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

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

Starting from basic λ-calculus theory and building up to the systems of the lambda cube, we specify the syntax and semantics of a small language that I refer to as Montuno (Chapter~2). We go through the principles of λ-calculus evaluation, type checking and elaboration, implement an interpreter for Montuno in a functional style (Chapter 3). In the second part of the thesis, we evaluate the capabilities offered by Truffle and the peculiarities of Truffle languages (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 $(\square, \square)$, which represents higher-order types, also called type operators; terms depending on types $(\square, \star)$, called *polymorphism*; and terms depending on types $(\star, \square)$, which represents *dependent types*, and is the reason for the name of dependently-typed languages.

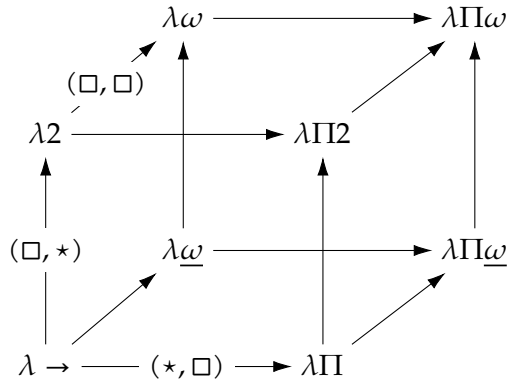

Figure 2.8: Barendregt cube (also λ-cube)

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

$$t : T : \star : \Box.$$

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 $\Box$. The symbols $\star$ and $\Box$ 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:

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

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

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

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

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

$\lambda\Pi$-calculus Types depending on terms, the dependency $(\star, \Box)$, 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).

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

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 $S = \{\star, \Box\}$ we can define relations A and R where, for example, $A = \{(\star, \Box)\}$ is translated to the axiom $\vdash \star : \Box$ by the rule **Start**, and $R = \{(\star, \Box)\}$[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.

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

Interpreter directives Another type of top-level expressions is a pragma, a direct command to the compiler. We will use these when evaluating the time it takes to normalize or elaborate an expression, or when enabling or disabling the use of "wired-in" types, e.g. to compare the performance impact of using a Church encoding of numbers versus a natural type that uses hardware integers. We will once again use the syntax of Agda:

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

Polyglot Lastly, 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:

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

The syntax and semantics presented here altogether comprise a working programming language. A complete listing of the semantics is included in Appendix B. The syntax, written using the notation of the ANTLR parser generator is in 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)