
**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

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

# Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

# Keywords

klíčová slova v anglickém jazyce, oddělená čárkami.

# Klíčová slova

klíčová slova v českém jazyce, oddělená čárkami.

# 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

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

# Just-in-Time Compilation of the Dependently-typed Lambda Calculus

## Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. X

The supplementary information was provided by Mr. Y

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

.....
Jakub Zárybnický
February 7, 2021

## Acknowledgements

Here it is possible to express thanks to the supervisor and to the people which provided professional help (external submitter, consultant, etc.).

# Contents


**1 Introduction** **4**

**2 Partial evaluation/JIT/Futamura** **6**

**3 JIT principles** **7**

**4 GraalVM/Truffle** **8**

**5 Dependently-Typed Lambda Calculus** **10**

**6 LambdaPi specification** **12**

**7 LambdaPi Interpreter** **13**

**8 Truffle-based compiler** **14**

**9 LLVM-based compiler** **15**

**10 Benchmarks** **16**

**11 Evaluation** **17**

**Bibliography** **18**

**Appendices** **19**

**A Contents of the attached data storage** **20**

# List of Figures

1.1 Methods of program execution . . . . . 4

4.1 GraalVM and Truffle (source: oracle.com) . . . . . 9

5.1 Untyped lambda calculus . . . . . 10
5.2 Simply typed lambda calculus . . . . . 10
5.3 Dependently typed lambda calculus . . . . . 11

# List of Listings

1 The constant function in LambdaPi . . . . . 13

# Chapter 1

# Introduction

When creating small experimental or research languages, writing a compiler may be too much effort for the expected gain. On the other hand an interpreter is usually not as performant as its creators may require for more computationally intensive tasks.

There is a potential third way, proposed by Yoshihiko Futamura in the 1970s, called the Futamura projection (or partial program evaluation), wherein an interpreter is specialized in conjunction with the source code of a program, yielding an executable. Some parts of the interpreter may be specialized, some optimized, some left off entirely. Depending on the quality of the specializer, the gains may be several orders of magnitude.

The goal of my thesis is to evaluate whether the GraalVM/Truffle platform is suitable enough to act as a specializer for functional languages, in particular for the dependently-typed lambda calculus. To illustrate in Figure 1.1, the question is whether the path {Native Image→Result} is fast enough compared to the path *Executable→Result*.

Truffle has already been used rather successfully for the (mostly) imperative languages Ruby, Python, R, Java, and WebAssembly, but (purely-)functional languages differ in their evaluation model and in particular the required allocation throughput, so it is still an open question whether GraalVM is a good enough fit.

The desired outcome—at least, of the first part of my thesis—is a set of implementations, and a set of benchmarks demonstrating a positive or a negative result. If the result is positive, there are many potential follow-up tasks: implementing a different, more complex language, maybe a language to be interpreted into the dependently-typed lambda calculus to attempt the approach implemented in the *Collapsing Tower of Interpreters* [1], or ex-

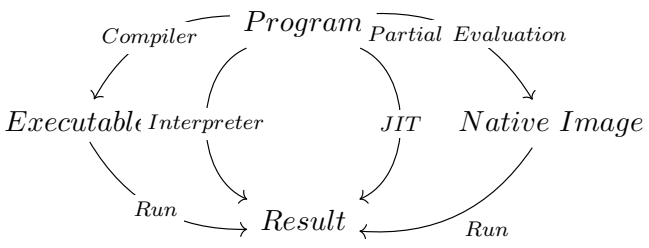


Figure 1.1: Methods of program execution

perimenting with different runtime models - all depending on the results of this preliminary proof of concept.

In the best case, the JIT-compiled program would be as close in performance to a program processed by a hand-crafted compiler as possible (not including JIT warm-up), and I would spend the second half of my thesis on different topics (like provably-correct program transformations) instead of hand-optimizing the primitive operations - I should find out which it is going to be as soon in the second term as possible.

As far as I am aware, there are no other native just-in-time compiled implementations of the dependently-typed lambda calculus, with the exception of the preliminary investigations done by the originator of this idea [3], although there are a few projects implementing a lambda calculus directly to the Java Virtual Machine byte code..

# Chapter 2

# Partial evaluation/JIT/Futamura

# Chapter 3

# JIT principles

# Chapter 4

# GraalVM/Truffle

**GraalVM** is a just-in-time optimizing compiler for the Java bytecode. **Truffle** is a set of libraries that expose the internals of the GraalVM compiler, intended for easy implementation of other languages. So far JavaScript, Python, Ruby, R, and WebAssembly have Truffle implementations, and therefore can run on the JVM.

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.

There is also the option to compile a *Native Image* to eliminate most program start-up costs associated with a just-in-time compiler, pre-compiling the program partially (ahead-of-time).

From the point of view of a programmer, Truffle makes it possible to write an interpreter, and then slowly add optimizations like program graph rewriting, node specializations, inline instruction caching or others. 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.

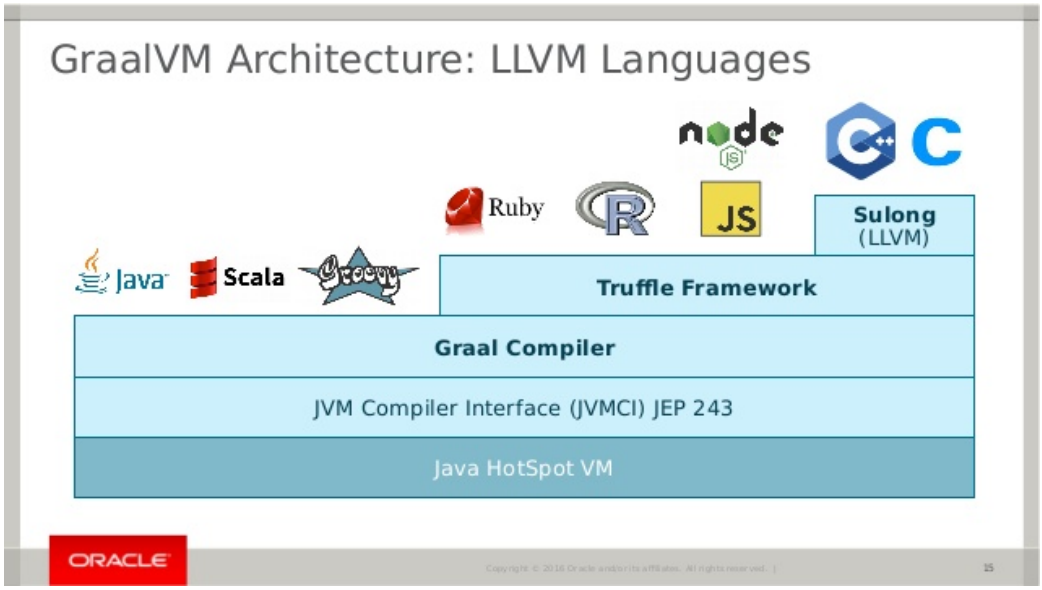


Figure 4.1: GraalVM and Truffle (source: oracle.com)

# Chapter 5

# Dependently-Typed Lambda Calculus

The untyped lambda calculus is introduced in the intermediate programming languages class but I haven't yet encountered its simply- or dependently-typed variants during my studies.

The untyped lambda calculus is a simple language consisting of just three kinds of forms: variables, function application, and abstraction.

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \lambda x.e & \text{abstraction}
\end{array}
$$

Figure 5.1: Untyped lambda calculus

The simply-typed lambda calculus adds a fourth kind of a term, type annotation, and its type language:

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & x : \tau & \text{annotation} \\
 & & & \\
\tau & ::= & \alpha & \text{base type} \\
 & | & \tau \to \tau' & \text{composite type}
\end{array}
$$

Figure 5.2: Simply typed lambda calculus

The dependently typed lambda calculus merges these two languages together, simplifying the grammar.

As I was already familiar with the use of dependent types in general programming e.g. in Agda or Idris, I took this opportunity to investigate the theoretical basis of type systems - type systems in general, as used in general programming languages, their various limitations—like the need to extend System F (System FC) as used in Haskell to support

$$
\begin{array}{rcll}
e & ::= & x & \text{variable} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \lambda x.e & \text{abstraction} \\
 & | & x : \tau & \text{annotation} \\
 & | & * & \text{the type of types} \\
 & | & \forall x : \rho.\rho' & \text{dependent function space}
\end{array}
$$

Figure 5.3: Dependently typed lambda calculus

fully dependent types [2], the lambda cube, the expressive power of different kinds of type systems, and where they are used.

# Chapter 6

# LambdaPi specification

# Chapter 7

# LambdaPi Interpreter

I have implemented a dependently typed lambda calculus called LambdaPi based on the prior work *A tutorial implementation of a dependently typed lambda calculus* [4]. The parser and interpreter are written in Kotlin, where I will also need to write the JIT implementation. This is a pure interpreter that will serve as a baseline for future benchmarks.

```
let const = (\ a b x y -> x) :: forall (a :: *) (b :: *) . a -> b -> a
```

Listing 1: The constant function in LambdaPi

# Chapter 8

# Truffle-based compiler

# Chapter 9

# LLVM-based compiler

# Chapter 10

# Benchmarks

# Chapter 11

# Evaluation

# Bibliography


[1] Amin, N. and Rompf, T. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages*. ACM New York, NY, USA. 2017, vol. 2, POPL, p. 1–33.

[2] Eisenberg, R. A. *Dependent types in haskell: Theory and practice*. University of Pennsylvania, 2016.

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

[4] Löh, A., McBride, C. and Swierstra, W. A tutorial implementation of a dependently typed lambda calculus. *Fundamenta informaticae*. IOS Press. 2010, vol. 102, no. 2, p. 177–207.

# Appendices

# Appendix A

# Contents of the attached data storage

...