NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Computer and Information Science

TDT4501 — Specialization Project

Implementing Haskell in RPython

Author:
Even Wiik Thomassen

Supervisor:
Dr. Magnus Lie Hetland

June 8, 2012

Abstract

RPython has been used to implement VMs for many different programming languages, but not for any which are purely functional or lazy. RPython VMs are written in a high-level language that greatly simplifies development, and provide great performance with its meta-tracing JIT. We have created PyHaskell, a VM for the Haskell language, which on the Fibonacci benchmark is only 3.75 times slower than GHC. PyHaskell allows us to show that RPython is suitable for purely functional and lazy programming languages, and to investigate if a statically compiled language such as Haskell can benefit from JIT techniques.

CONTENTS

LIST OF TABLES

LIST OF LISTINGS

ACRONYMS

APC Advanced PHP Cache. 13

AST abstract syntax tree. 23, 30

CLR Common Language Runtime. 5–7, 12, 14, 15

FFI foreign function interface. 32

GADT generalized algebraic data types. 4, 9

GHC the Glasgow Haskell Compiler. 4, 5, 9, 11, 19–21, 25, 27, 29–34

GHCi GHC's interactive environment. 25, 29, 30

JIT just-in-time compilation. 1, 2, 6–8, 11–16, 18, 20, 22, 24–28, 30, 31, 33, 34

JSON JavaScript Object Notation. 11, 19, 21, 25, 30, 31

JVM Java Virtual Machine. 5–7, 12, 14, 15

KLoC thousand lines of code. 15–17

LINQ Language Integrated Query. 3

PHP PHP: Hypertext Preprocessor. 13, 14, 18, 31

STG Spineless Tagless G-machine. 4

VM virtual machine. 1, 2, 5–9, 11–20, 22–25, 27, 29–34, 41

CHAPTER 1

INTRODUCTION

Both dynamic programming languages as well as functional languages are gaining popularity today. On the functional front the Haskell language is home to language research, while on the dynamic front one very interesting development is the PyPy project that have created an environment for creating dynamic virtual machines (VMs). While the PyPy project has proven its relevance with its significantly faster Python interpreter, the dynamic VM environment behind the interpreter is less known and proven (this environment is henceforth called RPython). RPython has been used successfully to implement a wide range of VMs covering many programming paradigms, but lazy and purely functional are missing. We plan to correct this situation by implementing a Haskell VM with RPython.

The objective of this paper is to contribute work for the Haskell-Python project. The project has two objectives:

- Show that RPython is suitable for purely functional and lazy languages, e.g. the Haskell language.
- Show that Haskell may benefit from just-in-time compilation (JIT) techniques. While languages such as Haskell are heavily optimized at compile-time, more information is available at run-time that a JIT can exploit.

The scope of the project prevents this paper from answering these two objectives, and instead the paper serves as a basis for further research. The goal of the paper is twofold. First, to investigate other RPython VMs to find relevant lessons for the Haskell-Python VM. Second, to advance the VM to a state where it can be translated to C to be able to use the RPython meta-tracing JIT, which required converting the codebase fully to RPython. The second part of the goal depend on the first part, as most RPython doc-

umentation can be found in the papers describing VMs implemented with RPython. The contributions of this paper are:

- A literature review of VMs implemented with RPython.
- A summary of benefits from implementing VMs with RPython.
- Lessons from other RPython VMs to use in the Haskell-Python project.
- Significant work and progress on the Haskell-Python project, which include converting the codebase fully to the RPython language.

We start by explaining important concepts and terms (chapter 2, page 3), such as Core, PyPy, RPython, and JIT. The main contribution of this paper is the literature review and summary of VMs implemented with RPython (chapter 3, page 9). This paper is the first to present a literature review on this topic. We describe the technical work we have done in the context of the Haskell-Python project (chapter 4, page 19), which include removing Z-encoded names; support for data constructors; special handling of primitive type aliases; and finally converting the codebase into RPython. Furthermore we present results from our work (chapter 5, page 27), which include an overview of supported Haskell language features and results of micro benchmarks. The Haskell VM is not yet able to handle the full Haskell language, mainly because of major problems with the pipeline, which we discuss (chapter 6, page 30), including possible solutions to these problems. Finally we conclude the paper and list further work required to complete the project (chapter 7, page 34).

CHAPTER 2
CONCEPTS AND TERMS

This chapter tries to give an explanation and description of important concepts and terms used in this paper.

2.1 Haskell language

Towards the end of 1980s more than a dozen similar (non-strict, purely functional) programming languages had been created. During a meeting at the conference on Functional Programming Languages and Computer Architecture it was decided that a committee should be formed to design a new standardized common language for this class of languages [19]. The committee designed a new language named Haskell, which is a general purpose, purely functional, declarative, high-level programming language. Haskell is strongly typed with support for type inference, therefor type annotations are rarely needed. Other features of the language include lazy evaluation, pattern matching, list comprehensions, type classes and monads [16, 22].

Haskell was designed to be "suitable for teaching, research, and applications, including building large systems. (...) usable as a basis for further language research" [19]. As Haskell started to become popular it evolved quickly, which was problematic for teaching and application that require stability. The committee therefor named an instance of the language "Haskell 98", a stable version of the language, which implementers committed to support indefinitely. Afterward the committee disbanded to encourage language innovation and experimentation. In 2005 design of Haskell´ was started, to succeed Haskell 98 and to cover heavily used extensions [19]. Today the current stable version is Haskell 2010 [15].

Hudak et al. [19] point out that many features in C# were pioneered by Haskell, such as polymorphic types and Language Integrated Query (LINQ) which was directly inspired by monad comprehensions. List comprehensions

and array comprehensions in JavaScript are both inspired by Haskell's list comprehensions [19]. The biggest contributions from Haskell might be type classes and monads. A study covering the 2005-2006 academic year showed that Haskell was used in 95 university courses, most of which covered functional and/or declarative programming and programming languages [19]. Haskell's relevance and importance come from its use in teaching and its impact in programming language research.

2.2 Glasgow Haskell Compiler

The Glasgow Haskell Compiler (GHC) was started in 1989 at the University of Glasgow, and is probably the most fully featured Haskell compiler today. It was designed as a complete implementation of Haskell in Haskell [19]. Today GHC implements the latest version of the Haskell language, known as Haskell 2010[1] [15, 22]. GHC can be divided into three distinct parts:

- The compiler itself.
- The boot libraries that the compiler depend on.
- The runtime system. Library of C code that handles running the compiled Haskell code, such as garbage collection, threads, and exceptions. [22]

The compiler contains a pipeline for converting Haskell source code into executable machine code. The pipeline starts with the GHC front-end that performs parsing of source code, resolves identifiers into fully-qualified names, type checking and desugaring into an intermediate language called Core. The pipeline then performs a series of optimizations and simplifications on the Core representation. Finally the GHC back-end translates Core into another intermediate representation called STG (Spineless Tagless G-machine [25]), and then performs code generation into either C or native code [22, 33].

2.3 Core language

The Core language, GHC's intermediate representation, was initially based on lambda calculus. It was upgraded to a polymorphic lambda calculus, System F_ω, to be able to decorate it with types [19]. Core was further extended to System F_C, to support type equality constraints and safe coercions [32, 33, 36]. System F_C provide simple support for `kinds`, which are simply the type of a type. Core's most recent upgrade, System $F_C^\uparrow$, has more complex `kinds` that provide better support for `type families` and `generalized algebraic data types (GADT)` [36, 37].

[1] Haskell 2010 report [15]: http://www.haskell.org/onlinereport/haskell2010/

While Haskell is implicitly typed, Core is explicitly typed [36]. Core also differ from Haskell by being tiny. But while Core is small, it is extremely expressive [22].

The parts of GHC we use in this project, and how they are used, are described further in section 3.1 on page 9.

2.4 Python language

Python is a high-level, dynamic programming language that supports several programming paradigms, such as imperative, object-oriented and functional [5, 24]. It is highly regarded for its simplicity and ease of use [17].

Python was created by Guido van Rossum in the late 1980s [24]. Now, over 20 years later, its development is guided by the Python Software Foundation[2].

The original Python implementation was written in C, and is known as CPython[3] [24]. Since its creation, several alternative implementations of the Python language have been created. CPython is considered the official or standard Python implementation [30], and acts as a reference of the Python language.

Other well known Python implementations are: Jython[4], which is written in Java and runs on Java Virtual Machine (JVM); IronPython[5], which is written in C# and runs on Common Language Runtime (CLR); PyPy, a Python interpreter implemented in Python itself [24].

2.5 PyPy project

The PyPy project consist of two major components: *a*) the RPython translation toolchain; and *b*) PyPy, the Python interpreter [24]. The PyPy Python interpreter is built with the RPython translation toolchain. These two components map directly to the two goals of the PyPy project:

- Be an environment for implementing complex dynamic languages that support multiple platforms [7, 27]; and
- provide a faster Python implementation [5, 14].

Traditionally one must implement one VM for each platform one wish to support. By implementing VMs in a high level language PyPy can support several very different platforms with one implementation [27]. High level languages keep the implementation free of low-level details such as object

[2]Python Software Foundation: http://www.python.org/psf/
[3]CPython homepage: http://www.python.org/
[4]Jython homepage: http://www.jython.org/
[5]IronPython homepage: http://ironpython.net/

layout, threading model, and memory management [7]. This achieves the first goal. The disadvantage of this is speed, and to achieve the second goal PyPy has developed a meta-tracing JIT that is a part of the RPython translation toolchain [5, 9, 27].

The idea for PyPy surfaced in late 2002 on a Python mailing-list, and the project started with a week long meeting in February 2002. The project received EU-funding of 1.3 million euro, from first of December 2004 until November 2006. The funding allowed PyPy to arrange 14 sprints during these two years, and gave the project a rapid progress [14].

2.6 RPython language

PyPy is implemented in a restricted subset of the Python language, called RPython. Other VM implementations that want to use the RPython environment must also be written in RPython. The RPython language is selected in a way that makes it possible to do type inference on it [9]. The RPython language is not formally defined, but considered informally as any Python code that the RPython translation toolchain can handle.

The restrictions imposed by RPython together with type inference makes it possible to translate RPython programs directly to low-level languages like C [9]. Major restrictions are:

- Variables need to be type consistent, for example a variable cannot hold an integer and then later a string [24].
- Types of all variables in the code must be inferable [12].
- Functions cannot be created at runtime [24].
- Bindings in classes and global namespaces are assumed to be constant [27].
- Runtime reflection is not supported [12].

Despite these restrictions, RPython is a high-level language that supports: garbage collection; exceptions; single inheritance[6]; classes with virtual functions; first class functions and class values; runtime isinstance and type checks; and good built-in data-structures [6, 8, 9, 12].

2.7 RPython translation toolchain

One goal of the RPython translation toolchain is to compile RPython programs to various environments, such as C, JVM, and CLR [7, 27]. Another goal is to automatically create JITing VMs through meta-tracing [5].

[6]RPython support explicitly declared mixins that offer many of the advantages of multiple inheritance [2, 6].

The overall architecture of the toolchain starts at the high-level RPython source of the VM, and performs stepwise translations steps until it reaches low level code of the target platform. Each level has a corresponding type system and uses a generic type interference engine, and each level adds support for features that were assumed primitive by the previous level [27].

The translator toolchain starts with building flow-graphs from RPython source code. These flow-graphs consist of linked blocks where each block has input arguments and a list of operations. Next is the annotation phase, where type information is assigned to the arguments and result of each operation [24]. The annotation phase basically performs type inference on the whole program.

The next phase is RTyping, which uses type information to expand high-level flow-graph operations into low-level operations. After RTyping several optimizations are performed. Traditional optimizations such as constant folding, dead code removal and more complex ones such as function inlining and malloc removal [24].

The final phase is the back-end, which generates source code from flow-graphs. The C back-end emits C code, but must first add explicit garbage collection and exception handling [24]. The final step of the back-end compiles the generated C code. There is an alternative object-oriented back-end for generating code that runs on JVM and CLR.

2.8 Meta-tracing just-in-time compiler

The objective of the JIT is to improve the speed of the language, by compiling frequently used code-paths into assembly at runtime [24]. For a dynamic language the first goal of the JIT is to remove overhead from the interpreter, for example bytecode dispatch and interpreter's data structures. The second goal is to remove the overhead from boxing primitive types [10].

A popular approach to JIT is a tracing JIT that works by observing the running program to detect commonly executed concrete paths [9, 10]. Detected paths are called a trace, and contains the history of operations executed. A trace is first optimized with well known compiler optimizations, then turned into machine code. As a trace is linear, many optimizers are easy to write and generating machine code for it is straight forward [3]. As a trace is a path of the code, any branching along that path is protected with guards. If a different path is used, a guard will trigger and the interpreter continues. If a guard fails often, the tracing JIT will start a new trace from the failed guard [10].

To be able to reuse the tracer on other RPython VMs, RPython contains a meta-tracing JIT that traces the execution of the interpreter instead of the user program running on top of the interpreter. RPython's meta-tracing JIT can almost automatically create a JIT for VMs implemented

with RPython [5, 7]. Two hints in the source code of the VM is required by the meta-tracer: `merge_point` and `can_enter_jit`. The latter hint specifies where in the interpreter a loop starts, and the former hint says where it is safe to return to the interpreter from the JIT [24]. There are a growing list of other hints one can provide to help improve the performance of the JIT, but they are optional.

CHAPTER 3
RELATED WORK

This chapter presents VMs implemented with RPython. A summary of these VMs can be seen in section 3.9 on page 16. The work described in this paper was done in the context of the Haskell-Python project. The history and status of this project are described in section 3.1.

3.1 Haskell-Python (Haskell)

Bolz, Fischer, and Christiansen [11] have implemented[1] a VM for lambda calculus with RPython. The project is named Haskell-Python, while the VM is named PyHaskell. Lambda calculus is a system for expressing computations with variable bindings and substitution. The VM is based on an operational semantic for evaluating an extended lambda calculus, designed by Launchbury [21].

Launchbury's semantic for lazy evaluation does not map directly to System $F_C^{\uparrow}$, which the Core language implements. System $F_C^{\uparrow}$ provides support for `newtype`, `GADT`, `associated types`, `functional dependencies` and `type families` [32, 36]. Some of these features turn the type system into an explicitly-typed programming language on its own [37].

We need to figure out which features of System F_C we want to support, how to translate them to the Launchbury semantic if possible, and which features to ignore. This is described in section 4.3 on page 20. Some of these features, e.g. `type families`, are only available through GHC extensions, which are outside the scope of this project.

Skrede [31] has extended the project by designing and implementing a pipeline that uses GHC as a front-end for the lambda calculus VM. The structure of the pipeline can be seen in Figure 3.1. The pipeline starts with

[1] Their implementation can be found in the *clean2* branch of Haskell-Python [11].

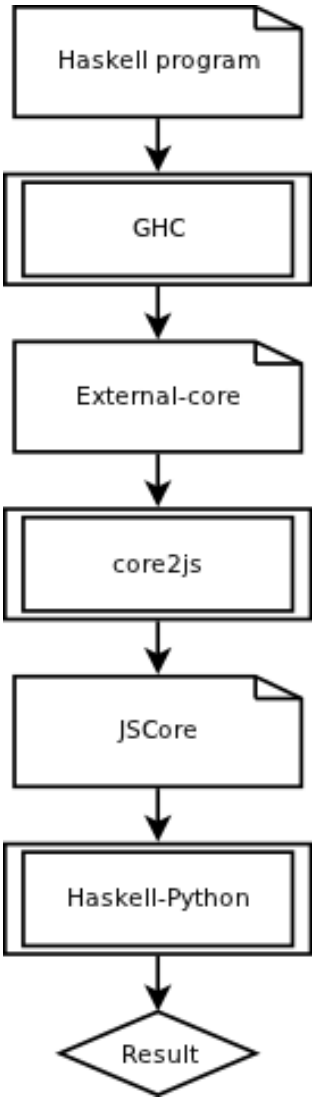

Figure 3.1: Haskell-Python pipeline by Skrede [31]

the GHC front-end that takes care of parsing, type checking and desugaring Haskell source code. GHC then generates an intermediate representation of the Core language that is written to a "hcr" file. The `core2js` Haskell program then converts the "hcr" file to JSCore, a JavaScript Object Notation (JSON) representation of Core, with the help of a Haskell package called extcore[2]. Finally the JSCore file is parsed and interpreted by PyHaskell [31].

The project is far from complete, especially the pipeline lacks features necessary for handling normal Haskell programs:

- GHC external core support has not been maintained during new releases of GHC. Many GHC libraries cannot be converted to the external Core representation.
- Extcore package fails to parse some "hcr" files.
- core2js program fails to generate some JSCore files.
- While PyHaskell supports case expressions, let statements and normal function applications, it lacks support for the following Haskell features: guards, pattern matching, list comprehensions, and more (see section 5.1 on page 27).
- PyHaskell is mostly written in Python instead of RPython, so it cannot be translated to C nor include a JIT. It can only run on top of another Python interpreter (CPython or PyPy).

These problems means that PyHaskell is so far unable to use any Haskell modules that comes with GHC. Possible solutions to these problems are presented in section 6.4 on page 32.

3.2 PyPy (Python)

This section refers to the second part of the PyPy project, the Python interpreter written in RPython. It is the most notable of the RPython VMs, and it is a complete Python version 2.7 compatible interpreter, and real alternative to the reference Python interpreter CPython [3]. One main goal of PyPy was to provide a faster Python interpreter [5, 14].

The PyPy interpreter consist of three parts: bytecode interpreter, built-in datatypes, and built-in libraries [5]. PyPy uses almost identical bytecodes and data structures as the CPython interpreter. The largest difference is an abstraction called object spaces. Object spaces encapsulates implementation details of Python objects. One benefit of object spaces is that a single data type may have multiple implementations, for example the Python long type can be a word-sized integer when it is small enough. PyPy also gains

[2]extcore package: http://hackage.haskell.org/package/extcore

performance benefits by specializing Python dictionaries when they have uniform keys [24].

The interpreter is implemented in RPython mostly, on top of the RPython translation toolchain [24]. The architecture of the toolchain is described in section 2.7 on page 6. The toolchain allows the interpreter to run on top of JVM and CLR in addition to generate C code.

The complex abstractions used by the interpreter comes with a performance cost. To remedy this cost, RPython contains a meta-tracing JIT that adds a JIT to the interpreter [5, 24]. PyPy is the largest user of the meta-tracing JIT, and so most of its improvements are driven by the goal of making PyPy faster. The meta-tracing JIT is described in more detail in section 2.8 on page 7.

PyPy has a comprehensive suite of macro benchmarks. Without the JIT, PyPy performs up to four times slower than CPython. With the JIT, PyPy is on average five times faster than CPython[3] [24].

3.3 Pyrolog (Prolog)

Prolog is a general-purpose, high-level, declarative, logic programming language that is based on a variant of first-order predicate calculus. Prolog has a single data type `terms`, which can be `atoms`, `numbers`, `variables` and `compound terms` (lists and strings) [4].

Bolz, Leuschel, and Schneider [8] have implemented a Prolog VM with RPython called Pyrolog[4]. Their aim was to show that declarative languages (such as Prolog) could benefit from a JIT compiler, and that RPython could be used to implement other programming languages than just Python [8].

According to Bolz et al. the Pyrolog interpreter consist of "about 5000 lines of RPython code, of which 1000 lines are implementing built-ins and 1700 are tests" [8]. The goal was to create a simple high-level object oriented Prolog implementation. Logic variables and non-variable terms are represented by classes such as `Var`, `Atom`, `Number`, and `Float`. Prolog objects are instances of subclasses of `PrologObject` base class [8].

The interpreter is based on continuations, either success or failure continuations. These encapsulates all the state and behavior of the interpreter. Code which remains to be executed is contained in the success continuation, while the failure continuation is used if backtracking is required. Interpretation calls the current success continuation until computations are finished, or it fails and calls the current failure continuation. Interpretation consumes and possibly replaces the current continuation [8].

The Pyrolog interpreter can be compiled with a tracing JIT, which requires some hints to perform well. The following hints were added to Pyrolog

[3] PyPy version 1.8 versus CPython version 2.7.2, see: http://speed.pypy.org/

[4] Pyrolog: https://bitbucket.org/cfbolz/pyrolog/overview

source code: [8]

- A hint to indicate the interpreter's main loop.
- A hint to annotate which variables belong to the interpreter.
- A hint to indicate the code that closes a loop.
- A hint to mark classes that are immutable.

Prolog lacks an explicit loop construct, but for the JIT a loop is simply a situation where the same rule is applied repeatedly [8].

Bolz et al. have evaluated the Pyrolog interpreter with micro benchmarks as well as well-known slightly larger programs. With the JIT Pyrolog was faster than all other implementations tested on all micro benchmarks except one. Without JIT, Pyrolog is significantly slower than other Prolog implementations. On the larger Prolog programs "the JIT gives a speedup of up to ten times" [8] on all but one. Memory footprints of each Prolog interpreter was also measured, and Pyrolog used about two to five times more memory on most benchmarks.

Bolz et al. argues that Prolog can greatly benefit from JIT compilation techniques, and concludes that the Pyrolog interpreter "is reasonably efficient and can be very fast in cases where the generated JIT works well" [8].

3.4 HappyJIT (PHP)

PHP: Hypertext Preprocessor (PHP) is a general-purpose imperative, object-oriented, procedural programming language. PHP has weak, dynamic types and are used mainly for server-side web-development [18].

Homescu and Şuhan [18] have implemented a PHP VM with JIT called HappyJIT, which reuses the PHP parser from Zend PHP engine[5] and all the existing compilation code from PyPy. HappyJIT therefor only need to implement the interpreter loop and data structures for representing a PHP program in memory. An extension to the Zend PHP engine, Advanced PHP Cache (APC), dumps Zend bytecodes to disk that HappyJIT reads back into memory. Zend bytecodes are then converted to a more efficient set of bytecodes before being interpreted by HappyJIT [18].

HappyJIT represents each basic PHP type (except resource) with a wrapper class, for example `W_float` represents float. These classes encapsulates values of constants and variables, and are organized in a hierarchy with `W_object` as the base class. As strings in PHP are mutable and RPython strings are not, HappyJIT implements strings as a list of characters. PHP arrays are also more flexible than RPython lists, so `W_array` holds a pointer

[5] Zend PHP: http://www.zend.com/en/

to either a linear (list) array implementation or dictionary array implementation [18].

HappyJIT implements a standard bytecode based interpreter that load bytecode from an array and handle them with a series of if statements. The interpreter supports executing several programs serially, with a global context containing immutable global data and built-in functions. A runtime context, which are reset after each script, stores all variables in one linear stack [18].

There are two versions of the interpreter, one without a JIT (Happy) and one with (HappyJIT). Homescu and Şuhan have evaluated both against Zend PHP engine and Roadsend PHP compiler with two set of benchmarks, PHPbench[6] and some PHP tests from Computer Language Benchmark Game[7]. They found that most tests were faster with HappyJIT than Zend and Roadsend, and the non-JIT version was slower. Their implementation had performance problems when it came to strings and recursive functions, but long-running loops had significant improvements when JIT-compiled [18].

3.5 Spy (Smalltalk)

Smalltalk is an object-oriented programming language that support dynamic typing and run-time type checking [35].

Bolz et al. [6] have implemented a Smalltalk-80 VM called Spy that is a re-implementation of a VM named Squeak. The goal of Spy was to port the Squeak platform to high-level runtimes such as JVM and CLR. These runtimes are possible back-end targets for RPython — any VM implemented with RPython could possibly be run on top of these two runtimes [6].

Squeak is written in Slang, a restricted subset of Smalltalk, that is designed to be easily translated into C. This is similar to PyPy Python interpreter that is written in a restricted subset of Python. But while Slang maps directly to C, RPython is much closer to the full Python language [6].

Spy consists of four main parts: bytecode interpreter, set of primitives, image loader, and an object model. The interpreter contains primitive methods for low-level operations, instead of special bytecodes for arithmetic and such. Spy's object model is a hierarchy of subclasses of the abstract base class `W_Object`, that represent a Smalltalk object. There are subclasses for floats, integers, pointers, bytes, words, and more [6].

Representing Smalltalk classes in RPython was challenging for Spy, that was solved by shadow objects. Any Smalltalk object can have an associated shadow object that can hold arbitrary information about the actual object. Shadow objects works as a general caching mechanism [6].

[6]The PHP benchmark: http://phpbench.com/

[7]The Computer Language Benchmark Game: http://shootout.alioth.debian.org/

Bolz et al. analyzed Spy performance on processed bytecodes per second and message sends per second. Spy translated to C and Spy running on CLR were compared with Squeak and other Smalltalk VMs. Squeak was ten times faster than Spy translated to C, and 100 times faster than Spy on CLR [6]. Spy does not use RPython JIT generator, which might provide the speed boost needed to compete with Squeak.

Two areas where Spy outperformed Squeak was size of the code base, and development time of the implementation. Squeak is almost three times as large, when measured in thousand lines of code (KLoC), and the Spy VM was implemented in a single week [6].

3.6 PyGirl (Gameboy)

Bruni and Verwaest [12] have implemented an executable VM prototype for a Nintendo Game Boy, that they named PyGirl. PyGirl is a port of an existing VM, Mario, that was implemented in Java. Bruni and Verwaest compared these two VMs to show how a high-level language can minimize code and reduce complexity, without substantial loss of performance [12].

The Game Boy hardware system consist of six pieces: CPU, Sound, Video, JoyPad, Ram, and ROM. These pieces communicate through shared memory. PyGirl provides one class for each hardware piece, and platform-specific parts are abstracted away with driver interfaces. The overall structure of PyGirl equals Mario, as it maps directly to the hardware [12].

In addition to Mario, PyGirl was compared against two other Game Boy VMs: JavaBoy[8] and AEP_{GB}[9]. Bruni and Verwaest found that by implementing PyGirl with RPython they significantly reduced the code complexity. For example, PyGirl implements CPU related classes in one KLoC, while the other three VMs require between 2.7 and 4.2 KLoC [12].

On benchmarks PyGirl performed about 40% slower than Mario after the JVM JIT had warmed up. PyGirl ran at linear speed, as it did not include RPython JIT. Therefor Bruni and Verwaest concluded that high-level prototypes, with help of meta-programming, can reduce code complexity without substantial loss of performance [12].

3.7 Converge (Converge)

Converge is a dynamically typed object-oriented programming language for exploring research on domain specific languages and compile-time meta-programming. Converge has an expression evaluation system that can perform limited backtracking [5]. Tratt [34] designed the Converge programming

[8] JavaBoy: http://www.millstone.demon.co.uk/download/javaboy/

[9] AEP_{GB} seems to have been discontinued.

language, which was first implemented twice in C and then reimplemented with RPython.

The Converge RPython VM is split into three parts: bytecode interpreter, built-in datatypes, and built-in libraries. Converge also has a compiler, written in Converge, that translates programs into bytecodes. The compiler can be invoked manually or transparently by the VM [5].

Tratt [34] mentions three areas of improvements from implementing his VM in RPython instead of C: 1) The C VM consist of 13 KLoC while the RPython VM is only about 5.5 KLoC, and the C VM is considerably more complex; 2) Tratt estimated that the second C VM required 18 man months to create, while the RPython VM took two to three man months; 3) The C VM was extremely slow, while the RPython VM (without a JIT) was two to three times faster, and with a JIT was over 20 times faster on the three benchmarks tested. Tratt points out very little time was spent optimizing the RPython VM and the speed-ups came almost free with RPython [34].

3.8 Other virtual machines

There are a few more partial or incomplete attempts at using RPython to implement VMs worth mentioning.

Schneider [29] created a VM for the Io language with RPython. Io is a small and dynamic object-oriented programming language with a prototype based object model. Io support runtime reflection and meta-programming [13].

Zalewski et al. have an incomplete implementation of JavaScript[10]. JavaScript is an object-oriented, functional programming language based on prototypes that support dynamic and weak types. It is widely used for web programming [26].

Bömmels have created an incomplete implementation of Scheme[11]. Scheme is a dynamically typed, functional programming language based lambda calculus, that was one of the first to support first-class continuations. Scheme is a dialect of the Lisp programming language, and both use a fully parenthesized prefix notation (s-expressions) for syntax [1].

3.9 Summary

The VMs implemented with RPython are considered high-level language VMs that only exist virtually, except for PyGirl which is a whole-system VM that emulates actual hardware [12].

PyPy and Converge VMs are considered complete implementations of their respective programming languages [24, 34]. PyGirl VM is considered

[10] RPython JavaScript implementation: http://bitbucket.org/pypy/lang-js/
[11] RPython Scheme implementation: http://bitbucket.org/pypy/lang-scheme/

almost complete as its sound support is incomplete [12]. Spy VM lack support for a handful of primitives, making it almost complete. Pyrolog VM is also considered almost complete. The remaining VMs are prototypes or partial implementations of their respective languages [6, 18].

Table 3.1 provides an overview of the different VMs implemented with RPython so far[12], including when development started and stopped[13].

The different VMs set out with different goals in mind, and therefor have achieved different results. Overall, we can summarize these different results as three advantages that one can expect to gain when using RPython.

Better performance with just-in-time compiler: PyPy, Pyrolog, HappyJIT, and Converge VMs all gained performance improvements on some or all benchmarks against alternative implementations [5, 8, 18, 24, 34].

Reduced code base and complexity from high-level language: Spy, PyGirl, and Converge VMs are re-implementations with smaller code bases than their original implementations [6, 12, 34].

Less developer-time used during implementation: Spy was created in a week, and Converge was implemented in only 2-3 man months [5, 6, 34].

3.9.1 Programming language paradigms

Table 3.2 list major programming language paradigms implemented with RPython. While a few implement the functional paradigm, Haskell is purely functional, which means that functions cannot mutate data or have any side effects. Haskell also support the lazy (non-strict) paradigm, which has yet to be implemented with RPython[14] [22].

[12]KLoC measures were taken on `default` branch of respective repositories May 12 2012, with Mercurial extension hg-cloc (`https://bitbucket.org/jinhui/hg-cloc`). Only files identified as Python code were included. KLoC for PyPy was not included as it contains code for both the interpreter and RPython.

[13]Stopped is last significant update unless it has been updated recently.

[14]Io does support lazy evaluation of function parameters, but the RPython Io VM is incomplete [13, 29].

Table 3.1: Overview over virtual machines implemented with RPython

			Development			
VM Name	Language	JIT	Started	Stopped	KLoC	Status
Converge	Converge	Yes	2011		5.8	Complete
HappyJIT	PHP	Yes	2011		4.1	Partial
io	Io	No	2009	2011	3.6	Partial
js	JavaScript	Yes	2006	2011	5.5	Partial
PyGirl	Game Boy	No	2008	2009	10.7	Almost complete
PyHaskell	Haskell	No	2011		1.9	Partial
PyPy	Python	Yes	2004			Complete
Pyrolog	Prolog	Yes	2006	2012	16.7	Almost complete
Spy	Smalltalk	No	2007	2011	5.3	Almost complete
Scheme	Scheme	No	2006	2012	4.0	Partial

Table 3.2: Overview over paradigms implemented with RPython

Paradigm	Languages
Declarative	Prolog
Functional	JavaScript, Python, and Scheme
Imperative	Converge, JavaScript, PHP, Python, and Scheme
Logic	Prolog
Meta	Converge, Io, Python, and Scheme
Object-oriented	Converge, Io, JavaScript, PHP, Python, and Smalltalk
Procedural	PHP, Python, and Scheme
Prototype	Io and JavaScript
Reflective	Io, PHP, Python, Scheme, and Smalltalk

CHAPTER 4

METHOD

This chapter describes my technical contributions to the PyHaskell VM.

4.1 Z-decoding

The external representation of Core made with GHC uses Z-encoded[1] names [33]. Some of these names, such as module identifiers, were decoded by PyHaskell during parsing. Other names were kept Z-encoded, which made the internal representation of parsed code very hard to read. For example (–>) was named ZLzmzgZR in PyHaskell when Z-encoded.

My first change to the project was moving the Z-decoding from the jscparser module into the core2js program, and make sure all parts of the external representation was Z-decoded before converted to JSON. The extcore package provided the function for Z-decoding strings.

Package, module, and identifier names were joined together into one string in core2js, and then split back up again when parsed by the jscparser module. I was free to remove this behavior after I had moved Z-decoding into core2js. I kept package and module as one single name as we do not need to deal with packages, everything can be modules. Identifiers are now an array of two strings: the module name and the reference name.

4.2 Repository structure and refactoring

My second change to PyHaskell was to take advantage of some of the lessons from other RPython implementations — their repository structure and module layout. The main inspiration came from repositories of Pyrolog and HappyJIT.

[1] Skrede [31] provides a more detailed description of GHC's Z-encoding

We adopted a flatter structure for Haskell modules implemented in Python and placed them in their own sub-package of PyHaskell named `builtin`. Modules for implementing the interpreter were moved to a sub-package called `interpreter`.

Other modules such as main, makegraph, and makehcj were moved into the root of the PyHaskell package. Implementation of primitive types such as `Char`, `Int` and `Float` were moved out of built-in modules and into a new module named primtype.

After these changes the PyHaskell VM consist of four main parts: the interpreter (jscparser, haskell); built-in modules (show, cstring, io, and more); primitives and primitive types (prim and primtype); and GHC libraries converted to the external Core representation (contained in the `ghc_modules` folder). Except for the last part, PyHaskell is divided into the same components as PyPy and Converge VMs [5].

4.3 Mapping Core to PyHaskell

The next step was to resolve which features of Core to ignore, and how to extend PyHaskell to support the remaining features of Haskell. We can ignore the complexity of System $F_C^{\uparrow}$, because we target GHC version 7.4.1 where the external Core representation is System F_C.

4.3.1 Kinds

System F_C has a much simpler kind system than System $F_C^{\uparrow}$, with only a few types of kinds: lifted kinds '*'; type constructors (that are nested with '->'); unlifted kinds '#' (unboxed/primitive values); equality kind ':=:'; and open kind '?' (that represent unspecified kind) [32, 33].

As unlifted kinds simply are unboxed types that are explicitly named with a magic `#` symbol (for example `Int#`) in `GHC.Prim` we choose to ignore them and implement these unboxed types boxed. GHC provides (and uses) unboxed types for improved performance [20]. We believe we can achieve similar performance by using the JIT to remove much of the overhead incurred by boxed types.

Furthermore, as we rely on GHC for type checking, also in the Core language, we are now free to completely ignore kinds. This allows our VM to be simpler than Core.

4.3.2 Types

Core contain type information in many places, for example function signatures and type variables [33]. As we rely on GHC for type checking, we can simply ignore any types in these instances.

For explicitly typed literals, which can only be primitive types otherwise they would be an expression of a data constructor, Skrede [31] have included the type in the JSON representation. During parsing we use this type when creating literals, and all other type information and usage is removed.

4.3.3 Constructors

Haskell has three types of user-defined datatypes: algebraic datatypes (data constructors); renamed (newtype declaration); and type synonyms (type constructors). Type constructors are used to create new types from primitive (or other) types, while data constructors group values together. Data constructors are first class values in Haskell, they may be passed to functions or be elements of other data types [15].

Data constructors were wrongly implemented as Symbol-instances, as RPython prevented Symbols to be mixed with instances of HaskellObject-subclasses. They should be what are called constructors in Launchbury's semantics, a collection of values with a named symbol, but because they can also be used as functions during parsing we implemented them as functions that return a constructor.

Type constructors were implemented as constructors in PyHaskell, but for reasons mentioned in subsection 4.3.2 we did not need to support them, and I removed them. Newtype, which is a simpler but strict (not lazy) version of type constructors [15], should not be supported either.

4.3.4 Type aliases

Some data constructors represent unboxed versions of primitive Haskell types, such as `Char#`, `Int#`, `C#` and `I#`. As mentioned earlier, GHC use these extensively for performance reasons [20], and we can represent them with their boxed primitive version.

These type aliases were implemented as constructors with a single value, which are problematic as any function that expect a primitive type must have special handing of aliases of the primitive type. For example `putStrLn` expects a list of `Chars`, and do not know what to do with a list of `Char#s` or `C#s`. Listing 5 on page 44 shows the source code of our implementation of `putStrLn`.

Therefor, to simplify the implementation of built-in modules and primitive operations, we check during parsing if a data constructor is a type alias for a primitive type and then use the primitive type directly. This also removes a level of abstraction that could lead to performance problems compared to unboxed values in GHC [20]. This simplification only affects PyHaskell during parsing, not evaluation.

4.4 Converting PyHaskell from Python to RPython

The original lambda calculus VM implemented by Bolz et al. [11] was implemented mostly in RPython, but the functionality added by Skrede [31] were written in standard Python. To be able to translate PyHaskell to C, and to use the RPython meta-tracing JIT, I translated the Python code to RPython.

As there is a limited amount of documentation of RPython, how to use it and what is valid RPython, converting Python to RPython is complex and time-consuming work. Specifically, one must often translate to validate ones work, and to discover if an operation is RPython or not. The error messages provided when translation fails are often very cryptic or sometimes without any useful information. In the latter case, one must use the Python debugger to try and discover what caused the problem. As we have shown in section 5.2 on page 27, translating even a small program such as PyHaskell takes substantial time.

A major restriction of RPython, that was not explicitly mentioned by any RPython papers, quickly became clear: Only a subset of a) built-in functions, b) built-in data-structures, and c) modules from the Python standard library are available as RPython. RPython has its own standard library, `pypy.rlib` sub-package, where this subset is implemented in RPython code.

Source code files were parsed by using the `open` function, which is not supported with RPython. It was replaced with `open_file_as_stream` from `pypy.rlib.streamio` module.

4.4.1 main.py

In the *main* module I had to remove `sys.exit` calls, change `subprocess.call` into `os.system`, change `os.remove` into `os.unlink`, and use `os.stat` instead of `os.path.getmtime` and `os.path.isfile`.

4.4.2 module.py

The *module* module which handles Haskell modules required few changes, but some new functionality. The `CoreMod` class gained methods for adding data constructors, which should be used by built-in Haskell modules implemented in RPython. These data constructors are described further in subsection 4.3.3 on page 21, and include aliases for primitive types that are explained in subsection 4.3.4.

One complex addition to this module was decorators for adding functions implemented in RPython to dictionaries in `CoreMod`-instances. These decorators replace the `expose_primitive` decorator from the *haskell* module. They should create instances of `PrimFunction`, type-check arguments given to the function, and finally register the `PrimFunction`-instance in the

correct `CoreMod` dictionary. As `PrimFunction` keeps arguments to the original function as a list, the decorators should also automatically unpack the arguments.

These decorators would be straight-forward to implement in Python, but the restrictions of RPython made it more complex. Decorators are executed when Python modules are imported (at what we refer to as import-time), and since the RPython toolchain runs on top of a complete Python interpreter, RPython support the full Python language at import-time [24]. Therefor the decorators are partially implemented in full Python, with the inner-most function implemented in RPython. RPython prevents us from unpacking a list with the '`*`' Python operator, so I used `eval` at import-time to create several anonymous functions that will convert known sized lists into tuples.

The source code of these decorators can be seen in Listing 1 on page 42, and the implementation of the `cons` operator that use one of these decorators can be seen in Listing 2 on page 43. The functions that run at import-time is marked with `NOT_RPYTHON` docstrings. In line seven of the latter listing, `@expose_data_constr` decorator is given three arguments: the module it belongs to; the name of the function; and a list of the types for the function arguments. The last argument is used for type-checking and unpacking arguments.

These decorators are a form of meta-programming that is not supported by low-level languages that are used for implementing VMs, where we would need to manually check the types of each argument and more. Listing 3 on page 43 shows an alternative implementation of the `cons` operator without these meta-programming techniques.

4.4.3 jscparser.py

The *jscparser* module required major changes — almost a complete rewrite. The main problem was traversing of nodes in the abstract syntax tree (AST), which was implemented with the use of the generic `visit` and `visit_object` methods. The former was used to visit sub-nodes in the tree and called the latter, which detected what type of node we visited with a long list of `if-elif` statements. First, these methods should not have been used directly, the dispatch method from RPython's AST traverser was designed for visiting the correct node. Secondly, RPython require these methods to have the same signature — to accept the same arguments and return the same type of results.

My solution was to move handling of specific types of nodes into their own methods, where I could type-check returned values with `assert` statements. Dealing with one node type at the time, I could gradually convert the whole module into RPython while ensuring that all tests still passed. This transformed `visit_object` from a single method of over 300 lines of code to several methods of 50 lines or less which each deal with a specific

node type, for example `visit_atomic_expr` handles the four types of atomic expressions in Core.

An added benefit of this work was that instead of having to check the type of each node, even when we knew from a parent node what type a child must be, we could directly call the right method and avoid unnecessary checks.

4.4.4 haskell.py

The *haskell* module, which implements Launchbury's semantic, was mostly implemented in RPython already by Bolz et al. [11].

The `constr` function provided a convenient way to create constructors, but as it was not RPython I removed it and changed any use of it to `make_constructor`.

`ForwardReference` provided a way to create a reference to something that did not exist yet, and was necessary for recursive functions to refer to itself in its rules. Instead of converting it to RPython I removed it, and allowed `Function`-instances to be created before the rules were ready. Then after the rules have been created, I simply update the instance.

4.4.5 targethaskellstandalone.py

The *translate* module in `pypy.translator.goal` is the entry point for translating VMs with RPython. This module require a target, which I named *targethaskellstandalone*. Our target was inspired by similar files in several other VMs implemented with RPython.

The following command, given the translate module and our target, will produce a Haskell VM written in C code:

```
pypy translate.py targethaskellstandalone.py
```

The *translate* module can be given arguments for specifying the level of optimizations that should be used during translation. The `--opt=jit` argument will produce a VM that contain RPython's meta-tracing JIT, which is described in section 2.8 on page 7.

4.4.6 Built-in modules

The built-in modules required two types of changes. First, to use the new decorators on functions implemented in RPython instead of Haskell, which is explained in subsection 4.4.2 on page 22. Second, as I introduced special handling of primitive type aliases such as `Char#` (see subsection 4.3.4), almost all functions in built-in modules could be simplified.

For example, the implementation of `unpackCString` uses the `expose_var` decorator, and `putStrLn` can handle a list of `Chars` or any primitive alias of

Table 4.1: Overview over Haskell benchmarks used

Benchmark	GHC compile time	Recursive	Source code listing
Fibonacci	1.06 s	Yes	Listing 10
Multiply	- s	Yes	Listing 11
Iterative case	1.69 s	No	Listing 12

`Chars` such as `C#s`. The source code of these two implementations are listed in section A.2 in the appendix on page 41.

The implementation of the list concatenation operator '++' had to be completely rewritten as it contained nested functions, was not lazy, and was implemented recursively. RPython does not support nested functions and Python is not tail-recursive. Listing 6 on page 45 list the source code of my implementation, which is iteratively.

4.5 Benchmarking

We need to benchmark PyHaskell before we can answer the research questions Haskell-Python asks. GHC uses the *nofib* benchmark suite [23], but as PyHaskell only support a subset of Haskell we will not be able to run *nofib* on our VM. Instead we have implement a few micro-benchmarks that only require features we support.

To see how RPython and its meta-tracing JIT affect the performance of our VM we will run the benchmarks on three different version of PyHaskell: first, PyHaskell not translated to C (running on top of CPython); second, PyHaskell translated to C without the JIT; third, PyHaskell translated to C including the RPython JIT.

We will also run the benchmarks on GHC compiled and GHC's interactive environment (GHCi). As PyHaskell is a back-end for GHC, the results cannot be compared directly. When benchmarking with PyHaskell we will use Haskell code converted to external JSON representation, to make it closer to the compiled code of GHC.

The benchmarks must also run long enough to provide statistically significant results when compiled with GHC, and run long enough allow the RPython meta-tracing JIT to warm up. On the other hand they must be able to finish with PyHaskell running on top of CPython. These restrictions and limited Haskell support PyHaskell provides made it quite complicated to design suitable benchmarks.

Table 4.1 list the benchmarks we have use, including time to compile them with GHC. Whether they are recursive or not is noted as currently PyHaskell will only JIT recursive functions. The results of running these benchmarks are listed in Table 5.3 on page 29.

The naive implementation of the Fibonacci sequence is often regarded as

Haskell's "Hello, world!". It is also very suitable for our benchmark needs, so it was the first benchmark we tried.

The multiply benchmark is very similar to the Fibonacci benchmark, the main difference is that multiply uses integer multiplication instead of addition.

To see how PyHaskell JIT behaves on code the JIT cannot compile, we created the iterative case benchmark that iteratively calls a function 200 times. The function uses case expression to match data constructors.

CHAPTER 5
RESULTS

This chapter present results from our work on the Haskell-Python project.

5.1 Haskell features supported

We have create a set of tests to get an overview over Haskell features supported by PyHaskell. Results of running these tests can be seen in Table 5.1. This is not an exhaustive list of Haskell features. These tests were run on PyHaskell (running on top of CPython 2.7.3) from the Haskell-Python repository (branch "even", revision `89f8127667bb`) [11].

As the goal is to be a back-end for GHC and reuse GHC's Haskell libraries, we only aim to implement a small subset of Haskell's features ourselves. Problems on the GHC end of the pipeline prevents us from reusing most GHC libraries, and we would rather fix the GHC problems than implement temporary fixes to support the missing features. The pipeline problems are explained in section 3.1 on page 9, while possible solutions are discussed in section 6.4 on page 32.

5.2 RPython translation timings

Table 5.2 show time used to translate PyHaskell into C. It is included to show that we are able to translate PyHaskell, which means PyHaskell is fully implemented with RPython. PyHaskell was translated on the hardware and software described in section 5.3 on page 29.

The different steps in the RPython translation toolchain are described in section 2.7 on page 6. The JIT step is only run when JIT optimizations are turned on (as described in subsection 4.4.5 on page 24), and adds RPython's meta-tracing JIT to the generated VM.

Table 5.1: Overview over Haskell features PyHaskell support

Feature tested	Success	Comment
Hello world!	Yes	
Case expression	Yes	
Let expression	Yes	
Data constructor	Yes	
Partial function application	Yes	
Recursive function	Yes	
List concatenation '++'	Yes	Source code in Listing 7
Cons operator ':'	Yes	Source code in Listing 8
Function composition operator '.'	Yes	
Function application operator '$'	Yes	
Indexing operator '!!'	No	Missing "GHC.List" support
Recursive let expression	No	Missing "GHC.List" support
Guards	No	Problem with extcore package
Pattern matching	No	Source code in Listing 9
List comprehension	No	

Table 5.2: Time used to translate PyHaskell to C

	Translation time in seconds	
Translation step	PyHaskell no-JIT	PyHaskell JIT
Annotate	59.0	57.8
RType	98.6	97.5
JIT		565.8
Back-end optimizations	62.9	215.1
Stack check insertion	2.0	15.6
C database	147.5	247.4
Generate C source code	67.7	191.9
Compile C source code	24.9	198.7
Total	462.7	1589.9

Table 5.3: Benchmark results

	Benchmark time in seconds		
Virtual machine or compiler	Fibonacci	Multiply	Iterative case
PyHaskell (on CPython)	5733.02 s	2095.90 s	171.40 s
PyHaskell no-JIT	27.24 s	9.71 s	1.48 s
PyHaskell JIT	2.10 s	2.17 s	1.38 s
GHC	0.56 s	0.24 s	0.005 s
GHCi	9.64 s	3.63 s	0.06 s

Table 5.4: Evaluation of benchmark results

VMs compared		Order of speed-up on benchmark		
Faster VM	Slower VM	Fibonacci	Multiply	Iterative case
PyHaskell no-JIT	PyHaskell (on CPython)	210	216	116
PyHaskell JIT	PyHaskell (on CPython)	2730	966	124
PyHaskell JIT	PyHaskell no-JIT	12.9	4.47	1.07
GHC	PyHaskell no-JIT	48.6	40.5	296
GHCi	PyHaskell no-JIT	2.83	2.67	24.7
GHC	PyHaskell JIT	3.75	9.04	276
PyHaskell JIT	GHCi	4.59	1.67	-

5.3 Benchmarks

We ran each version of the benchmarks three times, the average result of each version is listed in Table 5.3. We have calculated the order of speed improvements of the faster VMs versus the slower ones, which can be seen in Table 5.4.

PyHaskell[1] was translated with the RPython toolchain included in the latest stable PyPy release, PyPy version 1.9[2]. "PyHaskell (on CPython)" is PyHaskell running on top of CPython version 2.7.3. GHC and GHCi uses "The Glorious Glasgow Haskell Compilation System, version 7.4.1".

The benchmarks were run on an Intel Atom CPU D525 processor with 1.80 GHz and 512 KB of cache on a machine with 2 GB RAM running Ubuntu Desktop 12.04 32 bit, with Linux kernel 3.2.0.

[1] PyHaskell from Haskell-Python [11], branch "even", revision `89f8127667bb`.
[2] PyPy version 1.9 from: https://bitbucket.org/pypy/pypy

CHAPTER 6

DISCUSSION

This chapter discuss the results of our work and possible solutions to the problems of the current PyHaskell implementation.

6.1 Benchmark results

When we look at the results of the Fibonacci benchmark, the power of the RPython translation toolchain becomes clear. PyHaskell running on top of CPython is extremely slow compared to all the other VMs: it is over 200 times slower than PyHaskell translated to C without JIT; and almost 3000 times slower than PyHaskell JIT.

The power of RPython's meta-tracing JIT is also conclusive as PyHaskell JIT is over 12 times faster than PyHaskell no-JIT on the Fibonacci benchmark. On the iterative case benchmark PyHaskell JIT perform similar to PyHaskell no-JIT, which is expected as our JIT only trace recursive functions. PyHaskell JIT is slightly faster as it is translated with a higher optimization level than PyHaskell no-JIT.

I suspect the slow performance on the iterative case benchmark compared to the Fibonacci benchmark (regardless of JIT or not) is caused by our parser. The JSON encoded external representation of Core is significantly more verbose than Tolmach et al. [33] Core representation. Additionally, we have not yet added any RPython hints to the *jscparser* module that traverse the AST.

The most interesting comparison is PyHaskell against GHC. On the Fibonacci benchmark PyHaskell JIT is only 3.75 times slower than GHC. And surprisingly PyHaskell JIT is over four times faster than GHCi, which interpret instead of compile Haskell code.

Beside the initial work on PyHaskell by Bolz et al. [11], no effort has been taken to improve the performance of our VM. It is therefor very promising

to see how close we are to GHC performance on specific benchmarks that the JIT handles very well.

6.2 PyHaskell formal definition

Tolmach et al. [33] present a formal definition of the external representation of GHC's Core. Launchbury [21] present a formal definition of the operational semantic that PyHaskell's lambda calculus VM implement. Skrede [31] present a formal definition of JSCore, which is a JSON representation of the Core representation defined by Tolmach et al. I will not attempt to define PyHaskell as it is still under development and with frequent changes.

6.3 Lessons from related work

The literature review on VMs implemented with RPython allows us to reuse the experience of other in this field of research. Most of the VMs are bytecode-based, but the Prolog VM stands out from the rest with it continuations-based implementation. Therefor Pyrolog is probably the VM closest to PyHaskell, but there are some important differences between Prolog and Haskell. While Prolog is a first-order predicate calculus, Haskell is based on lambda calculus. Where Prolog has a single data type terms, Haskell has several data types such as complex numbers, functions, arrays, and more. Prolog also lack several features Haskell has such as higher order and anonymous functions and lazy evaluation [4, 5, 8, 16, 28].

The first lesson we have used is the structure of the repository, which is described in section 4.2 on page 19.

When convert PyHaskell from Python to RPython the papers describing the restrictions of RPython was very important, as there are very limited documentation on RPython the language and RPython the toolchain. These restrictions are listed in section 2.6 on page 6. Also the Pyrolog VM was useful as the `pypy.rlib.parsing` package was created for it, which we use in our *jscparser* module. I had to look at the Pyrolog source code to fully understand how to use the `RPythonVisitor` class. The process of converting PyHaskell into RPython is described in section 4.4 on page 22.

We believe the greatest value of the literature review will come when we try to optimize PyHaskell to achieve better performance. Bolz and Tratt [5] explain the optimization techniques used by PyPy and Converge, and concrete lessons learned from these techniques. Homescu and Şuhan [18] explain steps they had to take to make their VM performant for the PHP language. The need for performance improvements are explained in section 6.5.

Finally we think that Pyrolog and HappyJIT VMs [8, 18] might provide useful lessons on implementation of built-in modules, which we need

as we need to extend PyHaskell's built-in modules to support more Haskell features.

6.4 Pipeline problems

Two possible solutions for the shortcomings of the pipeline is to either modify GHC and the extcore package to fix the known bugs, or to rewrite the pipeline to use GHC API directly instead of external Core representation. These solutions are complex tasks without guarantees of success.

Modifying GHC is a more attractive solution, as it will only require minimal additional changes to our `core2js` program. But GHC is a huge project with a large and complex code base, so making substantial changes to GHC inner workings is a daunting task.

Using the GHC API seems on the surface as a simpler and more straightforward task, but handling GHC's Haskell library will be challenging. It might require us to load library modules on demand, instead of compiling them to an intermediate representation as we do today. This could affect our performance making us unable to compete with GHC.

Skrede [31] has created a work-around by implementing some of the important Haskell libraries in Python. This work-around is not a real solution as we cannot re-implemented the whole GHC Haskell module library in RPython, it would be too time-consuming.

Another problem with the pipeline is interfacing with C code. Both GHC and RPython have foreign function interfaces (FFIs) for interfacing with C code. GHC's FFI uses information that is handled with special macros in GHC's source code. To be able to translate GHC's FFI to RPython's we might need to preprocess GHC's source code to find the necessary information. Another option is to manually implement the functionality that interface with C in RPython using RPython's FFI.

6.5 Further work for answering questions

To answer the research questions the Haskell-Python project holds, the PyHaskell VM should fulfill two goals: one, be able to run the GHC test suite; two, be able to run the *nofib* Haskell benchmark suite that GHC uses [23]. The following work are therefor required to complete the Haskell-Python project:

- Either fix GHC external core support and the extcore package; Or rewrite core2js/PyHaskell to use the GHC API.
- Implement support for remaining Haskell language features and primitive GHC operations.

- Implement support for other basic Haskell types.

Performance is also a key question in both research questions — to be able to see if RPython is suitable for lazy and pure languages, and to see if Haskell can benefit from JIT techniques. To achieve speed closer to GHC we need to better use the advantages of the RPython meta-tracing JIT.

As mentioned in section 2.8 on page 7 the JIT require some subtle hints in the source code to better reason on optimizations techniques it can use. Bolz et al. [11] have added hints to the lambda-calculus evaluation part of PyHaskell (the *haskell* module), but our parser does not contain any hints. Bolz and Tratt [5] have described general lessons for improving the performance of VMs implemented with RPython.

The PyPy project have developed tools to better understand how the RPython JIT affect VMs, for example the jitviewer[1]. We need to use such tools to inspect the code compiled by the JIT to see how it can be further optimized [24].

[1] jitviewer: https://bitbucket.org/pypy/jitviewer

CHAPTER 7

CONCLUSION

The Haskell-Python project has two goals: show that RPython is a suitable platform for implementing purely functional and lazy languages; and investigate if Haskell can benefit from JIT techniques. PyHaskell, a VM for the Haskell language implemented with RPython was created to answer these questions. PyHaskell is only a partial implementation of Haskell, so this paper set out to improve PyHaskell so that it could answer these two questions.

Problems with the GHC front-end of PyHaskell prevent us from answering definitely at this time, but my contributions to the project allow us to preliminary suggest that the answer is affirmative on both questions.

I have achieved the goal of converting the VM to RPython, which allowed us to translate PyHaskell to C and to take advantage of the RPython meta-tracing JIT. With the help of the JIT, PyHaskell is only 3.75 times slower than GHC on one benchmark, the naive Fibonacci sequence. As we have achieved such promising results without any attempt at optimizing the VM for the meta-tracing JIT, we think that with further work PyHaskell will be faster than GHC in some situations.

The other major contribution of this paper is a literature review of RPython VMs, which was crucial for completing the technical contributions of this paper. Furthermore, the literature review gives us a solid basis for further work on optimizing PyHaskell, which is necessary to answer our research questions definitely and to outperform GHC.

Planned future work is twofold: first, to fix the mentioned problems with the GHC front-end; second, to improve the performance of PyHaskell. We also plan to extend PyHaskell's support of the Haskell language, so it can run the *nofib* benchmark suite and the GHC test suite.

Acknowledgments I wish to thank Carl Friedrich Bolz for his previous work on the Haskell-Python project, his help with RPython problems, and finally his valuable feedback on the content of this paper. I also wish to thank my advisor Magnus Lie Hetland and my brother Gøran Wiik Thomassen for their feedback on this paper. Finally I wish to thank the PyPy community[1], which helped me whenever I was unable to decode error messages given by the RPython translation toolchain.

[1] PyPy Internet Relay Chat, `#pypy` on *irc.freenode.net*.

REFERENCES

[1] Abelson, Dybvig, Haynes, Rozas, Adams, Friedman, Kohlbecker, Steele, Bartley, Halstead, Oxley, Sussman, Brooks, Hanson, Pitman, and Wand. Revised Report on the Algorithmic Language Scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, July 1991. ISSN 1045-3563.

[2] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a Step Towards Reconciling Dynamically and Statically Typed OO Languages. In *Proceedings of the 2007 symposium on Dynamic languages*, DLS '07, pages 53–64, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-868-8.

[3] Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. Loop-Aware Optimizations in PyPy's Tracing JIT. Unpublished draft, https://bitbucket.org/pypy/extradoc/src/a88377852aa3/talk/iwtc11/licm.pdf [Online; accessed 26-05-2012], December 2011.

[4] John M. Barton. The logic programming language prolog (tutorial presentation). *J. Comput. Small Coll.*, 16:67–68, March 2001. ISSN 1937-4771.

[5] Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-Tracing on VM Design and Implementation. Submitted to Science of Computer Programming, March 2012.

[6] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. Back to the Future in One Week – Implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 123–139. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-89274-8.

[7] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-541-3.

[8] Carl Friedrich Bolz, Michael Leuschel, and David Schneider. Towards a Jitting VM for Prolog Execution. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, PPDP '10, pages 99–108, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0132-9.

[9] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Runtime feedback in a meta-tracing JIT for efficient dynamic languages. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, pages 9:1–9:8, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0894-6.

[10] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijałkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '11, pages 43–52, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0485-6.

[11] Carl Friedrich Bolz, Sebastian Fischer, and Jan Christiansen. Haskell-Python bitbucket.org project; public source repository. http://bitbucket.org/cfbolz/haskell-python/, 2011. [Online; accessed 27-04-2012].

[12] Camillo Bruni and Toon Verwaest. PyGirl: Generating Whole-System VMs from High-Level Prototypes Using PyPy. In *Objects, Components, Models and Patterns*, volume 33 of *Lecture Notes in Business Information Processing*, pages 328–347. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02571-6.

[13] Steve Dekorte. Io: a Small Programming Language. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 166–167, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7.

[14] Beatrice Düring. Trouble in Paradise: the Open Source project PyPy, EU-funding and Agile practices. In *Proceedings of the conference on AGILE 2006*, AGILE '06, pages 221–231, Washington, DC, USA, July 2006. IEEE Computer Society. ISBN 0-7695-2562-8.

[15] Simon Marlow et al. Haskell 2010 Language Report. http://www.haskell.org/onlinereport/haskell2010/, April 2010. [Online; accessed 06-05-2012].

[16] Simon Peyton Jones et al. *Haskell 98 Language and Libraries: The Revised Report*. Journal of functional programming. Cambridge University Press, 2003. ISBN 9780521826143. [Online; accessed 07-03-2012] http://www.haskell.org/onlinereport/.

[17] Jose P. E. Fernandez. Programming Python, Part I. *Linux Journal*, 2007:2–, June 2007. ISSN 1075-3583.

[18] Andrei Homescu and Alex Şuhan. HappyJIT: a tracing JIT compiler for PHP. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 25–36, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0939-4.

[19] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7.

[20] Simon Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54396-1.

[21] John Launchbury. A natural semantics for lazy evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 144–154, New York, NY, USA, 1993. ACM. ISBN 0-89791-560-7.

[22] Simon Marlow and Simon Peyton Jones. The Glasgow Haskell Compiler. In *The Architecture of Open Source Applications*, volume II, chapter 5, pages 67–88. Independent, May 2012. ISBN 9781105571817.

[23] Will Partain. The nofib Benchmark Suite of Haskell Programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-19820-2.

[24] Benjamin Peterson. PyPy. In *The Architecture of Open Source Applications*, volume II, chapter 19, pages 279–290. Independent, May 2012. ISBN 9781105571817.

[25] Simon Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 184–201, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0.

[26] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3.

[27] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X.

[28] Juan Rodríguez-Hortalá and Jaime Sánchez-Hernández. Functions and Lazy Evaluation in Prolog. *Electronic Notes in Theoretical Computer Science*, 206(0):153–174, 2008. ISSN 1571-0661.

[29] David Schneider. Implementation of the Io language in RPython; public source repository. http://bitbucket.org/pypy/lang-io/, 2009–2011. [Online; accessed 12-03-2012].

[30] Anders Sigfridsson, Gabriela Avram, Anne Sheehan, and Daniel Sullivan. Sprint-driven development: working, learning and the process of enculturation in the PyPy community. In *Open Source Development, Adoption and Innovation*, volume 234 of *IFIP International Federation for Information Processing*, pages 133–146. Springer Boston, 2007. ISBN 978-0-387-72485-0.

[31] Knut Halvor Skrede. Just-In-Time compilation of Haskell using PyPy and GHC. http://github.com/khskrede/mehh, December 2011. Project report at NTNU, Trondheim. [Online; accessed 28-04-2012].

[32] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '07, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. Version from January 2011.

[33] Andrew Tolmach, Tim Chevalier, and The GHC Team. An External Representation for the GHC Core Language. http://www.haskell.org/ghc/docs/7.4.1/core.pdf, February 2012. [Online; accessed 01-05-2012].

[34] Laurence Tratt. Fast Enough VMs in Fast Enough Time. http://tratt.net/laurie/tech_articles/articles/fast_enough_vms_in_fast_enough_time, February 2012. [Online; accessed 03-03-2012].

[35] David Michael Ungar. *The Design and Evaluation of A High Performance Smalltalk System*. PhD thesis, University of California, Berkeley, Berkeley, CA, USA, 1986.

[36] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors — A compiler pearl. (draft submitted to ICFP 2012) http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/icfp12.pdf [Online; accessed 05-05-2012], March 2012.

[37] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1120-5.

APPENDIX A
SOURCE CODE

This appendix contains source code listings of some implementations I have contributed to the PyHaskell VM, and some Haskell tests and benchmarks.

A.1 Primitive function decorators

Listing 1 on page 42 shows the source code of the implementation of the RPython primitive function decorators. Listing 2 on page 43 shows the implementation of the Haskell `cons` operator, which use a primitive function decorator on line seven. Listing 3 on page 43 shows an alternative implementation of the `cons` operator without using a meta-programming decorator. These code-listings are described in subsection 4.4.2 on page 22.

A.2 Functions from built-in modules

Listing 4 on page 44 is our implementation of the unpackCString function that converts a C-like string into a head-tails list of Haskell Chars. Listing 5 on page 44 contains our implementation of `putStrLn`, which prints a list of Chars to standard out. Listing 6 on page 45 contains the source code of our implementation of list concatenation. These listings are explained further in subsection 4.4.6 on page 24.

```
def expose_var(module, name, arg_types):
    return _expose(module.qvars, name, arg_types)

def expose_data_constr(module, name, arg_types):
    return _expose(module.qdcons, name, arg_types)

def expose_type_constr(module, name, arg_types):
    return _expose(module.qtycons, name, arg_types)

def _expose(storage, name, arg_types):
    """NOT_RPYTHON"""
    def decorator(function):
        """NOT_RPYTHON"""
        arity = len(arg_types)

        def wrapped_f(args):
            # Check that arguments are instances of valid classes
            assert len(args) == arity
            for i in range(arity):
                assert isinstance(args[i], arg_types[i])

            # Unpack the args list and call self.function
            args_tuple = specialized_unpack_function[arity](args)
            return function(*args_tuple)

        func = PrimFunction(name, wrapped_f, arity, [True] * arity)
        storage[name] = func
        return func
    return decorator

def _make_unpack(n):
    """NOT_RPYTHON"""
    if n == 0:
        return lambda args: tuple()
    return eval('lambda args: (%s,)' %
            ','.join(['args[%i]' % i for i in range(n)]))

specialized_unpack_function = [_make_unpack(i) for i in range(11)]
```

Listing 1: Implementation of primitive function decorator

```
from pyhaskell.interpreter.module import CoreMod, expose_data_constr
from pyhaskell.interpreter.haskell import (AbstractFunction, Symbol,
            Value, evaluate_hnf, make_application, make_constructor)

mod = CoreMod("ghc-prim:GHC.Types")

@expose_data_constr(mod, ":", [Value, Value])
def cons(a, b):
    """Cons operator ':', for creating lists."""
    if isinstance(b, AbstractFunction):
        b = evaluate_hnf(make_application(b, []))
    return make_constructor(Symbol.get_symbol(":"), [a, b])
```

Listing 2: Cons implementation that use a primitive decorator

```
from pyhaskell.interpreter.module import CoreMod
from pyhaskell.interpreter.haskell import (AbstractFunction, Symbol, Value,
            PrimFunction, evaluate_hnf, make_application, make_constructor)

mod = CoreMod("ghc-prim:GHC.Types")

def cons(args):
    """Cons operator ':', for creating lists."""
    a, b = args
    assert isinstance(a, Value)
    assert isinstance(b, Value)

    if isinstance(b, AbstractFunction):
        b = evaluate_hnf(make_application(b, []))
    return make_constructor(Symbol.get_symbol(":"), [a, b])

cons = PrimFunction(':', cons, 2, [True, True])

mod.qdcons[':'] = cons
```

Listing 3: Cons implementation without meta-programming decorator

```
from pyhaskell.interpreter.module import CoreMod, expose_var
from pyhaskell.interpreter.haskell import make_constructor, Symbol
from pyhaskell.interpreter.primtype import Char, Addr

mod = CoreMod("ghc-prim:GHC.CString")

@expose_var(mod, "unpackCString#", [Addr])
def unpackCString(a):
    """Convert an Addr (Python string) into a Haskell list of chars."""
    current = make_constructor(Symbol.get_symbol("[]"), [])
    cons = Symbol.get_symbol(":")
    str_ = a.tostr()
    for i in range(len(str_) - 1, -1, -1):
        current = make_constructor(cons, [Char(str_[i]), current])
    return current
```

Listing 4: Implementation of unpackCString.

```
import sys
from pyhaskell.interpreter.module import CoreMod, expose_var
from pyhaskell.interpreter.haskell import Value

mod = CoreMod("base:System.IO")

@expose_var(mod, "putStrLn", [Value])
def putStrLn(a0):
    """Print a list of Chars to stdout."""
    t = a0
    str_ = ''
    while t.numargs() > 0:
        str_ += t.getarg(0).tostr()
        t = t.getarg(1)
    print str_
    return a0
```

Listing 5: Implementation of "base:System.IO.putStrLn"

```python
from pyhaskell.interpreter.module import CoreMod, expose_var
from pyhaskell.interpreter.haskell import Value, make_constructor, Symbol

mod = CoreMod("base:GHC.Base")

@expose_var(mod, "++", [Value, Value])
def concatenation(a, b):
    """List concatenation, ++ operator."""
    stack = []
    current = a
    while current.numargs() > 0:
        stack.append(current.getarg(0))
        current = current.getarg(1)

    cons = Symbol.get_symbol(":")
    current = b
    for i in range(len(stack) - 1, -1, -1):
        current = make_constructor(cons, [stack[i], current])
    return current
```

Listing 6: Implementation of list concatenation operator '++'

A.3 Haskell tests

Listing 7, Listing 8, and Listing 9[1] contain the source code of some tests we have implemented. The pattern matching test fails on PyHaskell as we do not yet support that feature.

```
main = putStrLn ("Hel" ++ "lo, " ++ "world!")
```

Listing 7: Haskell test of list concatenation '++' operator

```
main = putStrLn ('H':'e':"llo, world!")
```

Listing 8: Haskell test of cons ':' operator

```
main = putStrLn $ capital "Dracula"

capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

Listing 9: Haskell test of pattern matching

A.4 Benchmarks

Listing 10 show the naive Fibonacci sequence[2] algorithm implemented in Haskell. Listing 11 list the source code of a benchmark similar to Fibonacci, but with multiplication instead of addition. Listing 12 shows part of the code of the case benchmark, which iteratively calls a function with a case expression 200 times. Most of the function calls have been removed to allow the code to fit on one page, but the removed code lines are almost identical to the surrounding lines.

[1] Pattern matching test taken from:
http://learnyouahaskell.com/syntax-in-functions#pattern-matching

[2] Naive Fibonacci implementation taken from:
http://www.haskell.org/haskellwiki/The_Fibonacci_sequence

```haskell
main = putStrLn (show (fib 31))

fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Listing 10: Naive Fibonacci sequence benchmark

```haskell
main = putStrLn (show (mult 25))

mult :: Int -> Int
mult 1 = 1
mult 2 = 1
mult 3 = 1
mult n = mult (n-1) * mult (n-2) * mult (n-3)
```

Listing 11: Multiply recursive benchmark

```haskell
data Option = Something Int | Space | Question

case1 :: Option -> String
case1 (Something 1) = "one"
case1 (Something n) = show n
case1 (Space) = " "
case1 (Question) = "?"

main = putStrLn (
        (case1 (Something 1)) ++
        (case1 Space) ++
        (case1 (Something 2)) ++
        (case1 (Something 3)) ++
        -- 197 lines removed
        (case1 (Something 200)) ++
        (case1 Space) ++
        (case1 Question))
```

Listing 12: Iterative case benchmark