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

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 performnce 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 kalkul

Klíčová slova

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

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 6, 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 [40]. They also face serious performance issues when applied to problems or systems on a large-enough scale [21] [22]. Their performance grows exponentially with the number of lines of code in the worst case [39], 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 [9], 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 $\beta\eta$-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 is also a large number of research projects being actively developed 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. Reformulat

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 [31], 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* [52]. During partial evaluation, specifically the second Futamura projection [33], 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 [46]. Development tooling can also be derived for Truffle languages rather easily [48]. 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 λΠ 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 (Chapter 4), implement an interpreter for Montuno on the Truffle framework (Chapter 5), 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 7).

[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. [45]

[2] FastR is between 50 to 85x faster than GNU R, depending on the source. [17]

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 [6]. 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	$a \to b$ exponential (function) type
$\forall x \in A, p$ universal quantification	$\Pi_{x:A} B(x)$ dependent product type
$\exists x \in A, p$ existential quantification	$\Sigma_{x:A} B(x)$ dependent sum type

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

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

Figure 2.2: Curry-Howard correspondence between natural deduction and $\lambda\!\rightarrow$-calculus

length in Listing 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 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.

e	::=	v	variable
	|	$M\ N$	application
	|	$\lambda v.\ M$	abstraction

(a) Standard (Church) notation

e	::=	v
	|	$(N)\ M$
	|	$[v]\ M$

(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 args \ Reduce under abstraction	**Yes**	**No**
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.

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

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 [44]. 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:

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

Also notable is the fixed-point function, or the Y combinator.

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

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

$$Y\,f \longrightarrow_{\delta\beta} f\,(Y\,f) \longrightarrow_{\delta\beta} f\,(f\,(Y\,f)) \longrightarrow_{\delta\beta} \ldots$$

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 $\rightarrow$, 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 [23].

Church- and Curry-style There are two ways of formalizing the simply-typed λ-calculus: $\lambda\rightarrow$-Church, and $\lambda\rightarrow$-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

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

or using parentheses to clarify the precedence

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

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 $\lambda x.x : A \rightarrow A$. [7].

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 [16].

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.

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

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 [41]. 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.

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

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 $\lambda\!\to$ and any of the systems in the next chapter [10].

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

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

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

Figure 2.7: $\lambda\!\to$-calculus typing rules

2.2.3 λ-cube

The $\lambda\!\to$-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 $\lambda\!\to$-calculus can be organized into a cube called the Barendregt cube, or the λ-cube [7] (Figure 2.8). In $\lambda\!\to$ only terms depend on terms, but there are also three other combinations, represented as the three dimensions of the cube: types depending on types $(\Box, \Box)$, which represents higher-order types, also called type operators; terms depending on types $(\Box, \star)$, called *polymorphism*; and terms depending on types $(\star, \Box)$, which represents *dependent types*, and is the reason for the name of dependently-typed languages.

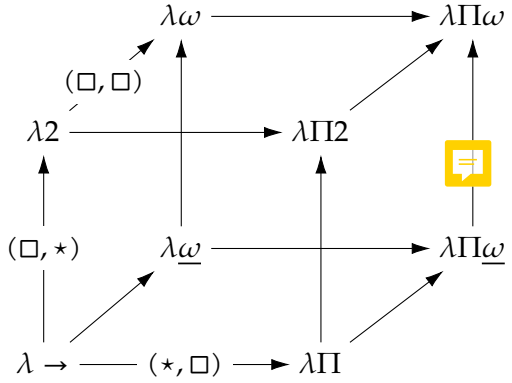

Figure 2.8: Barendregt cube (also λ-cube)

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 $\lambda\to$ is here:

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

$\lambda\underline{\omega}$-calculus Higher-order type operators, the dependency of types on types $(\square, \square)$, simply generalize the concepts of functions to the type level, adding λ-abstractions and applications to the language of types.

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

$\lambda 2$-calculus The dependency of terms on types $(\square, \star)$ 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.

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

$\lambda\Pi$-calculus Types depending on terms, the dependency $(\star, \square)$, allows the function type to depend on its term-level arguments, hence the name dependent types. This dependency adds the Π-type: $\Pi a : A.B(a)$. This system is well-studied as the Logical Framework (LF).

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

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).

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

The typing rules in Figure 2.9 apply to all of 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 $\lambda{\rightarrow}$-calculus, whereas including all of 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 $A \rightarrow B$ is exactly equivalent to the type $\Pi a : A.B(a)$ if the variable a is not used in the expression $B(a)$, the similarity to Figure 2.7 should be easy to see.

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

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

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

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

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

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

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

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 [10].

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.

Maybe 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 $\lambda\Pi\omega$-calculus, which is called the Calculus of Constructions. They add more extensions: induction and subtyping are common ones. We will discuss only a subset of them in the following section, as many of these are irrelevant to the goals of this thesis.

2.3 Types

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 $\lambda\!\to$-calculus function application, with the exception of 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 `printf` function that, when called as `printf "%d%d"`, produces a function $Nat \to Nat \to String$.

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

Implicit arguments The type-checker can infer many type arguments. Agda adds the concept of implicit function arguments [10] 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 [32]. 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.

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

Figure 2.11: Π-type syntax

In the plain *function type* $A \to B$ is simple to type but does not bind the value of 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*.

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

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:

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

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 $obj.a$.

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

Figure 2.14: Σ-type syntax

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

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

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

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

2.3.3 Value types

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

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

Booleans The above-mentioned type **2** has two inhabitants and can semantically be 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.

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

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

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

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

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 [10]. The constructions introduced in Figure 2.16 are simply the constructors *zero* and *succ*, and the destructor *natElim* that unwraps at most one layer of *succ*.

Alternatively, a dependent evaluator into ΠIn:Nat.T(n) although unused, e.g. ncatlab

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

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

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

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

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

Figure 2.16: `Nat` semantics

2.3.4 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.

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

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

Figure 2.17: `fix` semantics

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.

```
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.

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

Holes A construct that serves solely as information to the compiler and will not be used at runtime is a *hole*, which 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.

$$
term := \_
$$

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, the syntax for one language feature that will only be described and implemented in Chapter 5: a way to execute code in a different language in a *"polyglot"* mode, which is a feature that Truffle offers. This is a three-part expression containing the name of language to use, the foreign code, and the type this expression should have:

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

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 Figure 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: Montuno syntax (using simplified ANTLR syntax)

Chapter 3

Language implementation: Montuno

3.1 Introduction

We will first create an interpreter for Montuno as specified in the assignment, and also because evaluation and elaboration algorithms from the literature are quite naturally translated to a functional-style program, which is not really possible in Truffle, the target implementation, where annotated classes are the main building block.

The Truffle implementation, which we will see in the following chapters, has a much higher conceptual overhead as we will need to care about low-level implementation details, e.g. implementing the actual function calls. In this interpreter, though, we will simply use the features of our host language.

We will use Kotlin as our language of choice, as it is a middle ground between plain Java and functional JVM-based languages like Scala or Clojure. While the main target language of Truffle is Java, Kotlin also supports class and object annotations on which the Truffle DSL is based [1]. Functional style, on the other hand, makes our implementation of the algorithms simpler and more concise [2].

The choice of the platform (JVM) and the language (Kotlin) also clarifies the choice of supporting libraries. In general, we are focused on the algorithmic part of the implementation, and not on speed or conciseness, which means that we can simplify our choices by using the most widely used libraries:

- Gradle as the build system,
- JUnit as the testing framework,
- ANTLR as the parser generator,
- JLine as the command-line interface library.

[1] 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)

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

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. This means that we need to design our own supporting data structures based on the primitive structures provided directly by Java.

The overall program flow of our interpreter will not be unusual for interpreters:

- receive some input from the user, either from a file, or from an interactive prompt;
- parse the textual content into an intermediate representation;
- if we are in batch mode, sequentially process the top-level statements, accumulating the entries processed so far into a global table of names. Given a top-level definition containing a type and a term, we will:
  - *check* that the type provided has the kind $\star$
  - *infer* the type of the term
  - *check* whether the provided and inferred types can be unified
  - *simplify* both the type and term, and store them into the global name-table.
- if we are in interactive mode, normalize the expression given and print it out;
- if we encounter an `elaborate` or `normalize` command, print out the full or normalized form of the expression.

```
fun infer(pre: PreTerm): Pair<Term, Val>
fun check(pre: PreTerm, wanted: Val): Term
fun eval(term: Term): Val
fun quote(value: Val): Term
```

Listing 3: Simplified signatures of the principal functions

Throughout this process, we will evaluate and quote `Terms` from and into `Values`. The signatures of the most important functions are mentioned in Listing 3. These are the main parts of our interpreter–and the main component of evaluation is function application, which is what we will focus on first.

3.2 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 [3]. It comes with a large library of languages and protocols from which to take inspiration [4], so creating a parser for our language was not hard, despite me only having prior experience with parser combinator libraries and not parser generators.

[3] https://www.antlr.org/

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

ANTLR provides has two recommended ways of consuming the result of parsing using classical design patterns: a listener and a visitor. I have used neither as they were needlessly verbose or limiting [5].

I have instead implemented a custom recursive-descent AST transformation that converts `ParseContexts` created by ANTLR into our `Presyntax` data type that we can see in Listing 4. This is actually a slightly simplified version compared to the original as I have omitted the portion that tracks which position of the input file corresponds to each subtree, which is later used for type error reporting.

```
sealed class TopLevel
data class RDecl(val n: String, val type: PreTerm) : TopLevel()
data class RDefn(val n: String, val type: PreTerm?, val term: PreTerm) : TopLevel()
data class RTerm(val cmd: Command, val term: PreTerm) : TopLevel()

sealed class PreTerm

data class RVar(val n: String) : PreTerm()
data class RNat(val n: Int) : PreTerm()
object RU : PreTerm()
object RHole : PreTerm()

data class RApp(
    val icit: Icit, val rator: PreTerm, val rand: PreTerm
) : PreTerm()
data class RLam(
    val name: String, val icit: Icit, val body: PreTerm
) : PreTerm()
data class RFun(
    val domain: PreTerm, val codomain: PreTerm
) : PreTerm()
data class RPi(
    val name: String, val icit: Icit, val type: PreTerm, val body: PreTerm,
) : PreTerm()
data class RLet(
    val name: String, val type: PreTerm, val defn: PreTerm, val body: PreTerm,
) : PreTerm()
data class RForeign(
    val lang: String, val eval: String, val type: PreTerm
) : PreTerm()
```

Listing 4: The Presyntax data type

The data structures are represented in a way that Kotlin recommends [6]–using *data classes*. These are classes whose primary purpose is to hold data, so called Data Transfer Objects (DTOs), and that have special language support in Kotlin. We have a `TopLevel` class with three children that represent: definitions that assign a value to a name, optionally with a type; declarations (sometimes called postulates) that only assign a type to a name; and commands like `%normalize` that we will see in later sections.

[5] 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.

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

Their ancestor is a `sealed class` which tells the compiler there will only ever be the subclasses defined in this module. In particular, this means that in any pattern match on the type of a `TopLevel` object we only ever need to handle three cases.

The remaining classes generally map to elements from the language syntax as specified in the previous section: a λ-abstraction, function application, Π-abstraction, a `let` local binding. The `RFun` class is a specialization of the `RPi` class that binds an unnamed, non-dependent type argument.

We will not use these classes immediately–only in Chapter 3.6 will we implement a way to convert this pre-syntax into correct-by-construction `Syntax` objects which can be evaluated and quoted.

3.3 Representing values

RecapValues

The main requirement for a λ-calculus runtime system is fast function evaluation, which is where we will start.

Closures as values of the λ-calculus

- [19] - normal values, closures

Three constructs - lam, app, var (1 para)

A closure consists of an unapplied function, and an environment that *closes over* the free variables used in the function.

HOAS is a specific representation of closures, wherein the function is represented as function in the host language, using the host language's support for closing over free variables.

While 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 explicit closures, this is not possible for us using Truffle, as function calls need to be nodes in the program graph and therefore objects, as we will see in Chapter 4, so we will represent closures using explicit closures in the pure interpreter as well.

Cite Kovacs benchmarks where HOAS on GHC wins

HOAS vs Closure code example

Single versus multi-argument (1 para)

Resulting data structure (TLam, VLam, VCl) (1 figure)

3.3.1 Primitive operations

A primop needs:

- a symbol (nullary constructor)
- a neutral head (when it gets stuck)
- evaluation rule in eval (as many VLams as necessary to apply it, so that the computation rule can be tested)

Not projections, though:

- Fst:Term, (`eval env Fst = VLam \$ \v -> vFst v`),
- projections should go to spines, not heads
- neutrals should make access to the Ex that blocks computation as easy as possible,
- see https://github.com/AndrasKovacs/setoidtt/blob/master/proto/Evaluation.hs#L293 for an example

3.4 Representing variables and environments

The way we define functions leads us to environments (Γ), which is a context where we will look for variables.

Named versus nameless representation (2 paras, example)

As we saw in , evaluation in λ-calculus is defined in terms of α- and β-reductions. Verifying expression equivalence also uses variable renaming. However, traversing the entire expression in the course of function application and substituting variables is not efficient and there are several alternative ways.

Where's WHNF?

In typed λ-calculus implementations in "real world" applications, evaluation in an abstract machine or into a different semantic domain are used instead. Then we have a clear distinction between terms (as the source code) and runtime objects (on which we perform reductions). Machines like STG, Zinc, SECD, ... Normalization-by-evaluation works in this way, we eval/quote, [...]

3.4.1 Variables and names

Indices are widely used but drawback: use numbers instead of names, and a single value can be referenced by multiple different numbers and keeps changing.

Levels less used, "reversed de Bruijn indexing", stable names.

[34]

Depth of a substitution - indices change during a renaming

All three of them mentioned here work by replacing variable names with their indices in a variable stack, informally *counting the lambdas*.

- de Bruijn indices - well-known, starting from the current innermost λ

	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 3.1: Normal forms in λ-calculus

- de Bruijn levels - less well-known, starting from the outer lambda, stable
- locally-nameless - dB indices for bound variables,, names for free variables

Given an environment stack, indices count from the start of the stack, levels count form the bottom. Two ways of indexing the environment, indices useful for a stable context, levels without a stable context.

- [28] (pros & cons)
- [14] - motivation for indices/levels

???

- [30] - STG compiler JITed?
- [43] - trace-based interpreter for GHC, a different approach
- [27] - well described contexts + language specification - can I take as inspiration?
- [1]
- [4], [2] - "I had discovered the ΠΣ paper when finishing my thesis: too late, unfortunately"
- [14] is an interesting tutorial of a dependent interpreter of dependent languages

In values, we have a known context, so it is easier to carry a level counter around, and use levels in variables = Val → Term uses levels

In terms, we do not have a known context, so it makes sense to refer to use indices. If we used levels, we would have to renumber them anyway, if we e.g. performed an unfolding or otherwise used the term in a bigger context. = Term → Val uses indices.

It is trivial to convert between them: given a context of size equal to the current depth level d, level to index is $d - lvl - 1$ and index to level is $d - ix - 1$.

3.4.2 Environments

Environments - arrays, cons lists, stack, mutable/immutable (2 paras)

Snippet of code - binding a variable, looking up a variable (1 figure)

3.5 Normalization

Equals (W) (H) NF, Single-variable functions - trivial transcription (1 para)

What are our neutrals (1 para?)

Eval Quote, very briefly (2 paras)

Extensions (Pi, Sigma, Eta) (2 paras)

3.5.1 Normalization strategies

There are *normalization strategies* that specify the order in which sub-expressions are reduced. 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. These closely correspond to the call-by-need and call-by-value evaluation strategies, differing only in ...

We do not want to do η-reduction, as it might introduce non-termination or undecidability (and in general is not compatible with subtyping relations). η-expansion is sound for producing $\beta\eta$-long normal forms.

If we do not have a $\beta\eta$-long normal form and want to unify/compare (λx:A.M) with N, you unify/compare

$$N\ X$$

and

$$M[x := X]$$

for a fresh X. This is sufficient for $\beta\eta$-equality (for the easy η-rule for Π/λ).

Normalization-by-evaluation:

- normalization = bring an expression with unknowns into a canonical form
- evaluation = compute the value of an expression relative to an environment
- machine code: normalizer ~ JIT compiler, evaluator ~ stack machine (env = stack)
- NbE = adapt an interpreter to simplify expressions with unknowns

NbE works with any abstract machine, like STG, Zinc, ... - a term only needs to be evaluated into a final value - these values are manipulated, compared, and finally quoted/uneval'ed back into a term.

NbE similar to machine code + equations from http://www.cse.chalmers.se/~abela/talkHabil2013.pdf

3.5.2 Evaluation

- [14] - exact algorithm description

$$x \xrightarrow{name} x$$

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

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

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

(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 3.2: Reduction strategies for λ-calculus

[44]

One evaluation strategy for one normal form each! We will only use call-by-name to WHNF, and normal order to NF.

Careful, call-by-name ≅ normal order, not "is very closely related"

While reduction strategies mentioned (many pages back) are related to the evaluation strategy and rewriting rules specify the behavior of an evaluator exactly, "real-world" implementations of the λ-calculus do not use rewriting as their primary method of evaluation.

What is commonly done is a translation of terms into another domain, which then produces values, which are then compared or unified or whatever. This domain can be an interpreter, compiled code, ...

First, however, we need to introduce evaluation models that connect reduction strategies to terms used in implementation of programming languages.

Call-by-value, otherwise called eager evaluation, corresponds to applicative order reduction strategy [5]. 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, 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 lazy evaluation is emulated using closures or zero-argument functions: e.g.,

$$\begin{aligned} A &::= UB|\Sigma(i \in I)A_i|1|A \times A \\ B &::= FA|\Pi(i \in I)B_i|A \to B \end{aligned}$$

Figure 3.3: Call-by-push-value values and computations

in Kotlin a variable can be initialized using the syntax `val x by lazy { ackermann(4, 2) }`, and it will be initialized only if is ever needed.

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 does this by defining additional operators *delay* and *force*, 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.

Levy's call-by-push-value [35] formalism subsumes both (by means of a translation strategy): it defines a single evaluation order in terms of operations with a stack using two additional operators. It **distinguishes values and computations**, operator U (delay) creates a *thunk*, a delayed computation, and operator F (force) that forces its evaluation.

- value of type UB is a thunk producing a value of type B
- Σ is a pair (tag, Value)
- type A × A′ is a value of type (V, V′)
- type 1 is a 0-tuple
- computation of type FA produces a value of type A
- computation Π pops a tag i from operand stack, then is a computation of type B
- computation A → B pops a value of type A, then behaves as type B

https://www.cs.bham.ac.uk/ pbl/papers/tlca99.pdf

3.5.3 Normalization-by-evaluation

[Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

10]

NbE

- well investigated in [36] where there is a comprehensive of NbE techniques as applied to ML
- also in [36] there is a treatment of η reduction/expansion - READ
- [3] - NbE for dependent types including sums, renamings!
- [11] - Haskell simple NbE
- [19] - Tm, NfTm, NbTm; NfVal, NeVal, Val split, NbE well-written, normals, closures

Normalization-by-evaluation in the style of Abel. Reminiscent of partial evaluation for λ-calculus as we use "sticky" *neutral values* instead of values we currently do not have. Provably confluent, a viable normalization strategy.

As we use typed reduction rules, we do not need "subject reduction" algorithms(?)

Conversion checking = type (or expression) equivalence checking, includes evaluation (NbE = full comparison of normal forms), checking equivalence "as described in the previous section"

Describe motivation for NbE, the process, what is a neutral, eval/quote (3 paras)

3.5.4 Conversion checking

We will also need to compare two λ-terms to see if they are equal. There are two general notions of equality, intensional and extensional. By intensional we mean that they are written in the same way, whereas extensional equality means that they behave in the same way with regards to an observer. While in general programming languages, it is impractical to check whether two functions are equal, whether intensionally or extensionally, we will need this later when we are talking about type-checking: to compare two types expressed as λ-terms.

The usual notion of equality in λ-calculus is *α-equivalence of β-normal forms*, meaning that we first reduce both expressions as much as possible and then compare the symbols they are made of, ignoring differences in variable naming; this is formally expressed as variable permutation.

3.6 Elaboration

[20]

In the semantics in Figure **??** this is written using a different notation that we will use later: the expression $\Gamma \vdash e \Rightarrow t$ means "given the context Γ and expression e, infer its type t", whereas $\Gamma \vdash e \Leftarrow t$ means "given the context Γ, expression e and type t, the type t checks against the expression t".

Approach - infer, check + eval/quote used, global contexts (2 paras)

Metavariables and holes, sequential processing (1 para)

How do we do unification (2 paras + 1 figure)

- [24] - well-specified unification algorithm, pruning
- [1] - unifying sigma

Bidirectional typing

- [16] - elaboration, bi-di
- [13] - full rules, polarized??

Bidirectional typing (https://www.cl.cam.ac.uk/~nk480/bidir-survey.pdf) = now standard approach, combines type-checking and inference, simpler to implement even if inference is not required

Intro, motivation, list alternatives, pros and cons (2 para)

Sketch the process? Write out lambda + app rules? (half a page)

- elaboration uses metavariables, "holes" of unspecified type to solve
- unification instead of conversion checking - it solves metas
- metas are at the top-level, but to depend on local values, they are functions
- higher-order unification - to produce functions
- pattern unification, as higher-order is indecidable in general - distinct variables, not recursive
- spine is a list of arguments to apply
- neutrals - flexible is a meta with spine, rigid is local var with spine, flex is flex because it can be unblocked if the meta is solved can unblock
- meta forcing - brings is up to date with respect to the environment, up to WHNF
- meta solving should be destructive (sharing)
- renaming - maps variables from the spine to final de Bruijn level in the solution; is partial, requires depth of metacontext, and final context

3.7 User interface

The user interface we will use is a command-line one: a multi-purpose command that can start an interactive command-line session, execute a file, or pre-compile a source file into a Native Image.

From my research, JLine is the library of choice for interactive command-line applications in Java, so that is what I used for the REPL (Read-Eval-Print Loop). It is a rather easy-to-use library: for the most basic use case, we only need to provide a prompt string, and a callback. We can also add auto-completion, a parser to process REPL commands, custom keybindings, built-in pager or multiplexer, or even a scripting engine using Groovy.

Look at [18] for other interaction modes:

- golden tests (`:verbose` as a command, e.g.)
- :verbose = write out after every reduction
- :ski = express in terms of combinators
- interpreter + batch processor that reads REPL commands as well
- Jupyter kernel
- :types on, switches between untyped and simply-typed
- @@(expr) = print out as a proof (type derivation) using typing rules

State keeping - load, reload + querying the global state (1 para)

commands:

- `:l` create a NameTable
- `:r` recreate a NameTable
- `:t` inferVar, print unfolded
- `:nt` inferVar, print folded
- `:n` inferVar type, gQuote term, show
- `:e` print elaboration output including all metas

Example CLI session

3.8 Results

[10]

Quick look at everything that this toy can do (2-3 examples?)

Evaluation, simplification, elaboration with holes, unification using eqRefl

Error reporting

Chapter 4

Adding JIT compilation to Montuno: MontunoTruffle

4.1 Just-in-time compilation

- slow AST interpreter
- meta-tracing optimizing interpreter
  - PyPy
  - Records a path "trace" of interpreter execution
  - Programmer provides hints, but has no control over the paths traced (only some are)
  - unpredictable peak performance
- partial evaluation
  - old idea
  - offline and online partial evaluators
  - offline doesn't have deoptimization support
- Graal is "metacircular", it is a Java optimizer written in Java
- projections - partial evaluation
  - first = PE an interpreter with a program = executable
  - second = PE the specializer with an interpreter = compiler
  - third = PE the specializer with the specializer = compiler-maker

Mention the general JIT theory - classical, tracing, rewriting

Mention partial evaluation, Futamora projections (2 paras + 3 items + example)

Where is the partial evaluation in Graal (1 para)

Graal compiles "hot code" to machine code by partially evaluating it. Partial evaluation is TODO (Futamora). In particular, in dynamic languages it helps to eliminate dynamic dispatch / megamorphic call overhead.

4.2 GraalVM and the Truffle Framework

4.2.1 GraalVM

–Re-read, simplify (what is it, why we want to use it, what specifically do we use?)

GraalVM is an Oracle research project that was originally created as a replacement for the HotSpot virtual machine written in C++. It has since expanded to include other features novel to the Java world.

Cite Oracle

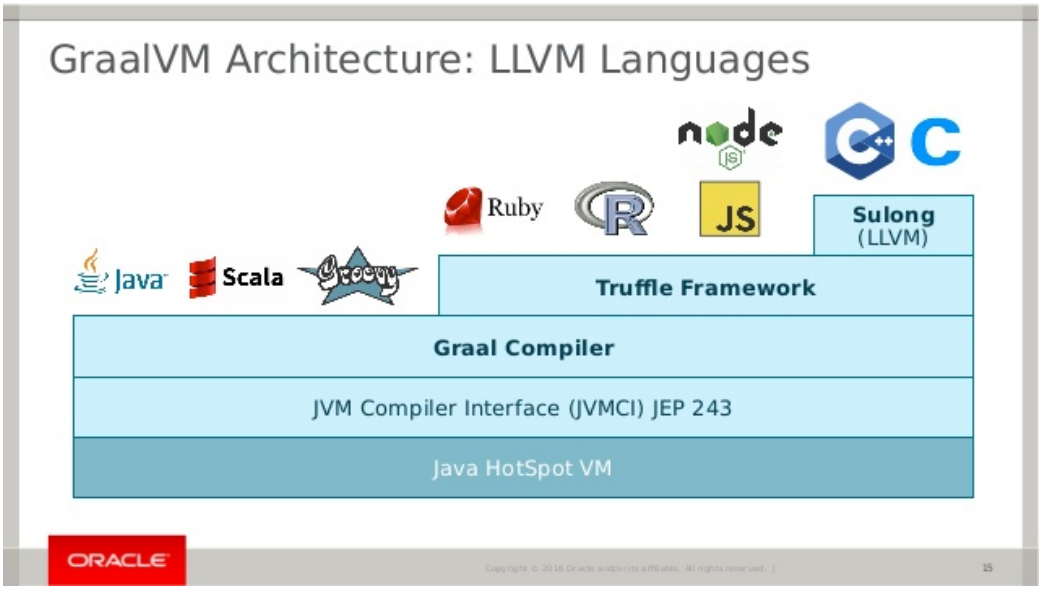

Figure 4.1: GraalVM and Truffle (source: oracle.com)

The project consists of several components which can be seen in 4.1, the main components being the following ones:

Graal is an optimizing just-in-time compiler based on partial evaluation. Graal uses the JVM Compiler Interface which allows the main JVM to offload compilation to external Java code. It can also use C1 (the old JVM JIT) which implies tiered compilation (...disabled with `-XX:-TieredCompilation`). When using the GraalVM, the only JVMCI-compatible compiler is Graal, so automatically used.

Simplify language

SubstrateVM is an alternative virtual machine that uses aggressive ahead-of-time compilation of Java bytecode into a standalone executables, a so-called /Native Image. The project aims to guarantee fast start-up times, relatively small binary files, and low memory footprint–as opposed to slow start-up times due to JIT compilation and large memory usage common to JVM-based languages.

Cite SubstrateVM

Truffle (and **Truffle DSL**) is a language implementation framework, a set of libraries that expose the internals of the Graal compiler to interpreter-based language implementations. It promises that its users only need to write an interpreter with a few framework-specific annotations in order to automatically gain:

- access to all optimizations available on the Java Virtual Machine

- debugger support
- multi-language (*polyglot*) support between any other Java-based or Truffle-based languages (currently JavaScript, Python, Ruby, R, C, C++, WebAssembly)
- the ability to gradually add optimizations like program graph rewriting, node specializations, or inline instruction caching

Images from

images from https://www.slideshare.net/jexp/polyglot-applications-with-graalvm

GraalVM is also intended to allow creating *polyglot applications* easily, applications that have their parts written in different languages. It is therefore easy to e.g. call R to create visualizations for the results of a Python program, or to call any Truffle language from Java.

This seems like a good middle ground between spending large amounts of time on an optimized compiler, and just specifying the semantics of a program in an interpreter that, however, will likely not run quickly.

While GraalVM/Truffle is open-source and released under GPL v2, an enterprise edition that claims large performance improvements is released commercially.

Graal-compiled code still runs on the original HotSpot VM (written in C++)

Graal is also a graph optimizer, which takes the data-flow and instruction-flow of the original program and is able to reorder them to improve performance. It does what the original HotSpot VM would do, optimize JVM bytecode by reordering, pre-compiling, or entirely rewriting instructions.

This graph is also extensible [12]

- Canonicization: constant folding, simplification
- Global value numbering: prevents same code from being executed multiple times
- Lock coarsening: simplifies `synchronized` calls
- Register allocation: data-flow equals the registers required, optimize
- Scheduling: instruction-flow implies instruction order

Graal by itself is just better HotSpot, but there are other technologies in the mix. SubstrateVM is an ahead-of-time compiler for Java which takes Java bytecode and compiles is into a single binary, including the Graal runtime, pre-compiling application code to greatly reduce warm up times (that are otherwise shared by all JIT compilers).

In the ideal world, what GraalVM can do with the code by itself would be enough, if that is not sufficient then we can add specializations, caches, custom typecasts, ... We can also hand-tune the code, adding our hand-generated bytecode into the mix.

4.2.2 Truffle

- trace-based optimization frameworks (RPython) vs partial evaluation frameworks (Truffle)
- JIT compilation for (suitably instrumented) interpreters
- Truffle provides a set of class and method annotations which reduce boilerplate and implementation effort. They drive a compile-time code generator

Introduce Truffle specifically, polyglot, graph (de)optimization (3 paras + code sample)

Tiered compilation: C1, C2, Graal replaces C2

Multiple uses of Graal - it can function as instead of the C2 (server) compiler or as a PE/AoT for SubstrateVM

Instruments, Chrome and other debugging support

Warm-up benchmark: http://macias.info/entry/201912201300_graal_aot.md

It really is fast enough - FastR, Python benchmarks (2 paras + graph)

4.3 Truffle language features

General features What does a typical Truffle language look like? (1 para)

"We can look at a piece of Ruby code as a graph" (1 para + graph)

Another visualization option: Seafoam

Introduce the canonical example - Literal and addition, trivial execute fn (1 para, 1 figure)

Truffle: framework for implementing high-performance interpreters, in particular for dynamic languages, and AST interpreters. Each node implements the semantics of the operation/language construct. Graph flow - execution flows downwards, results flow upwards.

Graph manipulations:

- replace() to a more specific variant
- adopt() a new child node
- GraalVM aggressively inlines stable method calls into efficient machine code
- [51] - Truffle boundaries
- [50] - deoptimization, on-stack replacement

Type specialization Type system

Basic case is type specialization - when an addition node only encounters integers, there is no need to generate machine code for floats, doubles, or operator overloads - only verified by fast checks. When these fail, the node is de-optimized, and eventually re-compiled again.

Inline caching for e.g. method lookups, virtual method calls are typically only ever invoked on a single class, which can be cached, and the dispatch node can be specialized, perhaps even inline the operation. (uninitialized/monomorphic/polymorphic/megamorphic = working with jumptables + guard conditions)

Specializations are general, though, and nodes can go be specialized on arbitrary conditions, using custom assumptions and *compilation final* values. In general, node states form a directed acyclic graph - "a node can ever become more general".

Using a graph visualizer, we can look at this process on a simple example commonly used to demonstrate this part of the Truffle framework: the + operation.

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

The Truffle framework is said to be a domain-specific language, which in this case means a library, a set of annotations, and a code generator. This code generator finds classes that inherit from the `Node` class and generates, among others, the logic behind switching specializations.

The program graph is formed from a tree of Truffle `Nodes` from which we derive our language-specific base class, `LangNode` in this case. We define two classes that inherit from this class, one representing integer literals, and one for the + operator.

The abstract method `execute` in `LangNode` is the evaluation of this node. It takes a `VirtualFrame`, which represents a stack frame, and its return value is also the return value of the node. In addition, methods starting with `execute` are special in Truffle. Truffle will pick the most appropriate one based on return type (with `Any` being the most general) and parameters.

In `IntLiteralNode` we directly override the method `execute`, as there is only one possible implementation. In `AddNode`, however, we keep the class abstract and do not implement `execute`, and instead rely on Truffle to generate the appropriate specialization logic.

Truffle will decide between the specializations based on parameter types, and on user-provided guards (we will see further). Fallback specialization matches in cases where no other one does. Names are irrelevant.

(Maybe show generated code?) Active an inactive specializations: can be multiple active, execute method is based on state first, and only then on type checks - smaller and possibly better optimized result. If no specialization matches, then fall through to `executeAndSpecialize` which invalidates any currently compiled using `CompilerDirectives.transferToInterpreterAndInvalidate` and sets state bits for newly activated specializations.

(@Specialization(Replaces=[""]))

How to run? Need to wrap in a `RootNode`, which represents executable things like methods, functions and programs. Then create a `CallTarget` using `Truffle.getRuntime().createCall` Truffle uses CallTargets to record, among others, how often a particular graph is called, and when to compile it. Also it creates a VirtualFrame for this call target out of the provided arguments.

IGV receives JIT compilation output - shows the Graal graphs produced during optimization. Compilation is only triggered after a certain threshold of calls, so we need to run a call target more than just once.

Graal-graph

Another option: a CountNode (public int counter; execute = counter++). Green == state, grey is floating (not flow dependent), blue lines represent data flow (data dependencies), red means control flow (order of operations)

Object Storage Model [20] - Polyglot + OSM intro

- [38] - URI language, object-oriented
- OSM (Object Storage Model) - Frame (~typed HashMap)
- VirtualFrame - virtual/optimizable stack frame, "a function's/program's scope"
- MaterializedFrame - VirtualFrame in a specific form, not optimizable, can be stored in a ValueType

FrameDescriptor - shape of a frame FrameSlot FrameSlotKind

VirtualFrame - can be optimized, reordered

MaterializedFrame - an explicit Java Object on the heap, created from a VirtualFrame by calling `frame.materialize()`

Object Storage Model is a common way to organize data with layouts - Objects, Shapes, Layouts.

TruffleRuby uses it for FFI to access Ruby objects directly as if they were C `structs`. DynSem uses these to model frames/scopes instead of Frames, as it is a meta-interpreter and uses Truffle Frames for its own data.

Dispatch dispatchNode, frames, argument passing, ExecutableNode, RootNode, CallTarget

- RootNode - can be made into a CallTarget. is at the root of a graph, “starting point of a function/program/builtin”, “callable AST”

Direct/IndirectCallNode

VirtualFrames can be eliminated altogether, which results in highly efficient code

Caching @Cached()

Polyglot ValueTypes, InteropLibrary

[20] - Polyglot + OSM intro

Structure Common components - Launcher, LanguageRegistration, Nodes, Values, REPL (5 items)

Engine, Context, TruffleLanguage, Instrument

```
N: unbounded
P: N for exclusive, 1 for shared context policy
L: number of installed languages
I: number of installed instruments

- 1 : Host VM Processs
 - N : Engine
   - N : Context
     - L : Language Context
   - P * L : TruffleLanguage
   - I : Instrument
     - 1 : TruffleInstrument
```

4.4 Functional Truffle languages

4.4.1 Criteria

Evaluate languages on:

- overall project structure and runtime flow
- global/local names and environment handling
- calling convention
- lazy evaluation
- closure implementation
- graph manipulation, TruffleBoundaries, specializations

Truffled PureScript Old project, but one of the only purely-functional Truffle languages.

Purescript is a derivative of Haskell, originally aimed at frontend development. Specific to Purescript is eager evaluation order, so the Truffle interpreter does not have to implement thunks/delayed evaluation.

Simple node system compared to other implementations:

- types are double and Closure (trivial wrapper around a RootCallTarget and a MaterializedFrame)
- VarExpr searches for a variable in all nested frames by string name
- Data objects are a HashMap
- ClosureNode materializes the entire current frame
- AppNode executes a closure, and calls the resulting function with a { frame, arg }
- CallRootNode copies its single argument to the frame
- IR codegen creates RootNodes for all top-level declarations, evaluates them, stores the result, saves them to a module Frame
- Abstraction == single-argument closure

Mumbler An implementation of a Lisp

FastR One of the larger Truffle languages

[47]

Replacement for GNU R, which was "made for statistics, not performance"

Faster without Fortran than with (no native FFI boundary, allows Graal to optimize through it)

Interop with Python, in particular - scipy + R plots

Node replacement for specializing nodes, or when an assumption gets invalidated and the node should be in a different state (AbsentFrameSlot, ReplacementDispatchNode, CallSpecialNode, GetMissingValueNode, FunctionLookup.

```
val ctx = Context.newBuilder("R").allowAllAccess(true).build();
ctx.eval("R", "sum").execute(arrayOf<Int>(1,2,3));
```

```
benchmark <- function(obj) {
    result <- 0L
    for (j in 1:100) {
       obj2 <- obj$objectFunction(obj)
       obj$intField <- as.integer(obj2$doubleField)
       for (i in 1:250) { result <- obj$intFunction(i, obj$intField) }
    }
    result
```

```
}
benchmark(.jnew("RJavaBench"))
```

Special features:

- Promises (call-by-need + eager promises)

Cadenza

- FrameBuilder - specialized MaterializedFrame
- Closure - rather convoluted-looking code

Generating function application looks like:

- TLam - creates Root, ClosureBody, captures to arr, arg/envPreamble
- Lam - creates Closure, BuilderFrame from all captures in frame
- Closure - is a ValueType, contains ClosureRootNode
- ClosureRootNode - creates a new VirtualFrame with subset of frame.arguments

Enso A very late addition to this list, this is a project that originally rejected Truffle (and dependent types in general, if I recall correctly) and used Haskell instead. However, the project Luna was renamed to Enso, and rebuilt from scratch using Truffle and Scala not long before my thesis deadline.

TruffleClojure Implemented in a Master's thesis [15]

DynSem

- [49] - OSM for frames/scopes

Chapter 5

Language implementation: MontunoTruffle

5.1 Introduction

Truffle is not primarily aimed at statically-typed languages or functional languages. Its most easily accessible benefits lie in speculative optimization of dynamically typed code and inline caches, where generic object-oriented code can be specialized to a specific value type. Statically-typed languages have a lot more information regarding the values that will flow through a function, and e.g. GHC has a specific *specialization* compiler pass.

However, there is a lot of overlap between the static optimizations done by e.g. GHC and runtime optimizations done by Graal. An example would be unfolding/inlining, where the compiler needs to make a single decision of whether to replace a call to a function with its definition – a decision that depends on the size of the definition, whether they are in the same module, and other heuristics [26]. A Truffle interpreter would be able to postpone the decision until execution time, when the definition could be inlined if the call happened enough times.

—mental model = graph through which values flow, values may contain graphs

Its execution model is a tree of nodes where each node has a single operation `execute` with multiple specializations. The elaboration/evaluation algorithm from the previous chapter, however, has several interleaved algorithms (infer, check, evaluate, quote) that we first need to graft on to the Truffle execution model.

There are also several features that we require that are not a natural fit for it, but where we can find inspiration in other Truffle languages. In particular, lazy evaluation (FastR promises), partial function application (Enso), ???

We also have several options with regard to the depth of embedding: The most natural fit for Truffle is term evaluation, where a term could be represented as a value-level Term, and a CallTarget that produces its value with regard to the current environment. We can also embed the bidirectional elaboration algorithm itself, as a mixture of infer/check nodes.

In dynamic interpreters that Truffle is aimed at, it is easy to think of the interpreter structure as "creating a graph through which values flow".

—???

There are several concerns here:

- algorithmic improvement is asymptotic – the better algorithm, the better we can optimize it
- Truffle's optimization is essentially only applicable to "hot code", code that runs many times, e.g. in a loop
- We need to freely switch between Term and Value representations using eval/quote

The representation is also quite different from the functional interpreter where we have used functions and data classes, as in Truffle, all values and operations need to be classes.

Specific changes:

- everything is a class, rewrite functions/operations as classes/nodes
- annotations everywhere
- function dispatch is totally different
- lazy values need to be different
- ???

5.2 Parser

ParsingRequest/InlineParsingRequest

Unfortunately, Truffle requires that there is no access to the interpreter state during parsing, which means that we need to perform elaboration inside of a `ProgramRootNode` itself, "during runtime" per se.

need to perform elaboration inside a programRootNode (not while parsing)

stmt;stmt;expr -> return a value

5.3 Representing values

As with the previous interpreter, we will start with function calls.

dispatch, invoke, call Nodes, argument schema (copying), ?

eta is TailCallException (2 para + example)

Passing arguments - the technical problem of copying arguments to a new stack frame in the course of calling a function.

Despite almost entirely re-using the Enso implementation of function calls, with the addition of implicit type parameters and without the feature of default argument values,

I will nonetheless keep my previous analysis of calling conventions in functional Truffle languages here, as it was an important part of designing an Truffle interpreter and I spent not-insignificant amounts of time on it.

I have discovered Enso only a short while before finishing my thesis, and had to incorporate the technologically-superior solution

Several parts of creating an AST for function calls:

- determining the position of arguments on the original stack - or evaluating and possibly forcing the arguments
- determining the argument's position on the stack frame of the function
- using this position in the process of inferring the new function call
- dispatch, invoke, call nodes???

Value types Data classes with call targets

...depends on what will work

Term and Val are ValueTypes that contain a callTarget - eval/quote(?)

We could use Objects/Shapes/Layouts for dependent sums or non-dependent named coproducts.

5.4 Representing environments and variables

We need to use arrays, Collections are not recommended

Frames and frame descriptors for local/global variable

References, indices, uninitialized references

5.5 Normalization

Evaluation order We need to defer computations as late as possible - unused values that will be eliminated (1 para)

CBPV concepts, thunks with CallTargets (3 paras, example)

Calling convention the need for the distinction - in languages with currying

the eval/apply paper is a recipe for a stack-based implementation of currying and helpful in our case when we need explicitly manage our stack via Frames as opposed to the interpreter where we relied on the host language for this functionality

known/unknown calls, partially/fully/over-saturated calls

[37]

push-enter - arguments are pushed onto the stack, the function then takes as many as it requires

eval-apply - the caller sees the arity of the function and then decides whether it is over-applied (evaluates the function and creates a continuation), appllied exactly (EVAL), or under-applied (creates a PAP, a closure-like value)

exactly describe the rules from eval-apply paper KNOWNCALL, EXACT, CALLK, PAP

5.6 Elaboration

5.7 User interface

REPL needs to be implemented as a `TruffleInstrument`, it needs to modify and otherwise interact with the language context.

5.8 Polyglot

Demonstrate calling Montuno from other languages

Demonstrate Montuno's eval construct

Demonstrate Montuno's FFI construct - requires projections/accessors

5.9 Implementation

How to connect this together?

evaluation phases - translate to Code, run typecheck, run eval vs glued, ???

How to edit the REPL

Language registration in mx/gu

5.10 Results

how fast are we now? Are there any space/time leaks?

show program graphs: id, const, const id; optimized graphs

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))

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

- [8] - potential optimizations, LLVM impl, closures
- [21] - Coq experience, a few reasons, comparison
- [22] - a lot of reasons in Coq
- [14] - CSE

Ruby uses threads, can we? Automatic parallelism

- [42] - concurrency & parallelism in GHC evaluation
- [25] - CAFs? Lazy evaluation?

Think about the fast vs slow path!

- [53] - 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 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.

[29]

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

6.4 Splitting

type specializations/dict passing

6.5 Function dispatch

lambda merging

eta expansion

6.6 Caching and sharing

Sharing computation and common values

Multiple references to the same object

let-floating

inlinable functions

6.7 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.

6.8 Profiling

Now then, what tools to use to find the problems

6.8.1 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.

6.8.2 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.

Chapter 7

Evaluation

We want to evaluate a few programs of equivalent functionality, as evaluated by elaboration, type-checking, and simplification in a number of dependently-typed languages.

We also want to evaluate the performance of general β-normalization which only requires a functional language–this is secondary, however.

We are mainly interested in asymptotic behaviors and not on constant factors. Just-in-time compiled languages especially suffer from long warm-up times, which means that common evaluation of "repeatedly running a command" will not perform well.

7.1 Subjects

Subjects for elaboration: Agda, Idris, Coq, GHC. Also cooltt, smalltt, redtt, Lean, ?

Subjects for normalization: as above, but also Cadenza (STLC), Clojure, GHC, Scala, OCaml, ML, Eta, Frege, ?

Also subjects: Montuno, MontunoTruffle, and possibly other optimized versions.

7.2 Workload

- Nats - large type elaboration, call-by-need test
- Nats - type-level calculation
- Nats - value-level calculation
- Nats - equality/forcing
- Functions - nested function elaboration, implicits
- Functions - embedded STLC?
- pairs - large type elaboration, call-by-need test
- pairs - nested accessors

- typical programs - see typical Agda
- computation-heavy tests (numerics,)
- memory-heavy tests (id id id...)
- nofib suite? (typecheck, circsim, infer, anna)
- hanoi, sort an array using trees
- use FFI in benchmarks - externalize a FFT?

Mention sources (this is from smalltt ^^)

Brief descriptions, what does each one evaluate/stress?

7.3 Methodology

We need to use in-language support, if available. We want to avoid measuring interpreter start-up, program parsing time, and other confounders.

To measure: memory usage (curve), compilation speed (in a type-heavy test), evaluation speed (in a compute-heavy test)

hyperfine to benchmark - measures speed (what about `prof`?) memory profile from stderr output

Krun benchmark runner + its $\text{warmup}_{\text{stats}}$ functionality for statistical analysis of steady states, number of iterations it took to stabilize.

memory profile from GHC's RTS for agda/idris/smalltt (+RTS -p) (what about coq? - https://github.com/coq/coq/blob/master/dev/doc/profiling.txt) Graal's default memory profiler

Mention specific parameters (X iterations, machine specs, ?)

7.4 Results

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

7.5 Discussion

Size of codebase

Effort required

Effect produced

Is this road viable?

7.6 Next work

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] Altenkirch, T., Danielsson, N. A., Löh, A. and Oury, N. ΠΣ: Dependent types without the sugar. In: Springer. *International Symposium on Functional and Logic Programming*. 2010, p. 40–55.

[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] Altenkirch, T. and Oury, N. ΠΣ*: A Core Language for Dependently Typed Programming*. Citeseer, 2008.

[5] 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.

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

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

[8] Blaguszewski, M. *Implementing and Optimizing a Simple, Dependently-Typed Language*. Master's thesis.

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

[10] 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.

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

[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] Feichtinger, T. *TruffleClojure: A self-optimizing AST-Interpreter for Clojure/submitted by: Thomas Feichtinger*. 2015. Dissertation. Linz.

[16] 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.

[17] 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.

[18] Garcia, M. R. *Category Theory and Lambda Calculus*. Master's thesis.

[19] 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.

[20] 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.

[21] 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.

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

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

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

[25] 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.

[26] Jones, S. P. and Marlow, S. Secrets of the glasgow haskell compiler inliner. *Journal of Functional Programming*. 2002, vol. 12.

[27] Juan, V. L. and Seminar, L. *Practical Unification for Dependent Type Checking*. Licentiate Thesis (draft). Department of Computer Science and Engineering ..., 2020.

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

[29] 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.

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

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

[32] 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.

[33] 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.

[34] 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.

[35] Levy, P. B. Call-by-push-value: A subsuming paradigm. In: Springer. *International Conference on Typed Lambda Calculi and Applications*. 1999, p. 228–243.

[36] Lindley, S. *Normalisation by evaluation in the compilation of typed functional programming languages*. Master's thesis.

[37] Marlow, S. and Jones, S. P. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In: *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming - ICFP '04*. 2004, p. nil. Available at: https://doi.org/10.1145/1016850.1016856.

[38] Mouton, Y., Mens, K. and Laurent, N. *"Incremental compiler development*. Master's thesis.

[39] 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.

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

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

[42] 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.

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

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

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

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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=COMMAND? term                         #Expr
     ;
term
     : 'let' id=binder ':' type=term '=' defn=term 'in' body=term #Let
     | LAMBDA (rands+=lamBind)* '.' body=term                     #Lam
     | (spine+=piBind)+ ARROW body=term                           #Pi
     | rator=atom (rands+=arg)* (ARROW body=term)?                #App
     ;
arg
     : '{' (IDENT '=')? term '}' #ArgImpl
     | atom                      #ArgExpl
     | '!'                       #ArgStop
     ;
piBind
     : '(' (ids+=binder)+ ':' type=term ')'     #PiExpl
     | '{' (ids+=binder)+ (':' type=term)? '}' #PiImpl
     ;
lamBind
     : binder                   #LamExpl
     | '{' binder '}'           #LamImpl
     | '{' IDENT '=' binder '}' #LamName
     ;
atom
     : '(' term ')'             #Rec
     | IDENT                    #Var
     | '_'                      #Hole
```

```
    | '*'                       #Star
    | NAT                       #Nat
    | '[' IDENT '|' FOREIGN? '|' term ']' #Foreign
    ;
binder
    : IDENT #Bind
    | '_' #Irrel
    ;

IDENT : [a-zA-Z] [a-zA-Z0-9']*;
NAT : [0-9]+;
COMMAND : '%elaborate' | '%normalize' | '%parse';

END : (SEMICOLON | NEWLINE) NEWLINE*;
fragment SEMICOLON : ';';
fragment NEWLINE : '\r'? '\n' | '\r';
SPACES : [ \t] -> skip;
LINE_COMMENT : '--' (~[\r\n])* -> skip;
BLOCK_COMMENT : '{-'~[#] .*? '-}' -> skip;
LAMBDA : '\\' | 'λ';
ARROW : '->' | '→';
FOREIGN : [^|]+;
```

B.2 Semantics