https://www.fit.vut.cz

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

Jakub Zárybnický

Abstract
When building a programming language, the choice is often between writing a compiler or an interpreter, a compromise between speed and ease of implementation respectively. Just-in-time compilation offers a compromise, interpretation combined with gradual optimization during program runtime. The goal of my goal is to investigate whether just-in-time compilation offers the same advantages in the dependently-typed lambda calculus just as it does in imperative languages. The implementation environment is the Java Virtual Machine, and in particular the GraalVM runtime and the Truffle language implementation framework.
In the scope of this term project, I have investigated the relevant topics of JIT compilation, dependent types, and lambda calculi. I have also created a reference implementation of the dependently-typed lambda calculus LambdaPi based on prior work, and started two other implementations—one a LLVM-based compiler, and a Truffle-based JIT interpreter/compiler.
These three implementations—together with a set of benchmarks—will form the basis of my thesis and after omtimizations, will either prove or disprove the usefulness of JIT compilation for the dependently-typed lambda calculus.

xzaryb00@stud.fit.vut.cz, *Faculty of Information Technology, Brno University of Technology*

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, the question is whether the path {Native Image→Result} is fast enough compared to the path *Executable→Result*.

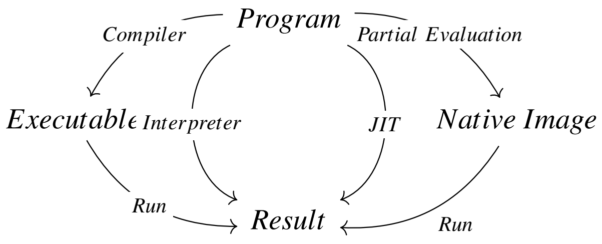

Figure 1. Methods of program execution

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 experimenting 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..

2. Assignment

The assignment for the first term was given as follows:

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.

2.1 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 2. Untyped lambda calculus

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

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

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

$$\begin{array}{rcll} \tau & ::= & \alpha & \text{base type} \\ & | & \tau \to \tau' & \text{composite type} \end{array}$$

Figure 3. Simply typed lambda calculus

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

Figure 4. Dependently typed lambda calculus

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

2.2 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.

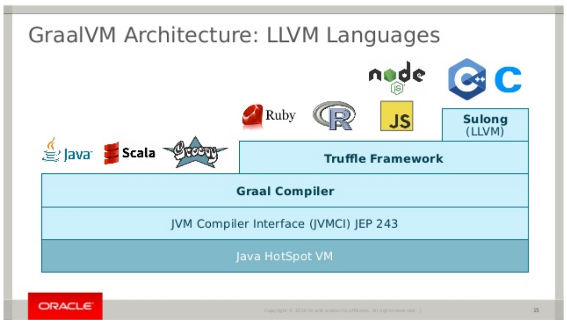

Figure 5. GraalVM and Truffle (source: oracle.com)

2.3 LambdaPi implementation

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

I have also translated parts of the language to Truffle, starting a second language implementation that will become the baseline JIT implementation, as well as preparing the groundwork for a third implementation, a compiler based on the LLVM.

Push/enter and eval/apply, the last items of the first task of my assignment, are relevant here, they are the two options for expressing partially-applied function (PAP) objects on the heap. While they are mostly equivalent in performance when applied to compilers [5], they differ in how difficult their implementation is, and their performance as part of Truffle will likely not be equal. A part of my work next term will be figuring out how well they map to the execution model of Truffle.

3. Conclusions

The next step is finalizing the basic Truffle implementation, applying the specializations as recommended by the Truffle documentation, and investigating other possible optimizations, especially pertaining to trampolining and specializing recursive functions.

There are a few different functional languages that compile directly to the JVM bytecode, and these may serve as inspiration as to how the JVM limits them—especially given that the JVM does not directly support tail calls.

After this, creating a set of benchmarks written in LambdaPi is the next step. Any further work will depend on the results of these benchmarks, after seeing how the performance of the individual implementations differs.

References

[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.

[5] MARLOW, S. and JONES, S. P. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *ACM SIGPLAN Notices*. ACM New York, NY, USA. 2004, vol. 39, no. 9, p. 4–15.