope.

A CallTarget can only be called with an array of objects, so this is what a Closure stores. The CallTarget points to a ClosureRootNodw, that first copies the array of arguments it was given into the local scope, and executes the body. Variable names in the local scope (fd.findFrameSlot(i)) are equal to the de Bruijn levels that are used in values. This can all be see in Listing 4.12.

```
@ValueType
class Closure(val env: Array<Any>, val target: CallTarget) : TruffleObject {
  fun call(arg: Object) = target.call(env.plus(arg))
}
@ValueType
class Thunk(val env: Array<Any>, val target: CallTarget) : TruffleObject {
  fun force() = target.call(env)
}
class ClosureRootNode(@NodeChild val body: Node) : Node {
  @Override
  @ExplodeLoop
  fun executeGeneric(f: VirtualFrame): Any {
    val args = f.getArguments()
    val size = args.size
    val fd = f.getFrameDescriptor()
    for (int i = 0; i < size; i++) {
      f.setObject(fd.findFrameSlot(i), args[i])
    }
    return body.execute(f)
  }
}
```

Listing 4.12: A sketch of the closure implementation

4.5.3 Normalization

Terms are be the nodes of the program graph through which flow the above-defined values. Each term is either fully evaluated, or produces one layer of a value.

The Terms share a common super-class, and each defines the correct number of NodeChildren and the relevant execute method. The super-class Term also defines a number of other methods, one for each value type in the TypeSystem. These methods serve to specify the expected return type of a node using the type assertions generated by the TypeSystem, e.g., TypesGen.asUnit(). This is necessary, because the execute method of all nodes returns the generic type Any, and cannot be further constrained.

Variables This is demonstrated in Listing 4.13, which also includes a local variable node TLocal. We will still use de Bruijn indices for terms and de Bruijn levels for values and still receive all the benefits they provide, as mentioned in the previous chapter, but this time any variable references will point to a Frame, and not to an array of values.

λ-abstractions The terms TLam and TPi create a single-argument closure value, shown in Listing 4.14. This closure is created by converting the local scope into an array of objects

```
abstract class Term : Node() {
  abstract fun execute(f: VirtualFrame): Any
  fun executeGeneric(f: VirtualFrame): Any = execute(f)
  fun executeUnit(f: VirtualFrame): Unit = TypesGen.asUnit(executeGeneric(f))
  // ...
}
data class TLocal(val n: Ix) : Term() {
  fun executeGeneric(f: VirtualFrame) {
    val fd = f.getFrameDescriptor()
    return f.getObject(fd.findFrameSlot(fd.getSize() - n - 1))
  }
}
```

Listing 4.13: The super-class `Term`, and a local variable node

(not shown), and including the `CallTarget` that refers to the body of the function. The opposite process, applying a λ-abstraction to an argument requires a `DispatchNode`. The `TApp` node first evaluates the values that make up the function and its argument, and hands them over to a `DispatchNode` that will perform the call.

```
data class TLam(@NodeChild val root: ClosureRootNode) : Term {
  val target = Truffle.getRuntime().createCallTarget(root)
  fun executeClosure(f: VirtualFrame)
    = Closure(frameToEnv(f.materialize()), target)
}
class TApp(@NodeChild val fn: Term, @NodeChild val arg: Term) : Term {
  val dispatchNode = DispatchNode()
  fun executeGeneric(f: VirtualFrame) = dispatchNode.executeDispatch(
    fn.executeClosure(f), arg.executeAny(f)
  )
}
```

Listing 4.14: Demonstration of a λ-abstraction and application in Truffle

Let-in Other constructs follow this pattern. For example, a `let-in` expression first evaluates the value it binds to a variable, assigns it to the `Frame` using a `FrameSlot` that was already computed during the process of elaboration, and then executes the term that contains its body, demonstrated in Listing 4.15.

```
class TLet(
  val fs: FrameSlot,
  @NodeChild val value: Term,
  @NodeChild val body: Term
) : Term() {
  fun executeGeneric(f: VirtualFrame) {
    f.setObject(fs, value.executeAny(f))
    body.execute(f)
  }
}
```

Listing 4.15: `Let-in` expression in the Truffle interpreter

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 elaboration. Each built-in term has its arity, the number of expected arguments. The elaboration process wraps this node with the correct number of λ-abstractions, and resolves the arguments they will produce to an array of arguments. These are passed to a `BuiltinRootNode` that does not copy them to the local scope, unlike the `ClosureRootNode`, but the built-in node uses them directly.

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.asInt(t.force()) + 1
}
```

Listing 4.16: A `Succ` node, an example implementation of a built-in term

```
class TEval(val lang: String, val code: String) : Term {
  fun executeGeneric(
    f: VirtualFrame, @CachedContext(MontunoLanguage::class) ctx
  ) {
    val src = Source.newBuilder(lang, code).build()
    val callTarget = ctx.parsePublic(src)
    return callTarget.call()
  }
}
```

Listing 4.17: The implementation of a foreign *eval* term

Polyglot Lastly, the implementation of the foreign evaluation term that was already mentioned in Chapter 2 is included in Listing 4.17. This term has three components, a language identifier, an expression written in that language, and the type of the expression. Truffle makes it straight-forward to implement such a term. The language identifier and the foreign code are first compiled into a Truffle `Source` object, which is then parsed using the Truffle-provided language `Context`. The context acts as a proxy to the foreign `Language`, which then uses the provided source to create a `CallTarget`. Calling it will result in a foreign value.

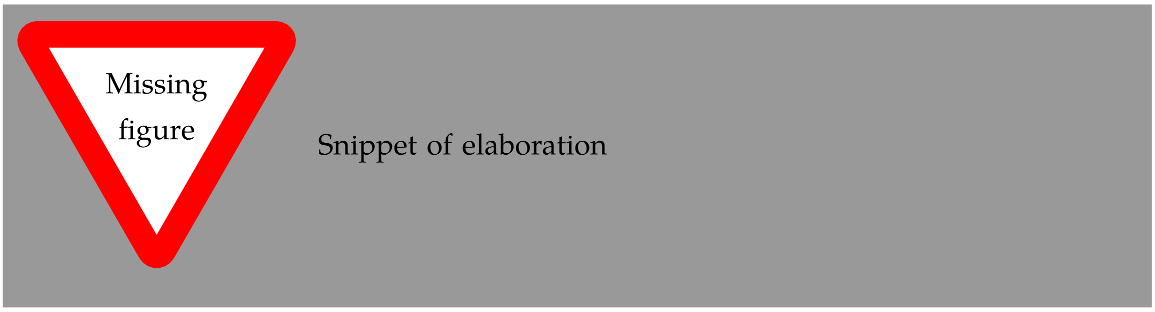
This `TEval` term needs to be wrapped in a node that will map this resulting value to a value that Montuno can use, integers to integers, or arrays to nested Σ pairs, but the result works as expected. Listing 4.18 shows the result of executing a JavaScript equivalent of the `succ` function.

```
Mt> [js|(x) => x + 1|Nat -> Nat|] 5
6
```

Listing 4.18: Calling JavaScript using the *eval* construct

4.5.4 Elaboration

The main change in elaboration is the fact that the functions *infer* and *check* now need to build up the `Terms` as program graphs. This is accomplished by inserting root nodes where necessary, and by keeping a `FrameDescriptor` in the local elaboration context, which is necessary so that we can declare all `FrameSlots` ahead of time and store them in term nodes, so that no variable lookup needs to take place during evaluation.



Snippet of elaboration

Meta-variables are stored in the top-level language context, and meta-variable references use a special node that either looks up the value of a solved meta-variable or forces its evaluation, returning control from a Truffle context back to the external elaboration process.

The changes in the implementation of the driver and frontend were largely described at the start of this section, so they will not be mentioned again.

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 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 Subjects

For the comparison of elaboration and normalization, we will use state-of-the-art proof assistants: I have chosen Coq, Agda, Idris, as they use a dependent type theory at their core. These will be compared to both Montuno and MontunoTruffle on both elaboration and normalization tasks.

Coq Coq is an interactive theorem prover that has been popularized by the proof of the four-color theorem [20]. Coq is a tactics-based theorem prover, meaning that the programmer will use a meta-language of various decision procedures (tactics) instead of writing proofs manually. It uses the Calculus of Inductive Constructions at its core that extends the Calculus of Constructions ($\lambda\Pi\omega$-calculus) introduced in Chapter 2. Its performance has been the subject of research [23, 24], and we will use some of the conclusions reached in later, when discussing optimizations in Chapter 6.

Agda Agda is a dependently-typed programming language that can also be used as a proof assistant. It uses a Haskell-like syntax, and its workflow is centered around meta-variables and their gradual refinement in a dialog with the compiler [42].

Idris Idris is a total, dependently-typed programming language [10]. Like Agda, it also uses a Haskell-like syntax and supports an interactive workflow. Like Coq, proofs in Idris use a tactics meta-language. Recently, a second version of the language was released as a self-hosting language Idris 2; we will attempt to use both in our evaluation.

Cadenza The project to which Montuno is a successor, Cadenza implements a simply-typed λ-calculus [34] using GraalVM and Truffle. It does not use elaboration, but we can evaluate its performance on normalization tasks.

5.2 Workload

The main 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 [36], partially extrapolated from the evaluation part of Jason Gross's doctoral thesis [24].

The third, additional part is a number of computational tasks to evaluate the time and memory performance of the Montuno interpreters. The tasks were mostly adapted from the `nofib` benchmark suite [45] that is used for evaluating the performance of GHC, and from a suite of simple benchmark programs [35] comparing performance across 24 languages. The usual language benchmarks, e.g. the "Benchmarks Game" were disregarded, as their implementation would be too involved.

5.2.1 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, and calculating the Ackermann number followed by evaluating it 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.

The second set of tasks is the evaluation of a term to a normal form. Where the first two tasks only use the *eval* part of the NbE algorithm, this next set also uses *quote* to produce a large normal form of a term, and not to evaluate to a single value. This is evaluated on a large Church-encoded natural number, on a deep Church-encoded binary tree.

Missing figure

(a) forceNat (b) n2M term

Missing figure

(a) ackermann (b) binary tree

5.2.2 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 is similar, the elaboration of a large Σ-type encoding a *Nat* indexed vector. Lastly, the third task especially tests unification: the task is to unify two large Church-encoded natural numbers, to check them for equivalence.

Missing figure

(a) id id id id with and without implicits (b) Sigma vector

Missing figure

Eq, refl, n2Mb

5.2.3 Computation

The last part consists of computational tasks, where we can compare the performance of Montuno with other, non-dependent languages. As Montuno does not implement complex data structures or memory buffers, we will need to limit the types of tasks to primarily computational ones.

I have selected the following: Fibonacci number computation, the previously implemented Ackermann function, solving the N Queens problem, matrix multiplication, and a Brainfuck interpreter.

Missing figure

(a) Brainfuck. (b) matrix multiplication in Montuno

5.3 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.

For this reason, we cannot use whole-program measurement using commands like `time`, which measures the entire run of a program including any interpreter start-up time, parsing, and other tasks unrelated to elaboration or normalization. We will need to use in-language support for measuring elaboration times in those languages that support it, and in those that do not, we will need to postprocess measurements with such confounders.

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`.

Missing figure

(a),(b),(c),(d) Commands for Coq, Agda, Idris, Montuno

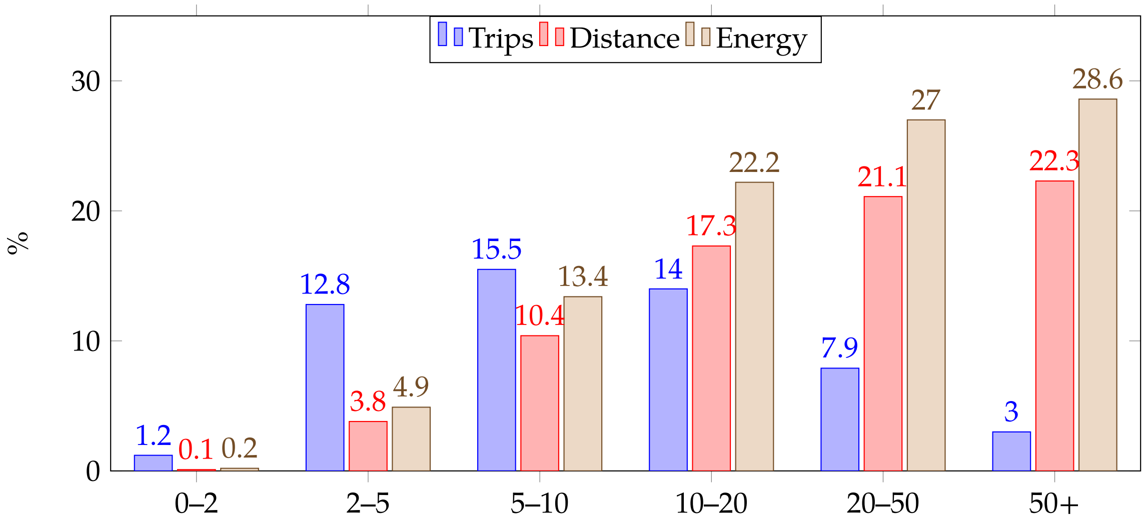


Figure 5.1: Preliminary results of normalization tasks

5.3.1 Configuration

Missing figure

Table with machine specs

5.4 Preliminary results

5.4.1 Normalization

Missing figure

A single large bar graph, grouped by language, four tasks in a group

[...]

5.4.2 Elaboration

Missing figure

A single large bar graph, grouped by language, three tasks in a group

[...]

5.4.3 Computation

Missing figure

A single large bar graph, grouped by language, four tasks in a group

[...]

Chapter 6

Optimizations: Making MontunoTruffle fast

6.1 Possible performance problem sources

Reiterate JGross

how to find out whether X is relevant to us or not? How to prove the effect of JIT?

Show asymptotes - binders, terms, sizes

Show the graphs - large values, many iterations (warmup), sharing

6.2 Possible optimizations

Show before and afters for each optimization

What does Enso do, optimization phases?

What can we do?

Hash consing = sharing structurally equal values in the environment. See below from Kmett: `https://gist.github.com/ekmett/6c64c3815f9949d067d8f9779e2347ef`

Inlining, let-floating

Avoid thunk chaining: box(box(box(() => x))

Frame slot clearing - simplifies Graal's role, as Graal tracks dataflow, and this shortens an object's lifetime

Static optimization - changing the structure of the interpreter so that it would be faster even without JIT

Dynamic optimization - using more Truffle-specific features, so that Graal can more efficiently optimize the code: CompilerDirectives, BranchProfiles, TruffleBoundaries, inline caches, ControlFlowExceptions

“Immutable, except to simplify” + assumptions Maximize evaluation sharing - globals, cache, ?

- [7] - potential optimizations, LLVM impl, closures
- [23] - Coq experience, a few reasons, comparison
- [24] - a lot of reasons in Coq
- [14] - CSE

Ruby uses threads, can we? Automatic parallelism

- [48] - concurrency & parallelism in GHC evaluation
- [27] - CAFs? Lazy evaluation?

Think about the fast vs slow path!

- [59] - reasons for deoptimization

OSM in DynSem:

- DynSem also had to consider concept mapping: a program graph starts with generic node operations that immediately specialize to language-specific operations during their first execution
- HashMaps are efficient, but bring downsides. The Graal compiler cannot see inside the HashMap methods, and so cannot analyze data flow in them and use it to optimize them.
- DynSem also had to deal with runtime specification of environment manipulation as this is also supplied by the language specification. Also split between local short-lived values inside frames, and long-lived heap variables.
- Relevant to us is their use of the Object Storage Model, which they use to model variable scoping which is the processed into fixed-shape stack frames (separate from the Truffle Frames, this is a meta-interpreter). OSM’s use case is ideal for when all objects of a certain type have a fixed shape. This is ideal for us, as tuples and named records have, by definition, a fixed shape (unlike Ruby etc. we do not support dynamic object reshaping, obviously).
- They did it separately from the Virtual/MaterializedFrame functionality to avoid the overhead of MaterializedFrames that Graal cannot optimize away.
- Truffle/Graal discourage the use of custom collections, and instead push developers towards Frames (which support by-name lookups) and Objects (same).

To enhance compilation specialization/inlining:

- Visualizations of call graphs - whether or not node children are stable calls
- Most DynSem calls are not stable calls, they are dispatched on runtime based on arguments - something that Graal does not see as stable (CompilationFinal)
- Two types of rules: mono- and polymorphic. based on whether they are called with different types of values at runtime. Poly- are not inlined
- DynSem found two types: dynamic dispatch (meta-interpreter depended on runtime info), and structural dispatch (based on the program AST and not on values). This is similar to our EvalNode, QuoteNode and similar, which depend on the type of the value
- Overloaded rules–rules with the same input shape–are merged into a single FusedRule node and iterated over with @ExplodeLoop.
- For mono/polymorphic rules, they use an assumption that a rule is monomorphic, specialize the rule, and recompile if it becomes polymorphic.
- Inlining nodes - polymorphic rules reduced to a set of monomorphic rules - a rule from the registry is cloned in an uninitialized state in a monomorphic call site and "inlined" (in a CompilationFinal field)
- They use a central registry of CallTargets that contain rules that they can clone and adopt locally if necessary to specialize–we can do the same!
- Disadvantages: there is more to compile and inline by Graal, instead of a CallTarget, they use a child. Likely to take longer to stabilize, but faster in the end.

6.3 Tools

The results of Montuno need to be further evaluated. Graal and Truffle provide

6.3.1 Ideal Graph Visualizer

A graphical program that serves to visualize the process of Truffle graph optimization. When configured correctly, the IGV will receive the results of all partial evaluations.

6.3.2 CPU Sampler

Running the language launcher with the options `--cpusampler --cpusampler.Delay=MILLISECO` will start the CPU sampler. This tool serves to profile the guest language (as opposed to the regular JDK Async Profiler which will profile the entire process.

`--cpusampler.Delay` helps to not include warm-up time in the results.

Using additional options (`--cpusampler --cpusampler.SampleInternal --cpusampler.Mode` `--cpusampler.Output=json`) and postprocessing the generated JSON with an additional script we can create a so-called flamegraph with the results of the sampling.

6.3.3 Renaissance

[47]

- a set of benchmarks and measurement tools
- measures: synchronized, object.wait, object.notify, atomic ops, park operations, average cpu usage, cache misses, objects allocated, arrays allocated, method calls, dynamic method calls
- needs us to package it in a special way, but useful to compare between truffle optimization versions
- https://github.com/Gwandalff/krun-dsl-benchmarks is an alternative that has examples with Truffle, measures only s/op

6.4 Glued evaluation

An optimization technique that attempts to avoid even more computation.

Parallel operation on two types of values, glued and local. Glued are lazily evaluated to a fully unfolded form; local are eagerly computed to a head-normal form but not fully unfolded, to prevent size explosions. This results in better performance in a large class of programs, although it is not an asymptotic improvement, as we have a small eagerly evaluated term for quoting, and a large lazily evaluated for conversion checking.

This is another case of specialization: we have two operations to perform on the same class of values, but each operation has its own requirements; in this case, on the size of the terms as in quoting we want a small folded value but require the full term for conversion checking.

[32]

https://eutypes.cs.ru.nl/eutypes_pmwiki/uploads/Meetings/Kovacs_slides.pdf

6.5 Splitting

type specializations/dict passing

6.6 Function dispatch

lambda merging

eta expansion

6.7 Caching and sharing

Sharing computation and common values

Multiple references to the same object

let-floating

inlinable functions

6.8 Specializations

Truffle recommended optimizations The optimization workflow recommended by the Truffle developers is as follows:

1. Run with a profiler to sample the application and identify responsible compilation units. Use a sampling delay (–cpusampler.Delay=MILLISECONDS) to only profile after warmup. See the Profiling guide.
2. Understand what is being compiled and look for deoptimizations. Methods that are listed to run mostly in the interpreter likely have a problem with deoptimization.
3. Simplify the code as much as possible where it still shows the performance problem.
4. Enable performance warnings and list boundary calls.
5. Dump the Graal graph of the responsible compilation unit and look at the phase After TruffleTier.
   (a) Look at the Graal graphs at the phases After TruffleTier and After PartialEscape and check if it is what you would expect. If there are nodes there that you do not want to be there, think about how to guard against including them. If there are more complex nodes there than you want, think about how to add specialisations that generate simpler code. If there are nodes you think should be there in a benchmark that are not, think about how to make values dynamic so they are not optimized away.
6. Search for Invoke nodes in the Graal IR. Invoke nodes that are not representing guest language calls should be specialized away. This may not be always possible, e.g., if the method does I/O.
7. Search for control flow splits (red lines) and investigate whether they result from control flow caused by the guest application or are just artifacts from the language implementation. The latter should be avoided if possible.
8. Search for indirections in linear code (Load and LoadIndexed) and try to minimize the code. The less code that is on the hot-path the better.

— Add more info on splitting!!

- `--engine.TraceCompilation` prints a line for each method compilation
- `--engine.TraceCompilationDetail` prints a line for compilation queuing, start, and finish

- `--engine.TraceCompilationAST` prints the entire compiled AST
- `--engine.TraceInlining` prints inlining decision details
- `--engine.TraceSplitting` prints splitting decisions
- `--engine.TraceTransferToInterpreter` prints a stack trace for each explicit invalidation
- `--engine.TracePerformanceWarnings=(call|instanceof|store|all)`
  - `call` prints when PE cannot inline a call
  - `instanceof` prints when PE cannot resolve virtual `instanceof` to a specific type
  - `store` prints when PE store location argument is not compilation final
- `--engine.CompilationStatistics` prints total compilation statistics
- `--engine.CompilationStatisticDetails` prints compilation histograms for each node
- `--engine.TraceMethodExpansion=truffleTier` prints a tree of all expanded Java methods
- `--engine.TraceNodeExpansion=truffleTier` prints a tree of all expanded Nodes
- `--engine.MethodExpansionStatistics=truffleTier` prints total Graal nodes produced by a method
- `--engine.NodeExpansionStatistics=truffleTier` also includes Graal specializations
- `--engine.InstrumentBoundaries` prints info about boundaries encountered (should be minimized)
- `--engine.InstrumentBranches` prints info about branch profiles
- `--engine.SpecializationStatistics` prints details about specializations performed
- `--vm.XX:+TraceDeoptimization` prints all deoptimizations
- `--vm.XX:+TraceDeoptimizationDetails` prints all deoptimizations with details

It is also possible to control what is being compiled, add details to IGV graphs dumped, and print the exact assembly produced: see `https://github.com/oracle/graal/blob/master/truffle/docs/Optimizing.md`.

How to debug specializations **Specialization histogram:** If compiled with -Atruffle.dsl.Generate
and executed with --engine.SpecializationHistogram, Truffle DSL will compile the nodes in a special way and show a table of the specializations performed during the execution of a program.

Example shown at https://github.com/oracle/graal/blob/master/truffle/docs/SpecializationHistogram.md, maybe include the table?

Slow path only: If compiled with -Atruffle.dsl.GenerateSlowPathOnly=true, Truffle will only execute the last, most generic specialization, and will ignore all fast path specializations.

Chapter 7

Discussion

7.1 Results

(A few pages)

One-to-one evaluation and discussion of directly comparable subjects, confidence intervals, likely causes of improvements/regressions, iterations to steady-state.

7.2 Discussion

Size of codebase

Effort required

Effect produced

Is this road viable?

7.3 Next work

(A few pages, subsections/mini-headers)

FFI, tooling

RPython, K Framework - exploration

SPMD on Truffle, Array languages

More type extensions OR totality (as a proof assistent)

Finite types, universes, no type in type, HoTT, CoC

Is this useful at all? What's the benefit for the world? (in evaluation)

next work: LF, techniques, extensions, real language

Chapter 8

Conclusion

We tried X to do Y. It went well and we fulfilled the assignment.

As a side effect, I produced a reference book for functional/dependent language implementation.

Original goal was X, it grew to encompass Y, Z as well.

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. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA. 2017, vol. 1, ICFP, 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] Blaguszewski, M. *Implementing and Optimizing a Simple, Dependently-Typed Language*. Master's thesis.

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

[9] 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.

[10] Brady, E. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.* 2013, vol. 23, no. 5, p. 552–593.

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

[12] 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.

[13] Ebner, G., Ullrich, S., Roesch, J., Avigad, J. and Moura, L. de. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA. 2017, vol. 1, ICFP, 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*. 2016. Available at: https://github.com/cesquivias/mumbler/.

[16] Feichtinger, T. *TruffleClojure: A self-optimizing AST-Interpreter for Clojure/submitted by: Thomas Feichtinger*. 2015. Dissertation. 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. Univerzita Karlova, Matematicko-fyzikální fakulta. 2017.

[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. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA. 2019, vol. 3, ICFP, 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] Hughes, R. J. M. Super-Combinators: A New Implementation Method for Applicative Languages. In: *In Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*. ACM, 1982, p. 1–10.

[28] Inc., N. B. O. *Enso-org/enso*. 2021. Available at: https://github.com/enso-org/enso/.

[29] Inc., S. *Slamdata/truffled-purescript*. 2015. Available at: https://github.com/slamdata/truffled-purescript/.

[30] Istasse, M., Van Roy, P., Daloze, B. and Maudoux, G. An Oz implementation using Truffle and Graal. 2017.

[31] 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.

[32] Kaposi, A., Huber, S. and Sattler, C. Gluing for type theory. In: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. 2019.

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

[34] Kmett, E. *Ekmett/cadenza*. 2019. Available at: https://github.com/ekmett/cadenza/.

[35] Kostya, M. *Kostya/benchmarks*. 2020. Available at: https://github.com/kostya/benchmarks/.

[36] Kovács, A. *AndrasKovacs/normalization-bench*. 2020. Available at: https://github.com/AndrasKovacs/normalization-bench/.

[37] Kovács, A. Elaboration with First-Class Implicit Function Types. *Proc. ACM Program. Lang.* New York, NY, USA: Association for Computing Machinery. august 2020, vol. 4, ICFP. Available at: https://doi.org/10.1145/3408983.

[38] 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*. New York, NY, USA: Association for Computing Machinery, 2019, p. 29–31. SPLASH Companion 2019. Available at: https://doi.org/10.1145/3359061.3361077. ISBN 9781450369923.

[39] 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.

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

[41] 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. Available at: http://arxiv.org/abs/1912.03028.

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

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

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

[45] Partain, W. The nofib benchmark suite of Haskell programs. In: *Functional Programming, Glasgow 1992*. Springer, 1993, p. 195–202.

[46] Pierce, B. C. and Benjamin, C. *Types and programming languages*. MIT press, 2002.

[47] Prokopec, A., Rosà, A., Leopoldseder, D., Duboscq, G., Tuma, P. et al. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2019, p. 31–47.

[48] Reid, A. Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable black-holes. In: Springer. *Symposium on Implementation and Application of Functional Languages*. 1998, p. 186–199.

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

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

[51] Shopify. *Optimizing Ruby Lazy Initialization in TruffleRuby with Deoptimization*. Mar 2020. Available at: https://shopify.engineering/optimizing-ruby-lazy-initialization-in-truffleruby-with-deoptimization.

[52] 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.

[53] Stadler, L., Welc, A., Humer, C. and Jordan, M. Optimizing R language execution via aggressive speculation. *ACM Sigplan Notices*. ACM New York, NY, USA. 2016, vol. 52, no. 2, p. 84–95.

[54] Stolpe, D., Felgentreff, T., Humer, C., Niephaus, F. and Hirschfeld, R. Language-Independent Development Environment Support for Dynamic Runtimes. In: *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages*. New York, NY, USA: Association for Computing Machinery, 2019, p. 80–90. DLS 2019. Available at: https://doi.org/10.1145/3359619.3359746. ISBN 9781450369961.

[55] 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.

[56] 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*. New York, NY, USA: Association for Computing

Machinery, 2017, p. 55–64. CC 2017. Available at: https://doi.org/10.1145/3033019.3033025. ISBN 9781450352338.

[57] 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.

[58] 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.

[59] Zheng, Y., Bulej, L. and Binder, W. An empirical study on deoptimization in the graal compiler. In: Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. 2017.

Appendices

Appendix A

Contents of the attached data storage

Appendix B

Language specification

B.1 Syntax

```
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=term #Lam
    | 'let' id=binder ':' type=term '=' defn=term 'in' body=term #LetType
    | 'let' id=binder '=' defn=term 'in' body=term #Let
    | (spine+=piBinder)+ ARROW body=term     #Pi
    | sigma ARROW body=term                  #Fun
    | sigma                                  #LamTerm
    ;
sigma
    : '(' binder ':' term ')' TIMES term #SgNamed
    | app TIMES 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
```

B.2 Semantics

B.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
- if : {A} → Bool → A → A → A
- fix : {A} → (A→A) → A
- the : (A) → A → A
- eval : {A} → String → A
- typeOf : {A} → A → Type

Appendix C

Montuno

C.1 Pre-terms

```
package montuno.syntax

sealed class TopLevel : WithLoc
data class RDecl(override val loc: Loc, val n: String, val ty: PreTerm) : TopLevel(
data class RDefn(override val loc: Loc, val n: String, val ty: PreTerm?, val tm: Pre
data class RTerm(override val loc: Loc, val cmd: Pragma, val tm: PreTerm?) : TopLeve

sealed class PreTerm : WithLoc

data class RVar (override val loc: Loc, val n: String) : PreTerm() { override fun te
data class RApp (override val loc: Loc, val arg: ArgInfo, val rator: PreTerm, val ra
data class RLam (override val loc: Loc, val arg: ArgInfo, val bind: Binding, val bod
data class RPair(override val loc: Loc, val lhs: PreTerm, val rhs: PreTerm) : PreTe
data class RLet (override val loc: Loc, val n: String, val type: PreTerm?, val defn
data class RPi  (override val loc: Loc, val bind: Binding, val icit: Icit, val type
data class RSg  (override val loc: Loc, val bind: Binding, val type: PreTerm, val bo
data class RProjF(override val loc: Loc, val body: PreTerm, val field: String) : Pre
data class RProj1(override val loc: Loc, val body: PreTerm) : PreTerm()
data class RProj2(override val loc: Loc, val body: PreTerm) : PreTerm()

data class RForeign(override val loc: Loc, val lang: String, val eval: String, val

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