NTNU – Trondheim
Norwegian University of
Science and Technology

Trace-based just-in-time compiler for Haskell with RPython

Even Wiik Thomassen

Master of Science in Computer Science
Submission date: Januar 2013
Supervisor: Magnus Lie Hetland, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

Can Haskell benefit from tracing JIT optimization techniques, and is the RPython translation toolchain suitable for purely functional, lazy languages such as Haskell? RPython has been used to implement VMs for many different programming languages, but not for any purely functional or lazy languages. Haskell has achieved impressive speed with ahead-of-time optimizations. Attempts at trace-based JIT optimizations of Haskell have so far not achieved greater speed than static compilation. PyHaskell, a prototype Haskell VM with a meta-tracing JIT compiler written in RPython, shows that the RPython toolchain is suitable for Haskell. While the meta-tracer greatly speeds up PyHaskell, it does not yet beat GHC.

Oppsummering (Norwegian abstract)

Kan Haskell bli raskere med hjelp av trace-baserte JIT optimiseringsteknikker, og er RPython translation toolchain egnet for rent funksjonelle og "late" språk som Haskell?

RPython har blitt brukt til å lage virtuelle maskiner for mange forskjellige programmeringsspråk, men ikke for noen rent funksjonelle eller "late" språk. Haskell har oppnådd imponerende hastighet med statiske optimaliseringer, mens forsøk på trace-baserte JIT optimaliseringer har vært mislykket.

PyHaskell, en virtuell maskin-prototype for Haskell med en meta-tracing JIT-kompilator skrevet i RPython, viser at RPython toolchain er egnet for Haskell. Selv om meta-traceren øker PyHaskells hastighet kraftig, kan den enda ikke slå GHC.

Acknowledgments

I first and foremost wish to thank Dr. Carl Friedrich Bolz for his help with optimizing PyHaskell and making sense of the RPython translation toolchain, which at times has very limited documentation. And for his previous work, as this thesis builds upon it, not only in the Haskell-Python project, but also the PyPy project; his many papers and research on meta-tracing just-in-time compilation. And finally I wish to express gratitude for his valuable feedback on the content of this report.

I would also like to thank Sebastian Fisher, Jan Christiansen, and Knut Halvor Skrede for their previous work on the Haskell-Python project.

Finally I would like to convey thanks to: my supervisor, Dr. Magnus Lie Hetland for his guidance and advice; my brother, Gøran Wiik Thomassen for his feedback on this report; and the PyPy community for their work and advice.

Contents

CHAPTER 1

Introduction

Both dynamic and functional programming languages have in recent years increased greatly in popularity. The functional language Haskell is home to exciting language research, while the PyPy project is a very interesting development on the dynamic language front. The PyPy project has built an environment for creating dynamic virtual machines (VMs) in a high-level language called RPython.

Haskell has received much research on improving its execution speed, almost all of it as static compilation techniques; most as high-level transformations that take advantage of its functional nature [25]. The PyPy project has however directed its efforts on improving the execution speed of dynamic languages, with the help of trace-based just-in-time (JIT) compilation [40]. The PyPy environment is called the RPython translation toolchain, which include a meta-tracing JIT that can be reused by any VM written in RPython [7].

The RPython meta-tracer has been used successfully on dynamic languages such as Python, Prolog, PHP, R, and JavaScript [8, 17, 18, 26]. The toolchain is very effective on object-oriented languages and the impressive results of the PyPy project motivates us to test if the RPython toolchain can be suited for purely functional, lazy languages, e.g., Haskell [26]. We believe the meta-tracing JIT can compete with state of the art ahead-of-time Haskell compilers. While Haskell is heavily optimized at compile-time, more information is available at runtime that a JIT compiler may exploit. This report therefore set out to answer two research questions:

- *Is the RPython translation toolchain suitable for purely functional and lazy languages, e.g., the Haskell language?*
- *Can Haskell benefit from trace-based JIT optimization techniques?*

Trace-based optimization of Haskell is a very recent development, only after the start of this report have two papers been published. Peixotto [25] has attempted a trace-based binary optimization technique with DynamoRIO, which was abandoned as the approach added too much overhead from just finding traces. Peixotto also attempted a trace-based ahead-of-time approach with LLVM, which was able to achieve an average speed up of 5% [25]. Schilling [38] has created a tracing JIT prototype that adopt ideas from LuaJIT 2 that is still in development, but with promising early results. This report describe the first attempt at optimizing Haskell with a meta-tracing JIT compiler.

PyHaskell, a new prototype backend for the Glasgow Haskell Compiler (GHC) has been created. GHC desugar Haskell to the Core language, hence PyHaskell is a VM for the Core language. As PyHaskell is written in RPython, it includes the RPython meta-tracing JIT compiler. PyHaskell's runtime performance will determine the answer to the two research questions stated in this report.

If the performance of our VM is reasonable, it will show that the RPython translation toolchain is suited for purely functional, lazy languages, and if PyHaskell can beat GHC on some benchmarks, Haskell should benefit from trace-based JIT optimizations.

Most programming paradigms have already been implemented in RPython, e.g., object-oriented, declarative, imperative, procedural, and more. Purely functional and lazy are needed to round out the paradigms the RPython toolchain has been tested on, hence the need for PyHaskell.

A goal for PyHaskell was *to support both the GHC test suite and the nofib benchmark suite*. This would require supporting the Haskell standard library, named Haskell Prelude. GHC can serialize the Core intermediate representation into a parsable syntax. Unfortunately, this functionality has not been sufficiently maintained. Therefore PyHaskell cannot reuse GHC's Prelude, and the goal mentioned above was not achieved.

1.1 Summary of contributions

The report starts by describing Haskell and GHC in chapter 2. It explains concepts such as PyPy, RPython, and meta-tracing JIT compilation in chapter 3. Issues that prevent PyHaskell from supporting the Haskell Prelude, and two viable solutions to these issues are described in chapter 4. Related works are described in chapter 11.

PyHaskell description and improvements After GHC has produced an external representation of Core, the `core2js` program converts it to JSCore, a JavaScript Object Notation (JSON) representation. The PyHaskell VM then parses JSCore to create constructs based on semantics for lazy languages designed by Launchbury [21]. These constructs are then evaluated, which represent the execution of Haskell code. Detailed description of PyHaskell can be found in chapter 5.

To be able to support more advanced benchmarks, PyHaskell required a number of improvements, which are presented in section 5.5. PyHaskell's support of the Haskell language has been summarized in section 5.6.

RPython hints The RPython meta-tracing JIT compiler can perform some JIT optimizations techniques, which are enabled with hints placed in the source code of VMs. These techniques can improve the performance of PyHaskell, but they are badly documented. Some hints are described in a few research papers [7, 9]; some in blog posts [4], but many were not documented at all. Chapter 6 therefore document the RPython hints and optimization techniques, how they can be used and for what purpose.

PyHaskell optimizations RPython can produce logs of JIT-traces, which can be used to spot optimization opportunities. This feature of RPython is not documented, hence chapter 7 starts with a description of RPython JIT trace-logs. Optimizations are crucial for PyHaskell's performance to reach GHC's. Chapter 7 present the efforts undertaken to optimize PyHaskell. The effects of individual types of hints on PyHaskell are summarized in section 7.4.

PyHaskell benchmark and evaluation A collection of benchmarks have been created to answer the research questions asked in this report. The benchmarks and their results are described in Chapter 8, and are further discussed in Chapter 9. Chapter 10 contains the conclusion and further work.

Utility As mentioned, parts of the RPython translation toolchain is not documented, especially the parts regarding optimization with source code hints. Therefore I believe this report (chapter 6 and chapter 7) should prove useful for anyone who wish to write a VM in RPython.

CHAPTER 2

Haskell and GHC

This chapter describes: The Haskell programming language (section 2.1); GHC, the main implementation of Haskell (section 2.2); and Core, a GHC intermediate language (section 2.3).

2.1 Haskell language

Haskell is a purely functional, lazy programming language. Purely functional means functions cannot have side effects or mutate data. Lazy means evaluation is delayed until required, e.g., function arguments are passed unevaluated and then evaluated on demand. Delaying evaluation comes with several costs, e.g., being less efficient and making it hard to predict space behavior of programs. Purity is pretty much required for lazy languages, but I/O is very clumsy in pure languages, so monadic I/O was invented for Haskell to solve this problem [19]. A monad is a powerful abstraction that consists of a type constructor and a pair of functions: return and bind. Monads provide syntax to express structured procedures that are not otherwise supported by functional languages. Haskell also introduced type classes to support overloading of built-in numeric operators. Type classes define behavior of types (which behave similar to Java's interfaces) to provide support for ad-hoc polymorphism [19, 22].

Other notable Haskell features include pattern matching, list comprehensions, and type inference. Type inference means type annotations are rarely needed despite Haskell being statically typed. Haskell's list comprehensions have been adopted by both Python and JavaScript [19, 22, 35].

Type classes and Monads are perhaps Haskell's most distinctive design features, which have influenced many other programming languages. For example: Scala, Coq, and Rust have adopted both monads and type classes, while C#'s Language

Integrated Query (LINQ) was directly inspired by Haskell's monad comprehensions [19].

Haskell was created to consolidate more than a dozen similar functional languages that was started in late 1970s and early 1980s. A meeting in January 1988 defined six goals for Haskell [19, p. 4]:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be permitted to implement the language and distribute it to whomever they please.
4. It should be usable as a basis for further language research.
5. It should be based on ideas that enjoy a wide consensus.
6. It should reduce unnecessary diversity in functional programming languages.

The second goal was not realized as Haskell's syntax and semantics have never been formally described. The plan for the last goal was to base Haskell on an existing functional language called OL. This plan was abandoned early. Haskell has successfully achieved most of the remaining goals, although some features such as type classes were added without regard for goal five [19].

As Haskell is designed by committee, it is a rather big language, and there are usually more than one way to do something in Haskell. As the language grew it also quickly evolved, which was problematic for teaching and application that require stability. The committee consequently defined a stable version of the language, "Haskell 98", that implementations committed to support indefinitely. In 2005 design of Haskell´ (pronounced Haskell Prime) was started, to succeed Haskell 98 and to cover heavily used extensions [19]. Haskell 2010 is the latest stable version of the Haskell programming language [23].

2.2 The Glasgow Haskell Compiler

Glasgow Haskell Compiler (GHC) is the most fully featured Haskell compiler today, and has been the main Haskell implementation since the release of "Haskell Report 1.0" in 1990 [19]. Marlow and Peyton Jones states that today "GHC releases are downloaded by hundreds of thousands of people, the online repository of Haskell libraries has over 3,000 packages, GHC is used to teach Haskell in many undergraduate courses" [22, p. 1].

GHC can be divided into three distinct parts:

- The compiler, a Haskell program that converts Haskell source code to machine code.
- The boot libraries that the compiler depend on.

- The runtime system. Large library of C code that handles running the compiled Haskell code, dealing with important runtime tasks such as garbage collection, threads, and exceptions. [22]

The compiler itself is also divided into three parts:

- The compilation manager, which manages compilation of multiple Haskell files, the order of compilation, and check which modules require recompilation.

- The Haskell Compiler, which compiles a single Haskell file to machine code, depending on the selected backend.

- The pipeline, which handles Haskell code that interface with or require external programs. For example if a source file require preprocessing with a C preprocessor. [22]

The compiler is also a library, called GHC application programming interface (API), which provides access to internal parts of GHC and allows working with Haskell code.

2.2.1 Compilation process

GHC's compilation process is a linear process divided into the frontend and backend with several optimization passes in the middle. Figure 2.1 provides an overview of this process [22].

The frontend starts with *parsing* Haskell source files with a fully functional parser that produces an abstract syntax tree (AST) where identifiers are simple strings. The second step is called *renaming*, where identifiers are turned into fully qualified names. This step also spots duplicate declarations, rearrange infix expressions, collect the equations of functions together, and more. The third step of frontend process is *type checking*. Any program that passes the type checker is type-safe and guaranteed to not crash at runtime. The last step of the frontend is *desugaring*, where all "syntactic sugar" is removed and the full Haskell syntax is converted into a much smaller intermediate language called Core. Core is further described in section 2.3.

The *optimization* stage, which joins the frontend and the backend process, consists of several passes where code represented as Core is transformed into more optimized code (which is still Core). These optimizations, and other optimizations performed by GHC, are described in subsection 2.2.2.

After the *optimization* stage the code can either be turned over to the backend to generate low-level code or it can be turned into bytecode. The bytecode is used by the interactive Haskell interpreter, GHC's interactive environment (GHCi). Before code generation, Core is transformed into another intermediate representation called Spineless Tagless G-machine (STG). STG is an A-normalized lambda calculus that defines GHC's execution model [31].

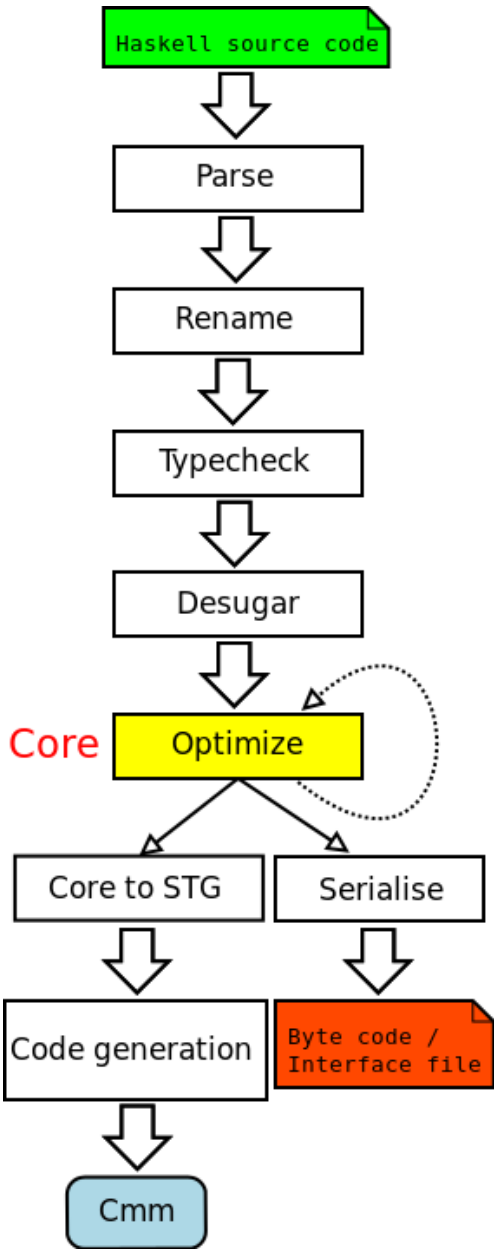

Figure 2.1: GHC's compilation process

The next step is the actual *code generation*, which converts STG into another intermediate representation, C minus minus (Cmm). Cmm is a variant of the C-- language, and is almost a subset of C that support proper tail calls [22, 47]. All remaining non-strict and functional aspects of Haskell are removed during code generation, which allow Cmm to be a simple intermediate language. Cmm is finally the input for one of three object code generating backends:

- The C code generator, which pretty-print Cmm to C code. The C code is then compiled with GNU Compiler Collection (GCC). Is very portable, as it can be used on most architectures that support GCC, but the produced code is not as fast as the other two backends and the compilation process is significantly slower. The C backend was deprecated around GHC 7.0.
- The native code generator (NCG) that only support a few architectures, but produces faster code than the C backend.
- The LLVM code generator, which produce LLVM intermediate representation (IR) that is compiled with LLVM. This backend is described in more detail in section 11.1 on page 55.

An overview over the process of these code generator backends can be seen in Figure 2.2 [22, 47].

2.2.2 Optimizations

GHC starts the optimization process after syntactic sugar is removed and Haskell is transformed to Core. As can be seen in Figure 2.3, there are several different optimization steps, and the simplifier is run in between. The simplifier applies a lot of small, local optimizations such as let-to-case transformation and case elimination. The actual steps taken depend on optimization level specified, where one can trade compilation speed for generated code with is faster [22, 32]. During these steps GHC performs traditional optimizations such as common sub-expression elimination, unboxing, inlining, and more [29].

The strictness analyzer finds variables and arguments that can be treated strictly, which enable optimizations such as unboxing that would not be allowed for lazy arguments [30]. Let-floating moves let bindings closer to where they are used, which avoids unnecessary allocations if they are on a branch that is never executed [33]. Constructor specialization enables specialization based on call-patterns, which specialize recursive functions according to their argument shapes [27].

GHC also do some optimizations at later stages of the compilation process. For example *code generation* include the tables-next-to-code (TNTC) optimization, which places meta-data of closures right before the code for the closure. TNTC allow accessing both closure meta-data and code from a single pointer [25, 47].

2.2.3 Extensibility

GHC support extensibility in several ways, the most significant is probably with the GHC API. In other words, GHC has been built as a Haskell library, and the

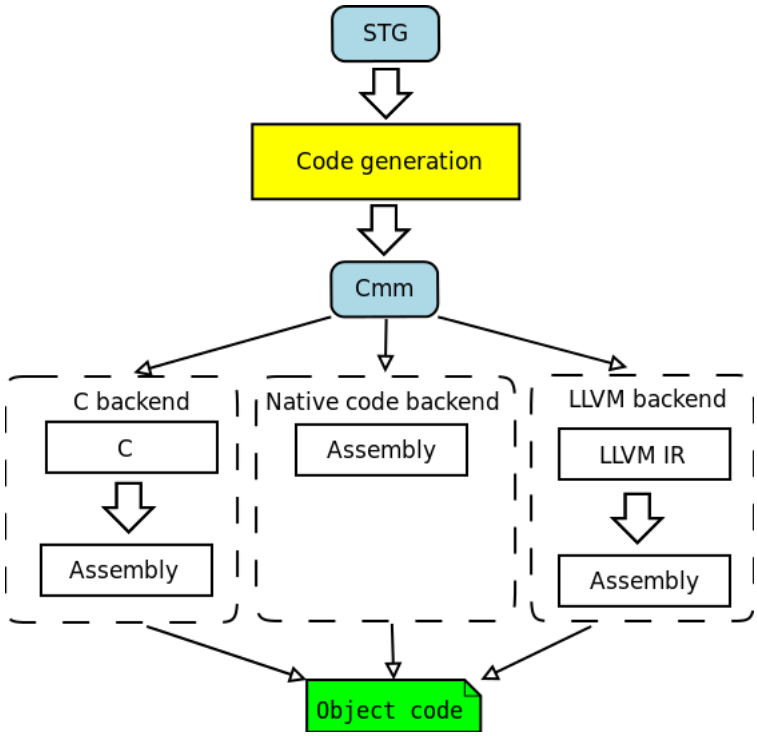

Figure 2.2: Code generator backends included with GHC

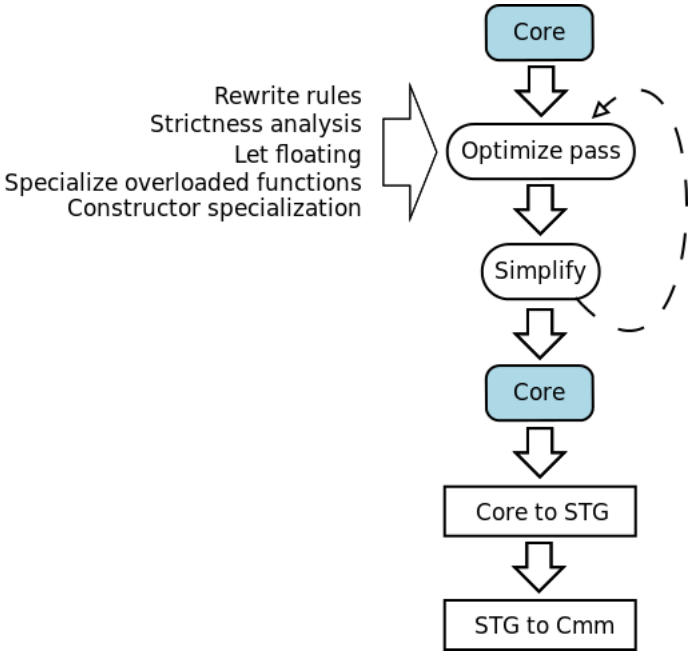

Figure 2.3: Optimizations steps performed by GHC

GHC executable is just a small Main module linked with this library. GHC's functionality is exposed through an API, which provide access to: the steps of the compilation process; data structures; and intermediate representations.

Another extensibility of GHC is external Core, which is a runtime option for GHC to serialize Core into an external human-readable representation. This option is further described in chapter 4 on page 19.

2.3 Core intermediate language

While Haskell is a very large implicitly-typed language, it can be fully translated into Core, an explicitly- and statically-typed intermediate language.

The theory behind Core has changed over the years, and the Core name is used for the implementation of the intermediate language in GHC. Currently Core is the implementation of System $F_C^{\uparrow}$ (which have been recently defined formally in a technical paper by Eisenberg [15]).

Core was initially based on lambda calculus. To be able to decorate Core with types, it was upgraded to a polymorphic lambda calculus, System F_ω, and then extended with data types, let-expressions, and case expressions [19]. System F_ω is based on System F, which was originally developed as a foundational calculus for typed computation. To support type equality constraints and safe coercions, Core was further extended to System F_C [44, 48, 50]. System F_C also provide simple

support for `kinds` (the type of a type). Weirich et al. [51] introduced System FC_2, which simplified System F_C and decorated `kinds` with roles to mark type contexts. System $F_C^{\uparrow}$ is Core's most recent upgrade, which has more complex `kinds` that provide better support for `type families` and `generalized algebraic data types (GADT)` [50, 52].

CHAPTER 3

RPython and PyPy

This chapter describes concepts related to JIT compilation and RPython, such as: The PyPy project (section 3.2); RPython language and RPython translation toolchain (section 3.4); and trace-based JIT compiling (section 3.5).

3.1 Python

Python is a high-level, dynamic programming language that supports several programming paradigms, such as imperative, object-oriented and functional [5, 26]. It is highly regarded for its simplicity and ease of use [16].

The original Python implementation, CPython, is a traditional bytecode interpreter written in C [26]. While several alternative implementations have been created since then, CPython is still the official or standard Python implementation [41]. Other well-known Python implementations are [26]:

- Jython[1], which is written in Java and runs on Java Virtual Machine (JVM).
- IronPython[2], which is written in C# and interfaces with Microsoft's .NET framework.
- PyPy, a Python interpreter implemented in Python itself.

3.2 PyPy project

The PyPy project consist of two major components: The Python interpreter PyPy and the RPython translation toolchain. PyPy is built with the help of the RPython

[1] Jython homepage: `http://www.jython.org/`
[2] IronPython homepage: `http://ironpython.net/`

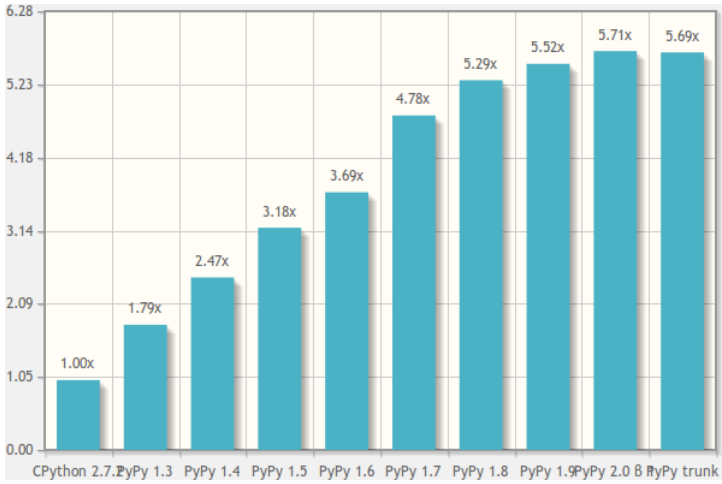

Figure 3.1: PyPy interpreter speed evolution [45]

translation toolchain [26]. These two components map directly to the two main goals of the PyPy project:

- Provide a faster Python implementation [5, 14].
- Be an environment for implementing fast, complex dynamic languages that support multiple platforms [7, 36, 40].

Traditionally one must implement one VM for each platform one wish to support. By implementing the PyPy VMs in a high level language it can support several very different platforms with one implementation [36]. High level languages also help keep the implementation free of low-level details such as object layout, threading model, and memory management [7]. The RPython toolchain used by PyPy achieve the second goal. The disadvantage of a VM implemented in a high-level language is lack of speed, so to achieve the first goal PyPy has developed a meta-tracing JIT compiler that is a part of the RPython translation toolchain [5, 9, 36]. Meta-tracing is explained in section 3.5.

The PyPy interpreter has achieved its goals. PyPy is a fully compatible alternative Python interpreter that is quite fast. Peterson writes that "PyPy is a geometric average of five times faster than CPython on a comprehensive suite of benchmarks" [26, p. 287]. The current and historic status of the speed difference of PyPy against CPython 2.7.2 can be seen in Figure 3.1[3] [45].

RPython has been used to implement a number of programming languages. An overview over these implementations can be seen in Table 3.1[4]. Only the PyPy

[3]PyPy speed progress (Figure 3.1) is taken from `http://speed.pypy.org`, which also has performance result of each benchmark.

[4]Hippy VM RPython implementation: `http://bitbucket.org/fijal/hippyvm/`
RPython JavaScript implementation: `http://bitbucket.org/pypy/lang-js/`
RPython Scheme implementation: `http://bitbucket.org/pypy/lang-scheme/`

interpreter and Tratt's Converge [5] are considered complete, while the rest are in different stages of development or abandoned as they have fulfilled their goals. A few of them does not include RPython's JIT compiler [6, 8, 12, 17, 18, 39].

Table 3.1: Overview over virtual machines implemented with RPython

			Development		
VM Name	Language	JIT	Started	Stopped	Status
Converge	Converge	Yes	2011		Complete
HappyJIT	PHP	Yes	2011		Partial
Hippy VM	PHP	Yes	2012		Partial
io	Io	No	2009	2011	Partial
js	JavaScript	Yes	2006		Partial
PyGirl	Game Boy	No	2008	2009	Almost complete
PyHaskell	Haskell	No	2011		Partial
PyPy	Python	Yes	2004		Complete
Pyrolog	Prolog	Yes	2006		Almost complete
Rapydo	R	Yes	2012		Partial
Spy	Smalltalk	No	2007	2011	Almost complete
Scheme	Scheme	No	2006	2012	Partial

3.3 Just-in-time compilation

A just-in-time (JIT) compiler converts code to compiled machine code dynamically at runtime, in contrast to static compilation, which compile code to machine code beforehand. A JIT system must limit the time it uses on optimization during compilation, as execution is paused during compilation. Static compilation can devote as much time as they want on program analysis and optimization. JITs can in some instances provide better performance than static compilation, as it has access to runtime information, e.g., control flow, input parameters, and specifics of the target machine [3].

JIT compilation can be divided into two categories, method-based and trace-based. Method-based JIT compilers convert code one method at a time, while trace-based JIT compilers compile frequently executed loops to machine code. Tracing JITs can greatly speed up programs if they spend most of their time in loops where they take similar code paths [7]. Tracing JITs are further explained in section 3.5.

Trace-based JIT has been used with success by both dynamically and statically typed languages, such as Python [7], Lua, JavaScript, Java, and C# [38].

3.4 RPython

The PyPy project has defined a subset of the Python language as the RPython language, short for Restricted Python. RPython is a statically typed object-oriented programming language. The PyPy interpreter is written in RPython, and other

VM that wish to use the RPython environment must also be written in RPython. The toolchain itself is also written in RPython [40].

As RPython is a strict subset of the Python language and the RPython translation toolchain is run on top of a Python interpreter, RPython VMs can also be run on Python interpreters. This allows debugging with Python tools and quick testing without translating with the toolchain [26].

The RPython language is selected in such a way that it is possible to do type inference on it [9]. The RPython language is not defined formally, 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]. These restrictions are mainly:

- Variables need to be type consistent, for example a variable cannot hold an integer and then later a string [26].
- Types of all variables in the code must be inferable [12].
- Functions cannot be created at runtime [26].
- Bindings in classes and global namespaces are assumed to be constant [36].
- Runtime reflection is not supported [12].

Despite its restrictions, RPython is a high-level language that supports: garbage collection; single inheritance[5]; exceptions; 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].

One goal behind the RPython translation toolchain was to allow compilation of RPython programs to various environments, such as C, JVM, and Common Language Runtime (CLR) [7, 36]. Another goal was to automatically create VMs with JITs through meta-tracing [5].

The toolchain performs step wise translations from the RPython source of a VM until it reaches the low-level code of a 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 [36].

The translator toolchain starts with an *abstract interpretation* phase, which builds flow-graphs from RPython source code. The flow-graphs consist of linked blocks where each block has input arguments and a list of operations. These flow-graphs are in static single assignment (SSA) form [26].

Next is the *annotation* phase, where type information is assigned to the arguments and result of each operation. The annotation phase performs type inference on the whole program [26].

RTyping is the next phase, which uses type information to expand high-level flow-graph operations into low-level operations. After RTyping several optimizations are performed on the low-level flow-graphs. Traditional optimizations such

[5]RPython support most of the advantages of multiple inheritance with explicitly declared mixins [1, 6].

as constant folding, dead code removal and more complex ones such as function inlining and malloc removal [26].

The final phase is the *backend*, which generates source code from low-level flow-graphs. The C backend emits C code, but must first add explicit garbage collection and exception handling. The generated C code is compiled as the final step of the C backend. There is an alternative object-oriented backend for generating code that runs on JVM and CLR [7, 26].

Writing RPython code can be time-consuming and challenging as there is little documentation on RPython, how to use it and what is valid RPython. To discover if an operation is valid RPython, one must translate it with the toolchain. Invalid code will generate error messages, which can be quite cryptic and contain a limited amount of useful information. During my work I discovered three major RPython restrictions that were not explicitly detailed by PyPy research papers:

- Only a subset of built-in functions is available.
- Only a subset of built-in data structures is available.
- Python's standard library is not available as RPython. RPython has its own standard library, `pypy.rlib` sub-package, where a very limited part of Python's library is implemented in RPython.

3.5 RPython's meta-tracing just-in-time compiler

As mentioned in section 3.3, JITs improves the speed of a language by compiling frequently used code-paths into assembly at runtime [26]. Many dynamic languages have two main challenges that a JIT might solve: a) overhead from the interpreter, for example bytecode dispatch and interpreter's data structures; and b) overhead from boxing of primitive types [10].

A tracing JIT observes the running program to detect commonly executed concrete paths [9, 10]. Detected paths are called a trace, and contains the history of executed operations. A trace is first optimized with well-known compiler optimizations, before compilation into machine code. As a trace is linear, many optimizations are easy to do and generating machine code for the trace is straightforward [2].

RPython's JIT optimizer does "a few classical compiler optimizations and many optimizations specialized for dynamic languages" [26, p. 285]. *Virtuals* and *virtualizables* are among the most important optimizing techniques according to Peterson [26]. Virtuals are objects that can be stored in registers and on the stack without allocation. Virtualizables are similar to virtuals, except virtualizables may escape the trace so they must be handled during tracing, while virtuals are handled during trace optimization [26].

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 that returns control back to the VM. If a guard fails often, the tracing JIT will start a new trace from the failed guard [10]. Schneider and Bolz have found that in the context of RPython's

JIT, "guards account for about 14% to 22% of the operations before and for about 15% to 20% of the operations after optimizing the traces generated for different benchmarks" [40, p. 1].

While a tracing JIT compiler normally traces the user program that runs on top of an interpreter, RPython's JIT traces the execution of the interpreter itself, which is why it is called a meta-tracing JIT. RPython's meta-tracing JIT can be used, almost automatically, by all VMs written in RPython. Only two hints in the source code of a VM is required by the meta-tracer: `can_enter_jit` and `merge_point`. The former hint specifies where in the interpreter a loop starts, and the latter says where it is safe to return to the interpreter from the JIT [5, 7, 26]. There are also other optional hints one can use to help improve the performance of the meta-tracing JIT. RPython hints are described further in chapter 6 on page 33.

CHAPTER 4

External Core

This chapter describes GHC's current implementation of external Core (section 4.1); issues and limitations hindering PyHaskell's support of the Haskell language (section 4.2; and possible solutions to these issues section 4.3.

PyHaskell use GHC to type-check and desugar Haskell source code into Core, and to create an external representation of Core. PyHaskell then convert external Core into a JSON representation, which our can VM parse and evaluate. PyHaskell's pipeline is further described in section 5.3 on page 27.

4.1 Current implementation

External Core is an external representation of the Core data structure used in GHC's compilation process[1]. As explained in section 2.2, GHC desugar Haskell to Core, after which GHC's main optimizations are performed as semantic-preserving transformations on Core. The Core language, which the Core data structure implements, is described in section 2.3.

External Core is designed as a round-trip in GHC, where Core is first converted to an external Core data type. This data type is then printed into the concrete syntax of external Core. The syntax is parsable by a parser that converts external Core into the Iface Core data type. GHC can then convert Iface Core into Core again [28].

Identifiers in external Core is Z-encoded, according to the rules in Table D.1, as explained in Appendix D on page 73.

External Core representation of a Haskell source file is produced by giving GHC

[1] GHC's compilation process can be seen in Figure 2.1 on page 8.

the flag `-fext-core`, and the *extcore*[2] package support parsing and working with external Core in Haskell.

4.2 Issues and limitations

While Core has been improved and upgraded to different iterations based of System F (currently System $F_C^{\uparrow}$), external Core has been neglected and drifted out of sync with Core.

Eisenberg [15] have presented a formal definition of Core and System $F_C^{\uparrow}$ for the latest GHC version, 7.6. Tolmach et al. [48] describe a precise definition of external Core that was last updated for GHC version 6.10. The time period between these two versions are about four years. In these four years Core has seen significant work, while external Core has only received minor improvements. For example, external Core does not support GADT, type families, integer literals, and left and right coercion.

It is not only external Core that causes problems for PyHaskell. The *extcore* package has been abandoned, and was last updated over a year ago. According to HackageDB[2], Haskell's online package library, *extcore* support GHC version 7.0 or older. Therefor there is some things external Core support that is not supported by *extcore*.

The *extcore* package also has some unwanted behavior. For example, it convert lambda-statements that have more than one variable into several lambda-statements with one variable each.

The `core2js`-program in Skrede [43]'s pipeline also has some issues, such as the JSON-encoding of external Core is too verbose. It is more verbose than what is required to encode it as JSON.

Appendix C list an example where *extcore* split a lambda into two lambdas. Line six and seven of Listing C.2 is a two-variable lambda, which is two single-variable lambdas in line 9 to 12 of Listing C.3. Listing C.3 also show the verbosity of JSCore, as it has double the lines of Listing C.2, and almost eight times the size. These examples also show how a simple function is represented in external Core as one lambda with two case-expressions.

Improvements to `core2js` would not be very beneficial until the problems further up the pipeline are solved. When other, larger issues are removed, it might be better to completely scrap `core2js`, and write a new parser that can parse external Core directly.

Hudak et al. [19] note that Haskell has gained "dozens of language extensions (notably in the type system)" [19, p. 29]. In the newer versions of the Haskell Prelude, these language extensions are used in great number. For example, the `GHC.Base`-module that is imported by almost all modules the Prelude use the following extensions: NoImplicitPrelude, BangPatterns, ExplicitForAll, MagicHash, UnboxedTuples, ExistentialQuantification, and RankNTypes. While most of these

[2]External Core parser package, *extcore*, on HackageDB:
`http://hackage.haskell.org/package/extcore`

are removed by GHC before Core, some leave their marks in the code produced by external Core [34].

Skrede, who designed and implemented PyHaskell's pipeline, concluded that: "the serialization of Haskell programs into the JSCore format have several problems. It is dependent on a buggy part of GHC, and on a poorly maintained package (extcore)." [43, p. 43].

4.2.1 Improvements

With small improvements of external Core, such as support for new integer literals and left and right coercion, a few parts of Haskell Prelude can be converted to external Core [34].

These improvements are not yet included in any GHC release, so to use them require building a custom version. As the *extcore* package is outdated, it is best to pull them onto the 7.4 branch of GHC, before building.

PyHaskell handles the Prelude by compiling parts of it to external Core beforehand, with a custom built GHC version. Other parts are implemented in PyHaskell with RPython code. If an unknown identifier is requested, PyHaskell will try to load the external Core file for its module. If it is not there, it will exit with an error [11].

As described in subsection 4.3.1, GHC's core developers believe external Core should be rewritten. I strongly agree with them as I have reached the same conclusion, which is explained in section 4.4,

4.3 Possible solutions

Core and external Core must be re-synced before PyHaskell can support a larger part of the Haskell language, and more importantly the Haskell standard library Haskell Prelude. There are two obvious solutions to this problem: a) rewrite external Core support in GHC; or, b) use the GHC API.

4.3.1 Rewrite external Core

Peyton Jones and Fischer [28] suggest how GHC's external Core support should be rewritten. The external Core data type should be dropped. Instead external Core should be based directly on Iface Core, as GHC already can convert Core into Iface Core. A new printer should be written that takes Iface Core and converts it into the current external Core syntax. This new printer could be based on an already existing Iface Core printer, BinIface, which support serialization and deserialization of Iface Core.

As GHC's external Core support has almost no tests, a first step of rewriting it must be to create tests, to ensure the rewrite generates the same grammar as the current implementation.

4.3.2 GHC API

External Core is not the only way to use GHC as a frontend. As described in subsection 2.2.3, the GHC API provide a Haskell interface to GHC's internal data structures, such as Core.

For PyHaskell to use GHC API instead of external Core, it would need to create a Haskell program that interface with Core and serialize it to an external representation. Just from this description, it should be obvious that this is the main goal of GHC's external Core support.

The main benefit from this approach would be that PyHaskell could use Core after the CorePrep transformation pass, which have more invariants than external Core, such as [37]:

- Function call arguments are atoms, either variable or literal.
- Case always means evaluation.
- Let always means allocation

We would also be able to remove kinds and types without an extra step, as our current approach has in the `core2js` program.

Lambdachine, an experimental tracing JIT backend for GHC under development by Schilling [38], use the GHC API. Lambdachine serialize Core to bytecode, which is further explained in section 11.3. Lambdachine struggles with some of the same issues as PyHaskell, even though it uses GHC API instead of external Core. Both can so far only support a very limited subset of the Haskell Prelude.

4.4 Discussion

While external Core support a large number of Haskell language features, it does not support the Haskell Prelude. Therefor PyHaskell cannot support GHC's test suite nor the nofib benchmark suite. It would require one of the solutions mentioned earlier, and even then it would probably be parts of the Haskell Prelude PyHaskell cannot support.

"Over the fifteen years of its life so far, GHC has grown a huge number of features (...) This makes GHC a dauntingly complex beast to understand and modify and, mainly for that reason, development of the core GHC functionality remains with Peyton Jones and Simon Marlow." [19, p. 29]

In my attempts at improving GHC's external Core, I must agree that understanding and modifying GHC is a huge undertaking, which is further complicated by the complexity of the Haskell language itself. I have been unable to complete either of the two possible solutions, because of my inexperience with Haskell and the complexity of GHC, but also because of trouble with building GHC. It has been a time-consuming undertaking that has not been fruitful, lot of time required to understand the issues and how to solve them.

It is my opinion that the best solution would be based on the GHC API. To use it to serialize Core after the CorePrep transformation pass, in a format based

on the current external Core grammar. This would allow a new parser to be useful for both solutions. If GHC core developers decide to rewrite external Core themselves, we could see which solution is/was better. Furthermore I believe the current pipeline should be scrapped, and a new parser should be written, as the current *jscparser*-module is designed for JSCore, which is too verbose. As an educated guess, either solution would require one to two work months for an experienced Haskell programmer to complete.

CHAPTER 5

PyHaskell

This chapter describes an RPython implementation of Haskell called PyHaskell, which more specifically is a tracing JIT backend for GHC (section 5.2). The Haskell-Python project started the work that led to the PyHaskell VM (section 5.1). GHC and our VM are connected with a pipeline (section 5.3) that has several problems and limitations (section 5.4). Some of these issues have been fixed during previous and current work. section 5.5 detail improvements that was required to support more advanced benchmarks, while section 5.6 describe our VM's current support of the Haskell language and the Haskell Prelude.

5.1 Haskell-Python project

Launchbury [21] designed an operational semantic for lazy languages with a heap as the only computational structure. The semantics are divided into two stages:

- A static transformation where creating and sharing of closures are explicit, all bound variables are distinct, and all applications are of an expression to a variable.
- A dynamic semantics with a heap binding variable names to expressions. Evaluation involves adding new bindings to the heap, or updating based on a reduction rule.

Bolz, Fischer, and Christiansen [11] implemented an RPython VM[1] for an extended lambda calculus, based on Launchbury's semantic for lazy languages.

[1] Haskell-Python's implementation of Launchbury's semantics can be found in the *clean2* branch of Haskell-Python's repository [11].

Lambda calculus is a system for expressing computations with variable bindings and substitution.

How GHC's external Core are translated to PyHaskell's implementation the natural semantics have been described by Skrede [43].

5.2 The PyHaskell virtual machine

As a GHC backend, PyHaskell only need to implement a VM for the Core language (see section 2.3), as opposed to the very large Haskell language. The goal of supporting the Haskell test suite and the *nofib* citepartain benchmark suite demand support for a large part of the Haskell Prelude. Implementing the Prelude in RPython would be a colossal undertaking, instead PyHaskell hope to reuse GHC's implementation of the Prelude, which is written mostly in Haskell.

PyHaskell consists of the operational semantics described in section 5.1; the pipeline described in section 5.3; the JSCore parser in the *jscparser*-module; a bunch of primitive operations and data types; and a set of built-in libraries implementing a small subset of Haskell and Haskell Prelude. These primitive operations and data types cannot be implemented in Haskell, and are instead provided through the virtual module `GHC.Prim`.

In GHC, the primitive operations are implemented[2] after the Core stage, hence PyHaskell must provide an implementation of them written in RPython. The implementation is provided by the *prim* and *primtype*-modules [11].

Launchbury's semantics have been implemented in the *haskell*-module of PyHaskell, including Launchbury's *Constructors* and *Constants* extensions. The *haskell*-module contains the following constructs:

- Var, a variable
- Rule, patterns that match an expression.
- Function, a named collection of rules.
- PrimFuncton, a function that is written in RPython.
- Application, the evaluation of a function.
- Constructor, a symbol and zero or more variables.
- Substitution, a functions body where variables are replaced with values.
- Thunk, an unevaluated application.

The evaluation of these constructs consist of a "todo" stack of expression to evaluate, as can be seen in Listing 7.6. Apply involves replacing variables with values in functions and applications, while step progress the evaluation one step further by removing or replacing an element on the stack.

[2] More details about GHC's primitive operations can be found in GHC wiki: `http://hackage.haskell.org/trac/ghc/wiki/Commentary/PrimOps`

GHC's Cmm representation is used as input for the backends included in GHC, but PyHaskell use the Core intermediate representation. Therefore some of GHC's optimizations are unavailable to PyHaskell, such as TNTC. Other optimizations are incomplete, only half-done, e.g., unboxing. This can be seen in external Core output where unboxed types end with # (see Listing C.2). GHC unbox extensively for performance reasons [29]. Optimizations performed by GHC are further described in subsection 2.2.2.

One aspect of the Haskell language not mentioned yet, is that functions may be applied to fewer or more arguments than the function arity. This is known as partial application and over-application. It must be handled at runtime [38]. While PyHaskell tries to support partial application, it does not support over-application.

5.3 PyHaskell pipeline

Skrede [42] extended the Haskell-Python project by designing and implementing PyHaskell's pipeline, which use GHC as a frontend and the lambda calculus VM created by Bolz et al. as backend. The structure of this pipeline can be seen in Figure 5.1. The GHC frontend handle parsing and type-checking of Haskell source code before it desugars Haskell into the Core intermediate representation. GHC then serialize Core into external Core, which is written to a "hcr" file[3].

External Core is converted to JSON with the `core2js` program, which `core2js` use the *extcore* package[4] to parse external Core. Core, now in a JSON representation called JSCore, is written to a "hcj" file [42].

The JSCore file is then interpreted by the PyHaskell VM. First the file is parsed with the *jscparser*-module, which create lambda calculus operations based on Launchbury's semantic. These operations are implemented in the *haskell*-module, and the final step of the PyHaskell pipeline is evaluation of the operations created [11].

It is easier for compilers to do type-checking and error-checking after synthetic sugar is removed, but GHC is unusual as it perform type-checking both before and after it desugar Haskell into Core [19]. PyHaskell therefore knows its input is type-safe.

Examples of the different representations that Haskell source code goes through before being fed to our VM can be seen in Appendix C on page 69. First, as Haskell code in Listing C.1. Then as external Core in Listing C.2, which `core2js` convert to JSCore as seen in Listing C.3.

5.4 Pipeline issues and previous work

The PyHaskell pipeline implemented by Skrede was incomplete, with three important limitations:

[3] GHC frontend is described in subsection 2.2.1 on page 7, while external Core is described in chapter 4 on page 19.

[4] extcore package: `http://hackage.haskell.org/package/extcore`

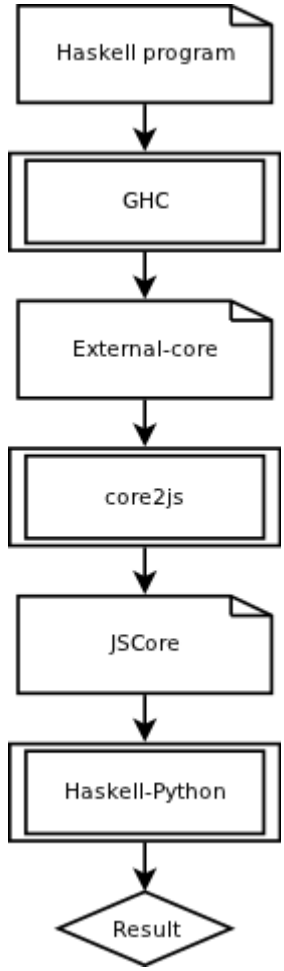

Figure 5.1: Haskell-Python's pipeline by Skrede [42]

1. Some issues related to GHC's external Core support and the *extcore* package, as explained in section 4.2. My attempt at fixing these issues are described in chapter 4.

2. The *jscparser*-module was written in Python, and not RPython, which meant it could not be translated with the RPython toolchain to C, and it could not use the RPython meta-tracing JIT.

3. The *jscparser*-module only supports a limited subset of the Haskell language.

I had to completely rewrite the *jscparser*-module in RPython to fix the second limitation. The code that walked the AST used different functions depending on the type of the node visited, and these functions failed to have compatible type signatures, as required by the RPython type system and its limitations.

I extended PyHaskell with support for `Constructors`; type aliases; a number of built-in Haskell functionality from the Haskell Prelude; and more. While this work improved PyHaskell's support of Haskell, the last limitation of the pipeline is still a work in progress, where progress is dictated by the needs of benchmarks. An overview over the current status can be seen in section 5.6. I also Z-decoded (see Appendix D) external Core identifiers in `core2js`; and removed kinds and some type information from the JSCore output of `core2js`.

5.5 Improvements

This section describe improvements undertaken to be able to run more advanced benchmarks with PyHaskell.

5.5.1 jscparser

My first bug fix was in the *jscparser*-module, where the code assumed that an atomic expression was one of: a variable; a data constructor; a literal; or a nested atomic expression. Obviously, it would be meaningless to nest an atomic expression if it could only be one of those, and the correction was to allow an atomic expression to be a nested expression. A small change that also allowed removal of a type check on all atomic expression nodes in the AST.

5.5.2 putStrLn

`putStrLn` is a simple function that takes a list of `Char`'s (a string), and the prints them (with a newline) to `stdout`. The type of `putStrLn` is defined in Haskell Prelude as:

```
String -> IO ()
```

While it was implemented with the correct behavior, in as far as it printed correctly to `stdout`, it returned the `Char` list instead of `IO ()`. It was done this way to

avoid having to implement parts such as IO, but it pushed the list further down the evaluation stack, which could be problematic.

Consider the simple example of printing the string "Hey" to `stdout`:

```
main = putStrLn "Hey"
```

In the external Core output produced by GHC (Listing 5.1[5]), one might see that the example simply calls `putStrLn` with the unpacked string "Hey". PyHaskell converts this to a set of operations that it evaluates, and in line six of Listing 5.2 the string is printed. The argument to `putStrLn` can be seen again in line seven, and it is given to `runMainIO` in line eight, where evaluation ends.

```
%module main:Main                                                    1
  main:Main.main :: (ghc-prim:GHC.Types.IO                           2
                     ghc-prim:GHC.Tuple.()) =                        3
    base:System.IO.putStrLn                                          4
    (ghc-prim:GHC.CString.unpackCString#                             5
     ("Hey"::ghc-prim:GHC.Prim.Addr#));                              6
  main::Main.main :: (ghc-prim:GHC.Types.IO                          7
                      ghc-prim:GHC.Tuple.()) =                       8
    base:GHC.TopHandler.runMainIO @ ghc-prim:GHC.Tuple.()            9
    main:Main.main;                                                  10
```

Listing 5.1: putStrLn example: GHC external Core output

```
runMainIO putStrLn unpackCString# Hey  =>                            1
putStrLn unpackCString# Hey  =>                                      2
unpackCString# Hey  =>                                               3
(Constr : H, (Constr : e, (Constr : y, (Constr [])))) =>             4
putStrLn (Constr : H, (Constr : e, (Constr : y, (Constr [])))) =>    5
Hey                                                                  6
(Constr : H, (Constr : e, (Constr : y, (Constr [])))) =>             7
runMainIO (Constr : H, (Constr : e, (Constr : y, (Constr [])))) =>   8
```

Listing 5.2: putStrLn example: PyHaskell evaluation log

I fixed this by having putStrLn create an `IO ()` constructor, and return it instead. This can be seen in Listing 5.3, where I show only the last two lines that now have the correct behavior.

```
(Constr IO (Constr ()))      =>                                      7
runMainIO (Constr IO (Constr ()))      =>                            8
```

Listing 5.3: putStrLn example: PyHaskell evaluation after

[5]Listing 5.1 has been Z-decoded according to Table D.1 on page 74.

5.5.3 runMainIO

`runMainIO` is a simple wrapper around `Main.main` that should catch otherwise uncaught exceptions, and should also flush standard output (stdout) and standard error (stderr) before exiting. Similarly to `putStrLn`, it would incorrectly return the argument it was given. I changed it to return the unit Constructor instead.

5.5.4 Show

The class `Show` and its instances implement the function `show`, which convert a value to a string. For values of type `Int show` will return a string with the value inside it. For example calling `show` with the value 10 returns the string "10":

```
show 10 = "10"
```

Show was only implemented for `Int` values, and in a very incorrect way. The show function simply returned the argument it was given, and "$fShowInt" converted an `Int` into a list of characters. $fShowInt should be a typeclass argument to the show method, telling it the type of the argument to show. I fixed this, as well as implement show for Integer and Char. Show for Lists has not been implemented, as it takes one more argument than regular show, and PyHaskell does not support over-application [11].

5.5.5 Built-in libraries

To be able to support more advanced benchmarks and tests, I've had to extend the support of the Haskell Prelude by adding functionality implemented in RPython. The list module have been extended with the functions `length`, `replicate`, and `index`. The num and prim modules were extended with integer division.

5.6 Summary of current status

This section describes the current status of PyHaskell. The Haskell language features are listed[6] in Table 5.1. Some features from the Haskell Prelude are listed at bottom of the table. The table also shows what did not work *before*[7] the work on this report was started.

External Core issues, as mentioned in chapter 4, are the main reason for some of these shortcomings. Still, a few elements of external Core are not yet implemented in the *jscparser*-module, i.e., cast, note, label, external ccall, and dynexternal ccall. Type coercion should not be necessary as we throw away type information, and C calls are a challenge we considered unimportant at this stage.

[6] The features are tested with Haskell source code from either PyHaskell's test suite (branch "even") [11], or the `feature`-folder of the *thesis-benchmarks* repository: `http://bitbucket.org/eventh/thesis-benchmarks`

[7] "Before" in Table 5.1 refers to PyHaskell revision `73ee6331dc5c` [11]

Table 5.1: Overview over Haskell features PyHaskell support

Feature tested	Before	Now	Note
Hello world!	Yes	Yes	
Case expression	Yes	Yes	
Let expression	Yes	Yes	
Data constructor	Yes	Yes	
Partial function application	Yes	Yes	
Recursive function	Yes	Yes	
List concatenation '++'	Yes	Yes	
Cons operator ':'	Yes	Yes	
Function composition operator '.'	Yes	Yes	
Function application operator '$'	Yes	Yes	
Indexing operator '!!'	No	Yes	
Pattern matching	No	Yes	
Guards	No	No	External Core error
Enumerations	No	No	Missing "GHC.Enum" support
Recursive let expression	No	No	Missing "%let" support
List comprehension	No	No	Missing "%let" support
Int division	No	Yes	
List take	No	Yes	
List replicate	No	Yes	
Show for Char	No	Yes	
Show for Integer	No	Yes	

CHAPTER 6

RPython hints and just-in-time optimization techniques

This chapter describes RPython hints that enable JIT optimization techniques. Hints are placed in the source code of VMs written in RPython, often in the form of decorators or function calls. While some of the hints are explained in research papers and blog posts, other hints are not described anywhere and I had to investigate the RPython source code to be able to explain how and why these RPython hints are used.

Chapter 7, and especially section 7.3, describes where these hints are used in PyHaskell. The consequence of these hints on the VM's performance can be seen in section 7.4.

6.1 Can enter jit and merge point

RPython's meta-tracing JIT compiler require two hints to work: `can_enter_jit` and `merge_point`. The first hint specify the start of a loop, and the second state where it is safe to return control from the JIT to the interpreter.

During translation `can_enter_jit` and `merge_point` are replaced with calls that invokes the JIT during runtime.

Tracing starts and ends at `can_enter_jit` hints. After a trace is detected to be hot, the list of operations in the trace is passed to the optimizer, before compilation to assembly. The `can_enter_jit` and `merge_point` hints are thus not like other RPython hints that turn on optimization techniques. Instead they enable tracing that makes it possible to use these JIT optimization techniques.

Guards often fail in between merge points, which according to Peterson, "is one of the most difficult parts of JIT implementation, since the interpreter state has to be reconstructed from the register and stack state at the point the guard

failed" [26, p. 286]. A *blackhole* interpreter then execute "jitcodes" (operations recorded at translation time, not runtime) until a merge point is reached.

To place these two hints in the source code of a VM, one must first create an `JitDriver`-instance[1], and give it two parameters, which list variable names split into two groups: "green" variables that represent a position in the interpreted program, which for bytecode-based interpreters are typically the program counter; while the remaining variables are "red" [7, 8]. JitDriver accept more arguments when instantiated, e.g., virtualizables; a function for printing debug information; and more. The `can_enter_jit` and `merge_point` hints are methods on the JitDriver-object.

6.2 Promotion

A variable is constant if its value is statically known by the optimizer. Literal values such as 1 or 2 are obvious constants, but also a variable can be constant, depending on its context. For example, in a trace, if a variable is protected by a guard it must be constant (from SSA), or the guard would fail.

Promotion is a technique to turn an arbitrary variable into a constant, which can be very powerful when used by JIT compilers. Promoting variables to constants gives the opportunity for optimization by constant folding. There are often places where a lot of computations depend on the value of a variable.

Bolz et al. [9] describe the promotion technique, and how it can be used in a VM written in RPython. The RPython source code hint `promote` is a function that accepts one argument, the variable that should be promoted.

Bolz et al. describe an example usage of the promotion technique, which can be seen in Listing 6.1, Listing 6.2, and Listing 6.3. The function `f1`, written in RPython, are shown in Listing 6.1. The example assumes that the variable `x` rarely change, and is therefore turned into a constant with the `promote` call. With `x` as the value 4, we can see the unoptimized trace of `f1` in Listing 6.2, while the same trace after constant-folding optimization can be seen in Listing 6.3.

If the value of `x`, in the above example, is not 4, the guard fails and execution return to the interpreter. A promotion hint will never produce wrong results, but if a variable changes often it will require repeated tracing and produce too much machine code. Therefore, promotion can slow down the VM when it used incorrectly.

```
def f1(x, y):                                                    1
    promote(x)                                                   2
    z = x * 2 + 1                                                3
    return z + y                                                 4
```

Listing 6.1: Promotion example: RPython code — Bolz et al. [9, p. 4]

[1] The *JitDriver* class comes from the `pypy.rlib.jit` module of the PyPy source tree.

```
guard(x1 == 4)
v1 = x1 * 2
z1 = v1 + 1
v2 = z1 + y1
return(v2)
```

Listing 6.2: Promotion example: unoptimized trace — Bolz et al. [9, p. 4]

```
guard(x1 == 4)
v2 = 9 + y1
return(v2)
```

Listing 6.3: Promotion example: optimized trace — Bolz et al. [9, p. 4]

6.3 Trace-elidable

A function is trace-elidable if "during execution of the program, successive calls to the function with identical arguments always returns the same result. In addition the function needs to have no side effects or idempotent side effects" [9, p. 4]. The RPython decorator `@elidable` is used to mark functions as trace-elidable. In a trace a call to a function decorated with the hint can be replaced with the result of the call.

An example of trace-elidable can be seen in Listing 6.4, which uses both `promote` and `elidable` hints. The unoptimized trace of the call `a.f(val)` (where `a` is an instance of class `A`) can be seen in Listing 6.5. The two hints allows the JIT to optimize the trace, as seen in Listing 6.6. The call to `a.c()` is replaced with the result of the call as the method is decorated as elidable. Furthermore the promote introduces a guard[2] on the instance, and constant-fold the value of `a.x` to 4. In this example, `promote` without the `elidable` hint would not remove the calculations in `a.c()` as `a.x` is not proven constant.

The `elidable` hint, as opposed to `promote`, will produce wrong code if it is used incorrectly. If any of the requirements for trace-elidable is broken, very subtle bugs will be introduced that are hard to debug.

6.4 Loop unrolling

If the RPython JIT tracer encounter a function call it must decide if it should inline it by tracing into the function, or just record the function call as an operation in the trace. Bolz explains that the default behavior is to "trace as much as possible (everything by default) except the functions which loops where tracing would produce code that is less general than it could be" [4].

Unrolling loops can be harmful, if the loop is large or consist of many iterations, the unrolled loop would produce an excessively large trace. Also, many functions should not be unrolled as it would specialize them in the trace for a specific behav-

[2] The value `0xb73984a8` is the address of the instance of `a` protected by the guard.

```
class A(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def f(self, val):
        promote(self)
        self.y = self.c() + val

    @elidable
    def c(self):
        return self.x * 2 + 1
```

Listing 6.4: Elidable example: RPython code — Bolz et al. [9, p. 4]

```
x1 = a1.x
v1 = x1 * 2
v2 = v1 + 1
v3 = v2 + val1
a1.y = v3
```

Listing 6.5: Elidable example: unoptimized trace — Bolz et al. [9, p. 4]

```
guard(a1 == 0xb73984a8)
v2 = 9 + val1
a1.y = v2
```

Listing 6.6: Elidable example: optimized trace — Bolz et al. [9, p. 5]

ior. For example, a function that loops over elements in a list, would be specialized for a list of a specific size. The optimized and compiled trace would then only be useful when it worked on a list of that size.

RPython provide two hints for controlling when to unroll and not: `unroll_safe` and `dont_look_inside`. The first is for false negatives, which specifies that the loop (in a trace) should always be unrolled; while the latter is for false positives, it says that the JIT should produce a call to the function and not trace into it. These two hints are used as decorators on RPython functions — `unroll_safe` should only be used when a loop is expected to run the same number of iterations [4].

In `pypy.rlib.unroll`, RPython provide a special hint that will unroll a loop (iterable) at translation time: `unrolling_iterable` [46].

An example of how the `@unroll_safe` decorator is used in PyHaskell can be seen in section 7.3.

6.5 Immutable fields

The `immutable_fields` hint is used to mark class fields that is immutable, so that reading them can be inlined. The hint is used by setting a special class attribute `_immutable_fields_` to a list with names of the fields that are immutable. Each name can be followed by one of three optional tags:

1. `"?"` mark the field quasi-immutable.
2. `"[*]"` mark a field as an immutable array.
3. `"?[*]"` for a quasi-immutable field pointing to an immutable array.

Quasi-immutable is used on fields that can change, but very rarely. If the field change, code that depends on the previous value is invalidated.

Listing 6.7 shows how `immutable_fields` hint can be used to mark three fields on a class as immutable. Using `immutable_fields` is equal to creating getters to access the fields and decorating them with `@elidable`.

```
class Test(object):
    _immutable_fields_ = ["quasi?", "one", "two[*]"]

    def __init__(self, quasi, one, two):
        self.quasi = quasi  # A quasi-immutable field
        self.one = one      # An immutable field
        self.two = two      # An immutable (virtualizable) list
```

Listing 6.7: RPython hint example: `immutable_fields`

6.6 Other hints and commands

In addition to the hints already mentioned, there are a few more hints that are not used as often, or used in a slightly different way[3]:

promote_string As `promote`, except it promotes a string by its value into a constant.

elidable_promote Same as `elidable`, but also promotes all function arguments to constants.

loop_invariant Decorate a function with no argument that return an object that is always the same in a loop.

look_inside_iff Only trace inside if the predicate provided is satisfied.

The `promote` and `promote_string` hints are shortcuts to the `hint`-command, which can also be used to mark variables as: `access_directly`, gives direct access to a virtualizable without treating it as one; or `fresh_virtualizable`, a virtualizable that was just allocated [46].

Runtime information from RPython's JIT can be accessed with three commands: `isconstant`, `isvirtual`, and `we_are_jitted`.

[3]The other RPython hints can all be found in *pypy.rlib.jit*-module from the PyPy repository [46].

CHAPTER 7

PyHaskell optimizations

This chapter describes optimizations of PyHaskell (section 7.2), with the help of JIT traces produced by the RPython toolchain (section 7.1).

7.1 RPython trace logs

The RPython translation toolchain can produce logs of low-level operations in traces. These trace logs allow us to find operations that can be removed to optimize PyHaskell. By rewriting PyHaskell or adding RPython hints time-consuming operations can be removed or replaced. The steps to create a log of operations in PyHaskell JIT traces for a Haskell source file are:

1. Translate PyHaskell to C with JIT optimization.
2. Set environment variable `PYPYLOG=jit-log-opt:jit.log`.
3. Run executable produced in step 1 with a Haskell source file.
4. `jit.log` now contains all JIT trace operations.

PyHaskell JIT trace logs are listed in Appendix B on page 63. These traces have been simplified slightly to make them easier to read (and to fit on one page) according to three simple rules:

- At start of lines, plus sign followed by a number are removed.
- The path of PyHaskell objects are removed.
- Instance memory addresses are removed.

```
[2e81e7f6ea1a] {jit-log-opt-loop
# Loop 0 (<Function> ds1dr4 dsdr3 ds1dr4) : loop with 115 ops
[p0, p1]
label(p0, p1, descr=TargetToken(1080639504))
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
guard_nonnull_class(p1, 137970976, descr=<Guard2>) [p1, p0]
p3 = getfield_gc_pure(p1, descr=<FieldP Substitution.inst_rhs 8>)
guard_value(p3, ConstPtr(ptr4), descr=<Guard3>) [p1, p0, p3]
p7 = getarrayitem_gc(p5, 0, descr=<ArrayP 4>)
guard_class(p7, 137971028, descr=<Guard4>) [p0, p5, p7]
p9 = getfield_gc(p7, descr=<FieldP Thunk.inst_application 8>)
guard_nonnull_class(p14, 137972304, descr=<Guard7>) [p0, p5, p12, p14, p7]
debug_merge_point(0, 0, 'None')
p30 = getfield_gc(ConstPtr(ptr29), descr=<FieldP CoreMod.inst_qvars 24>)
i34 = call(ConstClass(ll_dict_lookup_trampoline__v88___simple_call__function_ll),
                      p30, ConstPtr(ptr32), 360200661, descr=<Calli 4 rri EF=4>)
guard_no_exception(descr=<Guard14>) [p27, p20, p18, i34, p30, None, None, None, p0, p12, p7]
i40 = instance_ptr_eq(p18, p39)
i43 = int_sub(i41, i42)
i45 = int_eq(0, i43)
guard_false(i45, descr=<Guard17>) [p0, i43, None, None, None, None, p12, p7]
p47 = new_with_vtable(137970924)
setfield_gc(p47, i43, descr=<FieldS Int.inst_value 8>)
setfield_gc(p7, p47, descr=<FieldP Thunk.inst_application 8>)
i52 = int_is_true(i51)
jump(p61, p73, p30, p38, descr=TargetToken(1080639552))
--end of the loop--
[2e81e81d52ac] jit-log-opt-loop}
[309d5ce18a3] {jit-log-opt-bridge
# bridge out of Guard 12 with 79 ops
[p0, p1, p2, p3, p4, p5]
guard_value(p0, ConstPtr(ptr6), descr=<Guard116>) [p0, p1, p5, p2, p4, p3]
p57 = new_array(2, descr=<ArrayP 4>)
jump(p9, p5, i6, p3, p16, descr=TargetToken(-1223618496))
[309d8f63f93] jit-log-opt-bridge}
```

Listing 7.1: Simplified PyHaskell JIT trace log example

As an example, the simplified trace of Listing B.1 can be seen in Listing 7.1.

Listing 7.1 contain a selection of operations frequently encountered in PyHaskell trace logs. The first and last lines, as well as line 28 and 29, are artifacts of the trace log, signifying the start and stop of tracing and bridges. Start of loops and bridges are shown with lines starting with "# Loop" and "# bridge" respectively, as can be seen in line two and 30. The following line is a list of the arguments given to the trace. Line 15 is a dictionary lookup, which can frequently be removed with hints. Line 22 is a new instance, and as can be seen on line 23 it is an `Int` instance.

Operations that start with "guard" are different types of guards that protect points of divergence in traces, as explained in section 3.5. `debug_merge_point` operations are produced by the `merge_point` hint, which mark where it is safe to return interpreting back to the VM, as explained in section 6.1.

The remaining operations in are: getting and setting of fields and arrays; integer and pointer operations; and labels and jumps to those labels.

7.2 PyHaskell improvements from trace logs

This section describe improvements to PyHaskell, with the help of profiling and JIT trace-logs. The benchmarks used are described in section 8.1 on page 47. How these benchmarks were executed is explained in section 8.2.

7.2.1 Trace-elidable

I started with the simple addition benchmark in Listing A.1, which produce a JIT trace-log with few operations (Listing B.2 and Listing B.3). The trace contains two dictionary lookups, in line 36 and 74, that are obvious targets of improvement. These lookups stem from the hack used to implement numeric addition type-class, as can be seen in line 11 of Listing 7.3. Dictionary lookups can be removed with the trace-elidable technique described in section 6.3. I added an `@elidable`-decorated function, `get_var`, that did the `qvars` lookup, as can be seen in Listing 7.2. Line 11 of Listing 7.3 was then changed to call the `get_var` function.

```
from pypy.rlib.jit import elidable

@elidable
def get_var(module, name):
    return module.qvars[name]
```

Listing 7.2: Trace-elidable function for looking up in qvars dictionary [11]

Listing 7.4 is the part of the addition trace that perform the first dictionary lookup (before the trace-elidable changes). Those 10 operations are reduced to just two operations by the `@elidable` hint, as shown in Listing 7.5. Listing 7.5 is a small portion of the full trace log after these changes (Listing B.4 and Listing B.5). In total these changes reduce the number of operations in the JIT trace from 115 to 99, and reduced the number of guards from 21 to 15.

```
from pyhaskell.interpreter import prim
from pyhaskell.interpreter.haskell import make_application, Value
from pyhaskell.interpreter.module import CoreMod, expose_var
from pyhaskell.interpreter.primtype import Int

mod = CoreMod("base:GHC.Num")
mod.qvars["$fNumInt"] = Int(1)

@expose_var(mod, "+", [Int, Value, Value])
def add(ty, a, b):
    if ty == mod.qvars["$fNumInt"]:
        return make_application(prim.add, [a, b])
    else:
        raise NotImplementedError
```

Listing 7.3: PyHaskell numeric addition before `elidable` get_var [11]

```
p30 = getfield_gc(ConstPtr(ptr29), descr=<FieldP CoreMod.inst_qvars 24>)
i34 = call(ConstClass(ll_dict_lookup_trampoline__v32___simple_call__function_ll),
           p30, ConstPtr(ptr32), 360200661, descr=<Calli 4 rri EF=4>)
guard_no_exception(, descr=<Guard14>) [p27, p20, p18, i34, p30, None, None, None, p0, p12, p7]
i36 = int_and(i34, -2147483648)
i37 = int_is_true(i36)
guard_false(i37, descr=<Guard15>) [p27, p20, p18, i34, p30, None, None, None, p0, p12, p7]
p38 = getfield_gc(p30, descr=<FieldP dicttable.entries 12>)
p39 = getinteriorfield_gc(p38, i34, descr=<InteriorFieldDescr <FieldP dictentry.value 4>>)
i40 = instance_ptr_eq(p18, p39)
guard_true(i40, descr=<Guard16>) [p27, p20, None, None, None, p0, p12, p7]
```

Listing 7.4: Addition benchmark: trace excerpt, before elidable

```
i30 = instance_ptr_eq(p18, ConstPtr(ptr29))
guard_true(i30, descr=<Guard14>) [p27, p20, None, None, None, p0, p12, p7]
```

Listing 7.5: Addition benchmark: trace excerpt, after elidable

The result of these changes[1] on benchmarks from chapter 8, can be seen in Table 7.1. PyHaskell's performance compared to GHC can be evaluated with the results in Table 8.2.

Table 7.1: JIT results, before and after trace-elidable

	Before elidable			After elidable		
Benchmark	Runtime	Ops	Guards	Runtime	Ops	Guards
Addition	1.423 ± 0.026	115	21	1.390 ± 0.036	99	16
Fibonacci	2.757 ± 0.072	1327	417	2.102 ± 0.050	1119	351
Length	2.373 ± 0.042	159	30	2.317 ± 0.046	143	25

While the trace-elidable changes only gave a small improvement to the addition benchmark, it had a much larger effect on the naive Fibonacci benchmark. The operations count was reduced from 1327 to 1119, and the runtime was improved by 24%. PyHaskell is still far from GHC with all optimizations, but on Fibonacci it is only 1.4 times slower than GHC -O0.

7.2.2 Unboxed constructors

Some data constructors represent unboxed versions of primitive Haskell types, as mentioned in section 5.2. PyHaskell represent them with their boxed primitive version, but these may either be a value, or a function to change a value into the new constructor. As such, they were implemented as a function that takes a variable and return the same variable, which clearly could be a *noop* instead.

I added functionality that will use the literal value directly, if the constructor is an unboxed primitive type with a literal value. For example, `I# 10` is converted to the integer `10` directly, while `I# varA` is kept as a function that return the value of `varA`. This happens during parsing, and does not affect evaluation except removing a level of indirection.

The results of these changes[2] on benchmarks can be seen in Table 7.2. On Fibonacci PyHaskell is now 22% slower than GHC-O0, and around three times slower on the addition benchmark.

Table 7.2: JIT results, before and after unboxed constructors

	Before unboxed constructors			After unboxed constructors		
Benchmark	Runtime	Ops	Guards	Runtime	Ops	Guards
Addition	1.390 ± 0.036	99	16	1.057 ± 0.024	80	14
Fibonacci	2.102 ± 0.050	1119	351	1.835 ± 0.039	1010	334
Length	2.317 ± 0.046	143	25	2.084 ± 0.034	124	23

[1] Before trace-elidable refers to PyHaskell revision `0152989a42b1` [11], while after trace-elidable is PyHaskell revision `3841970a6bdd` [11].

[2] Before unboxed constructors refer to PyHaskell revision `3841970a6bb` [11], while after unboxed constructors is PyHaskell revision [11].

```
from pypy.rlib import jit

def get_printable_location(function):
    if function is None:
        return "None"
    return function.tostr()

jitdriver = jit.JitDriver(
        greens=["function"],
        reds=["todo", "expr"],
        get_printable_location=get_printable_location,)

def main_loop(expr):
    function = todo = None
    while True:
        jitdriver.jit_merge_point(
            function=function, todo=todo, expr=expr)
        if isinstance(expr, Substitution):
            expr = expr.apply()
        if isinstance(expr, Value) and todo is None:
            break

        expr, todo = expr.step(todo)
        function = None
        if isinstance(expr, Substitution):
            function = expr.rhs
            if expr.recursive:
                jitdriver.can_enter_jit(
                    function=function, todo=todo, expr=expr)
    return expr
```

Listing 7.6: PyHaskell haskell.py excerpt, `can_enter_jit` and `merge_point` [11]

7.3 PyHaskell's usage of RPython hints

Listing 7.6 is an excerpt from PyHaskell's *haskell*-module, that show PyHaskell's main loop. `can_enter_jit` and `merge_point` are only used inside the main loop, and a JitDriver-instance must be created before they are used. The `function` variable is marked as green, while `expr` and `todo` are red [11]. The role of these variables are explained in section 5.2.

PyHaskell treat only recursive functions as a possible trace heads, and every iteration of the main loop is considered a merge point (point where the loop may end). Lambdachine, on the other hand, treat every function as a possible entry point [38].

Listing 7.2 is an excerpt from PyHaskell that shows where I used the `elidable` hint. It is used to remove dictionary lookups, which is further explained in section 7.2.

`promote`, `unroll_safe`, and `immutable_fields` are used in many different places in PyHaskell, the excerpt in Listing 7.7 show some places where these hints are used. As most of the constructs in the *haskell*-module is immutable, the `immutable_fields` hint is used extensively. The loop in line 25 inside the

```python
from pypy.rlib.jit import unroll_safe, promote

class Substitution(HaskellObject):
    _immutable_fields_ = ["rhs", "subst", "recursive"]

    def __init__(self, rhs, subst, recursive):
        self.rhs = rhs
        self.subst = subst
        self.recursive = recursive

    def apply(self):
        promote(self.rhs)
        return self.rhs.substitute(self.subst)

class Constructor(Value):
    _immutable_fields_ = ["symbol"]

    def __init__(self, symbol):
        assert isinstance(symbol, Symbol)
        self.symbol = symbol

    @unroll_safe
    def substitute(self, subst):
        args = [None] * self.numargs()
        for i in range(self.numargs()):
            args[i] = self.getarg(i).substitute(subst)
        return make_constructor(self.symbol, args)
```

Listing 7.7: PyHaskell haskell.py excerpt showing three RPython hints [11]

substitute method will iterate once for each argument that a `Constructor` have, and since each Constructor has a constant number of arguments, the loop can be unrolled safely [11].

7.4 Performance improvements from hints

While the RPython hints often work together to improve performance, it is interesting to see the effect each hint has on PyHaskell. The effect of the trace-elidable hint can be seen in Table 7.1. The JitDriver hints, `can_enter_jit` and `merge_point`, must be present if one wish to include the RPython JIT.

Table 7.3 show the benchmark results from running on PyHaskell[3] with specific hints disabled, as well as PyHaskell with all hints enabled (PyHaskell JIT). Table 7.4 list operations and guards in JIT traces.

Table 7.3: PyHaskell performance with RPython hints disabled

Benchmark	PyHasjell JIT	No promote	No unroll	No immutable fields
Addition	1.037 ± 0.038	1.076 ± 0.017	3.113 ± 0.097	1.509 ± 0.082
Fibonacci	1.822 ± 0.026	7.094 ± 0.151	46.57 ± 1.956	13.00 ± 0.691
Length	2.0314 ± 0.033	2.293 ± 0.162	4.726 ± 0.116	2.277 ± 0.057

Table 7.4: PyHaskell jit trace operations with hints disabled

Benchmark	PyHasjell JIT	No promote	No unroll	No immutable fields
Addition ops	80	168	254	423
Addition guards	14	41	98	128
Fibonacci ops	1010	2610	2581	4681
Fibonacci guards	334	646	923	1466
Length ops	124	256	348	603
Length guards	23	64	140	179

[3]The different PyHaskell versions in Table 7.3 have all been made from PyHaskell revision `60e16de1181f` [11]. Benchmarks were executed as described in section 8.2.

CHAPTER 8

Benchmarks

This chapter describes benchmarks[1] used (section 8.1) used in this report, how they were executed (section 8.2), and the results they produced (section 8.3). The source code of these benchmarks can be found in Appendix A on page 61.

8.1 Design

To give the JIT room to warm up, these benchmarks were carefully crafted so that they would take more than one second to complete. Another requirement were that the benchmarks contain a recursive function, as JIT tracing is only turned on with recursive functions.

The issues and limitations of GHC's external Core prevent us from using the *nofib* [24] benchmark suite. Instead I have created a collection of micro-benchmarks, Appendix A show their source code.

Addition The *Addition* benchmark (source code in Listing A.1) was designed to produce a JIT trace-log with relative few operations. Fewer operations make the log easier to read, and to understand why operations are produced.

Fibonacci The naive implementation of the Fibonacci sequence (Listing A.2) is regarded as Haskell's "Hello, world!". It is co-recursive with simple integer addition and subtraction. It calculates the value of the 33rd element of the Fibonacci sequence, which makes it run sufficiently long to allow the JIT to warm up and take effect.

[1] Benchmarks, and software to run them, can be found in the *thesis-benchmarks* repository: `http://bitbucket.org/eventh/thesis-benchmarks`

Length The *length* benchmark (Listing A.3) creates a list of integers, and then recursively calculates the length of the list. PyHaskell lists are `Cons` constructors holding two items, the head and tail of the list. Length must recursively traverse the full list to create it, then once more to find its length.

Math Trace-based JIT compilers often perform better than ahead-of-time compilers when it comes to repeated calculations in a loop. The *Math* benchmark (Listing A.4) aim to do just that. It has three functions that are very suitable to inlining, and integer arithmetic that can be constant-folded away.

8.2 Execution

Each benchmark was run 50 times, and we report the mean average runtime with a 95% confidence interval, as recorded by the `time` Unix command. The reported time is in seconds. We also used the `chrt` command to give the process highest priority. For example, the naive Fibonacci benchmark was executed in the following way[2]:

```
chrt -f 99 /usr/bin/time -f \
    "\ntime: %e\ncontext switches: %c\nwaits: %w" \
    ./haskell-c fibonacci.hcj
```

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 12.10 32 bit, with Linux kernel 3.5.0-21-generic.

Our main interest is in the performance of PyHaskell translated to C with the RPython meta-tracing JIT enabled, compared to the Haskell reference implementation GHC with all optimizations enabled. It is also interesting to compare without the JIT, and GHC with optimization passes disabled[3]. The -O0 flag does not disable all optimizations, for example unboxing and TNTC are still active. The benchmarks were executed with the four[4] configurations[5] listed in Table 8.1.

8.3 Results

The results listed in Table 8.2 are the performance of the latest version of PyHaskell[6].

[2]The `time` command report context switches and wait times so we can re-run them if other processes influenced our results. The `time` command is provided by *GNU time 1.7*, while `chrt` is from *util-linux 2.20.1*.

[3]GHC optimizations are described in subsection 2.2.2 on page 9

[4]Lambdachine was not included as it require GHC 7.0.* and only works on x86-64 architecture. Htrace and GHC with DynamoRIO has not been released. PyHaskell untranslated is too slow to be of interested.

[5]PyHaskell was translated with PyPy 2.0 beta 1 (revision `07e08e9c885c`) [46], and compiled to machine code with GCC version 4.7.2. We used GHC version 7.4.2. The GHC LLVM backend used LLVM version 3.1.

[6]PyHaskell JIT in Table 8.2 were translated from PyHaskell revision `60e16de1181f` [11].

Table 8.1: Benchmark executable targets

Name	GHC flags	Note
PyHaskell JIT	-fext-core -fasm -O0	Translated to C with the JIT
GHC-O2	-fasm -O2	Native code generator, optimizations enabled
GHC-O0	-fasm -O0	Native code generator, optimizations disabled
GHC-LLVM	-fllvm -O0	LLVM backend, optimizations disabled

Table 8.2: Benchmark results

Benchmark	GHC-O0	GHC-O2	GHC-LLVM	PyHaskell JIT
Addition	0.361 ± 0.008	0.010 ± 0.000	0.375 ± 0.0283	1.039 ± 0.064
Fibonacci	1.504 ± 0.022	0.204 ± 0.004	1.618 ± 0.062	1.830 ± 0.106
Length	0.401 ± 0.009	0.042 ± 0.008	0.408 ± 0.032	2.034 ± 0.034
Math	11.73 ± 1.129	1.095 ± 0.077	12.36 ± 1.742	1.221 ± 0.167

8.4 Pipeline benchmarks

To be able to evaluate PyHaskell's performance, it is necessary to profile where the runtime is spent. When benchmarking, PyHaskell is provided with the JSCore representation directly, therefore it is a question of parsing versus evaluation. It might also be interesting to see how much time is spent in the different steps of the pipeline, and compare this to the `runghc` command. Runghc is a non-interactive interpreter that compiles and then execute Haskell code (it is interpreted, not just GHC's compile plus executing the machine code). Table 8.3 list the timings for the different benchmarks[7], while the runghc command had the following results: Addition 2.325s; Fibonacci 24.661s; Length 2.824s; Math 144.290s.

Table 8.3: PyHaskell pipeline benchmark results

Benchmark	External Core	`core2js`	Parsing	Evaluation	Total
Addition	1.220s	0.068s	0.034s	1.048s	2.336s
Fibonacci	1.202s	0.086s	0.013s	1.876s	3.123s
Length	1.210s	0.073s	0.010s	2.175s	3.332s
Math	1.294s	0.129s	0.029s	1.221s	2.576s

8.5 Built-in functionality or Prelude

The *Length* benchmark, as seen in Listing A.3 implement the length function in Haskell, instead of using the Haskell Prelude. Since external Core issues (as mentioned in chapter 4) prevents us from reusing GHC's Prelude implementation, I added the length function to PyHaskell directly in the VM written in RPython.

[7] PyHaskell that produce parse-timings was translated with PyHaskell revision `26df8324c54d` [11].

The RPython length function uses a while-loop, while the Haskell length uses recursion.

We can compare the performance of Prelude functions implemented in RPython against Prelude functions implemented in Haskell with the help of the *Length*, *Replicate* (Listing A.5), and *RPy-Length* (Listing A.6) benchmarks. Replicate is same as length, but the function that creates the list is written in RPython, while RPy-Length uses the RPython length implementation. We have included the result of using both length and replicate written in RPython (with the name "RPython"):

```
main = putStrLn (show (length (replicate 500000 9)))
```

Table 8.4: PyHaskell Prelude benchmark results

Benchmark	Runtime	Operations	Guards
Length	2.076 ± 0.250	124	23
Replicate	1.035 ± 0.157	45	9
RPy-Length	1.225 ± 0.196	79	14
RPython	0.126 ± 0.022	0	0

CHAPTER 9

Discussion

This chapter discusses the results achieved in chapter 8. As GHC-LLVM and GHC-O0 have very similar results we will focus on the NCG backend, and not the LLVM backend.

Addition benchmark PyHaskell is 2.87 times slower than GHC-O0, and over 100 times slower than GHC-O2.

Fibonacci benchmark PyHaskell is 21.7% slower than GHC-O0, and almost nine times slower than GHC-O2.

Length benchmark PyHaskell is five times slower than GHC-O0, and 48 times slower than GHC-O2.

Math benchmark *On this benchmark PyHaskell actually beat GHC-O0, and is very close to GHC-O2's performance.* GHC-O0 is 9.6 times slower than PyHaskell, which is 11.5% slower than GHC-O2

While I believe the JIT might eventually be able to beat GHC -O2, without using advanced optimizations in the frontend such as GHC, this is clearly not the case yet.

In section 7.4 I have presented the effect on PyHaskell from individual types of hints. Overall, it should be clear that the RPython translation toolchain with its meta-tracer is very potent. This is further shown with benchmarks results in Table 8.2.

Table 8.3 shows that a large majority of the runtime is spent in evaluation, and not parsing. Trace-based JIT compilers are not good at optimizing AST walking, therefore PyHaskell should be acceptable to further optimizations with the meta-tracing JIT.

CHAPTER 10

Conclusion

The RPython translation toolchain from the PyPy project can be used to write VMs in a high-level programming language, RPython. To reduce the overhead from the many abstractions provided by high-level languages, the toolchain provides a meta-tracing JIT compiler. The meta-tracer can be used by any VM written in RPython, with little or no work needed [10].

The VMs written in RPython cover many programming languages, most of them object-oriented, but none yet that are purely functional or lazy. Therefore this report asked the following question: *Is the RPython translation toolchain suitable for purely functional and lazy languages, e.g., the Haskell language?*

The Haskell language has achieved great speed with ahead-of-time compilation, and attempts at trace-based JIT optimizations of Haskell have not yet been able to beat static compilation. Hence this report also questioned whether *Haskell can benefit from trace-based JIT optimization techniques?*

To answer these two questions a prototype Haskell VM called PyHaskell was created. As PyHaskell is written in RPython, it includes the RPython meta-tracingJIT. The focus of this report has been to improve, optimize, and benchmark PyHaskell, with the hope of achieving performance comparable to GHC.

With the use of hints placed in the source code of VMs, the meta-tracer can enable optimization techniques that should improve runtime performance. Many of these hints and techniques were undocumented before this report, hence the report should have practical utility for anyone with interest in writing a VM in RPython.

Unfortunately, issues with GHC's external Core functionality limits PyHaskell's support of the Haskell language and Haskell Prelude. These limitations prevented evaluation of PyHaskell with the standard Haskell benchmark suite, *nofib* [24]. Instead I have created a small collection of micro-benchmarks, to compare PyHaskell's runtime performance with GHC.

PyHaskell is slower than GHC (with both -O0 and -O2) on thee of the benchmarks. On the *Math* benchmark, PyHaskell is actually faster than GHC -O0, and only 11.5% slower than GHC -O2. On the naive implementation of the Fibonacci sequence PyHaskell is 21.7% slower than GHC-O0, and almost nine times slower than GHC-O2.

As PyHaskell's performance is quite close to GHC's, the answer to the first question is *yes*: the RPython translation toolchain *is suitable* for purely functional, lazy languages.

While the meta-tracer greatly speeds up PyHaskell, it does not yet beat GHC. I cannot therefore answer the second question. More work is required to further improve PyHaskell, before a conclusion can be reached.

10.1 Future work

Possible extensions of the work described in this report should be further attempts at optimizing PyHaskell. Some of the other VMs written in RPython use optimization strategies in their object model, where they are able to use unboxed data structures. For PyHaskell, this could mean that instead of lists that are head-tail, it could use RPython lists that can be converted to arrays.

As mentioned in section 4.4, I believe the current PyHaskell pipeline should be scrapped. A better approach might be to use the GHC API with more invariants. The *extcore* package is no longer maintained, and it has some unwanted behavior (for example splitting up lambdas), a new pipeline should not depend on it. Furthermore, I would suggest writing a new parser, as the JSON representation of external Core is too verbose.

The constructs based on Launchbury's semantics for lazy languages perform well, but the mapping from Core to the constructs are far from optimal. For example, my changes to unboxed constructors described in section 7.2, reduced the number of constructs and gave decent improvements.

CHAPTER 11

Related work

This chapter draws connections to two related work on optimizing Haskell with trace-based compilation: One, Peixotto [25] with DynamoRIO and Htrace (section 11.2); And two, Schilling [38] with a method based on LuaJIT (section 11.3). While the LLVM backend for GHC contains a JIT, it is not trace-based (section 11.1). An overview over these optimizing attempts and how they relate to GHC are summarized in section 11.4 on page 59.

11.1 LLVM backend for GHC

Terei and Chakravarty [47] have created a new backend for GHC that uses LLVM, which is an optimizing compiler framework.

GHC started with translating STG to C, which made GHC portable across multiple architectures and operating systems. This C backend targets the GNU C compiler and its language extensions to get access to proper tail calls, first-class labels and more.

GHC's NCG was created to solve some of the downsides of the C backend: Relative long compilation time; Generated assembly code that are inefficient; And a complex Perl script nicknamed "the evil mangler" used to rewrite assembly code.

According to Terei and Chakravarty, the LLVM backend provides two main benefits compared to the NCG and C backends. First, offloading of work by outsourcing native code generating to the externally maintained LLVM project. Second, better performance by generating more efficient assembly and the LLVM JIT.

GHC's Cmm intermediate representation is the input to the LLVM backend, just as the other two backends. For that reason, the LLVM backend maintains

application binary interface (ABI) compatibility with the C and NCG backends. The backend performs three steps:

- Translate Cmm code to unoptimized LLVM code.
- Convert variables into SSA form, as demanded by LLVM. The backend allocate each mutable Cmm variable on the stack, and then uses `mem2reg` from LLVM to do SSA conversion.
- Map STG registers to hardware registers, by creating a new calling convention for LLVM where arguments of function calls are stored in registers. These arguments are only in the appropriate hardware registers on entry to any function, which is enough to satisfy GHC ABI.

Terei and Chakravarty have evaluated the LLVM backend against the C and NCG backends, in regards to their code size and the performance of the assembly code they generate. The LLVM backend is the smallest with 3.1 thousand lines of code (KLoC), the C backend is 5.3 KLoC, and the NCG backend is 20.5 KLoC.

The performance evaluation uses the *nofib* [24] benchmark suite and some additional benchmarks. The three backends had overall little difference on runtime for the nofib benchmarks. The NCG was 0.1% better than the LLVM, and the C backend was 2.7% slower than LLVM [47]. Terei and Chakravarty believes that the Cmm code used as input by all three backends are hard to optimize. The Cmm code is essentially memory bound as Haskell uses lazy evaluation. They tested this with other benchmarks with tight loops, where they saw considerable better runtimes for the LLVM backend.

11.2 DynamoRIO and Htrace

David M. Peixotto [25] has attempted to optimize low-level imperative code generated by GHC, with trace-based binary optimization techniques. He created a hand-coded case study that showed trace-based optimizations can be profitable for Haskell programs. Then he tested two different methods: First, with a dynamic binary trace-based optimizer; and second, with a static trace-based optimizer.

11.2.1 Nofib benchmark suite

Peixotto's initial investigations used the nofib [24] benchmark suite. He felt *nofib* was "difficult to collect accurate benchmark numbers" [25, p. 2] with, as many of them ran in less than one second. Peixotto therefore created the Fibon benchmark suite to more accurately evaluate the effects of compiler optimizations. Fibon consist of 32 benchmarks from four sources: Hackage[1], Shootout[2], Repa [20], and Data Parallel Haskell (DPH) [13].

[1] Hackage: `http://hackage.haskell.org`
[2] The Computer Language Benchmarks Game: `http://shootout.alioth.debian.org/`

11.2.2 DynamoRIO

DynamoRIO allows automatic tracing of programs at runtime, which Peixotto has used for exploring dynamic trace-based optimizations of Haskell. DynamoRIO build traces by monitoring application's stream of instructions, and therefore works on unmodified program binaries without need for the application's source code. Peixotto discovered that DynamoRIO added a 57% overhead to just find traces, so optimizations of traces must overcome this overhead for this method to be beneficial.

Peixotto believed the main problem was the heuristics used by DynamoRIO to build traces are not suitable for Haskell programs. Another issue was that the traces included arbitrary parts of GHC's runtime, such as the garbage collector.

11.2.3 Htrace

To avoid the runtime overhead from DynamoRIO, Peixotto created Htrace, which "finds traces in a separate profiling run and uses them to restructure the program offline" [25, p. 87]. Htrace consist of three distinct phases: One, finding traces; Two, restricting low-level code around traces; Three, optimizing the traces.

Htrace is built with GHC and LLVM — traces are optimized with compiler optimizations provided by LLVM. GHC required some small changes to enable dynamic linking of some C functions into LLVM. Htrace disables GHC's TNTC optimization[3] and uses a pure Haskell library for integer arithmetic, `integer-simple`. Two new passes were added to LLVM: inserting trace instrumentation and building traces. The LLVM bitcode interpreter, `lli`, was changed to add callbacks to the trace runtime. Htrace performs four main tasks:

- Create LLVM bitcode from program source.
- Create LLVM bitcode from Haskell libraries.
- Determine external libraries used.
- Create Makefile to do the build.

Peixotto compared Htrace against GHC with the LLVM backend. His Htrace results show an average speed up of 5% on the Fibon benchmarks, and a maximum speed up of 86% on a single benchmark. Only two benchmarks showed over 5% performance degradation.

11.3 Lambdachine

Thomas Schilling [38] has implemented a prototype VM with a tracing JIT compiler called Lambdachine. This VM use GHC as a frontend to compile Haskell code into Core. Core is then compiled into a custom bytecode format that Lambdachine

[3] TNTC is described further in subsection 2.2.2 on page 9

interprets. The VM and the bytecode format adopt ideas and techniques from LuaJIT 2[4].

Schilling has identified and formulated three challenges for optimizing lazy evaluation using trace-compilation:

Challenge 1 Can we use specialization to remove the overhead of evaluation and indirect function calls due to type class dictionaries. If the number of possible shapes per evaluation site is small, then the size of trace trees will remain small and thus remain efficient. It is, however, likely that there a few functions that are megamorphic, i.e., exhibit a large amount of different argument shapes [38, p. 5].

Challenge 2 Sharing information at runtime is important to enable deforestation with a dynamic compiler. Both static and dynamic approaches are possible and likely have different trade-offs in terms of accuracy, implementation cost, and runtime overhead. It is not clear which trade-offs will work best for dynamic optimization systems [38, p. 7].

Challenge 3 Evaluate different trace selection schemes by their coverage and code size. Functional programming benchmarks may exhibit different execution behavior from standard benchmarks for imperative/object-oriented languages [38, p. 8].

Lambdachine perform the following standard optimization techniques, where the forward optimizations are performed immediately, and the rest happen right before machine code generation:

- Common sub-expression elimination.
- Constant folding, algebraic optimizations and reassociation.
- Redundant load removal.
- Store-to-load forwarding.
- Redundant guard removal.
- Dead code elimination.
- Loop unrolling.
- Allocation removal.

According to Schilling, Lambdachine support: "basic Haskell programs that use only a small subset of built-in types, Char, Bool, and Int. All user defined types are supported, but the IO monad or arrays are not supported." [38, p. 13]. Loop unrolling are only applied to traces with constant stack usage, and side traces are not implemented.

Schilling has evaluated Lambdachine on two simple benchmarks with the JIT enabled or disabled. His results show that Lambdachine are able to remove significant amount of operations executed, when the JIT is enabled.

[4]The LuaJIT project: `http://luajit.org/`

Table 11.1: Overview over related work at optimizing Haskell

Name	Trace-based	JIT	GHC version	Creator
GHC LLVM backend	No	Yes	7.6+	Terei and Chakravarty [47]
GHC with DynamoRIO	Yes	Yes	7.4	Peixotto [25]
Htrace	Yes	No	7.4	Peixotto [25]
Lambdachine	Yes	Yes	7.0	Schilling [38]
PyHaskell	Yes	Yes	7.4	Bolz et al. [11]

11.4 Summary

The attempts to optimize GHC described in this chapter are summarized in Table 11.1. The LLVM backend for GHC has not achieved any clear performance advantage [47] over GHC's NCG. GHC with DynamoRIO was abandoned as tracing created too much overhead, and produced poor traces [25]. Htrace produced an average of 5% speed up over the LLVM backend [25]. Lambdachine is still in development, and has not yet published any results comparing it to any other GHC backends [38].

How these backends interface with GHC can be seen in Figure 11.1. Both PyHaskell and Lambdachine works from GHC's Core — PyHaskell though external Core and Lambdachine through the GHC API [38, 42]. The three backends included with GHC (LLVM, NCG, and C) are described in subsection 2.2.1. The GHC DynamoRIO backend works from the binary created by either the C or NCG backend, while Htrace works from the low-level code produced by the LLVM backend [22, 25, 47].

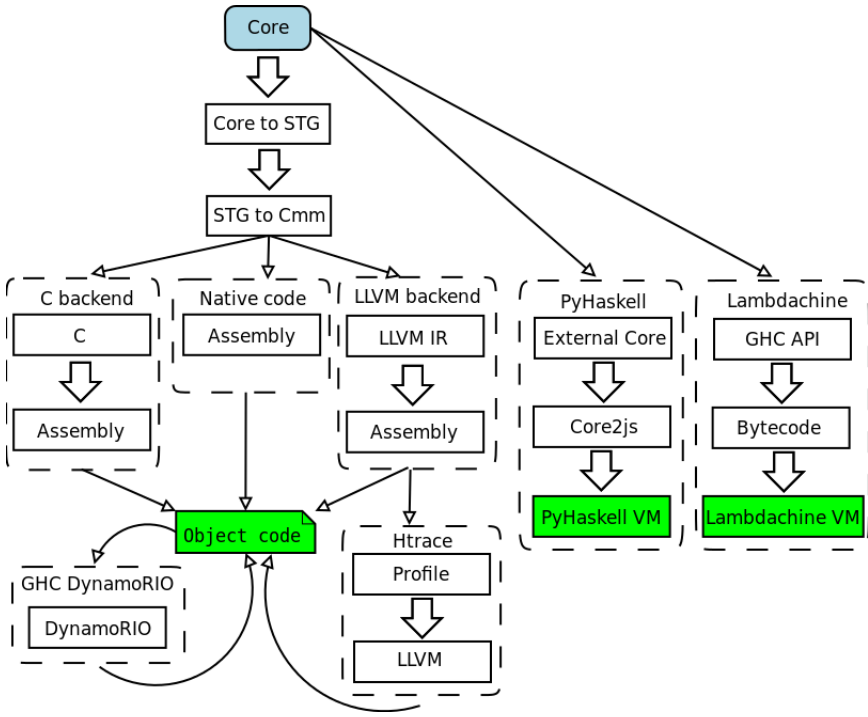

Figure 11.1: All GHC backends and their relationships

APPENDIX A

Benchmark source code

This chapter contain Haskell source code of benchmarks used in this report. The design of these benchmarks are explained in section 8.1 on page 47, while section 8.2 describe how they were run and how their performance was recorded.

```haskell
main = putStrLn (show (add 500000))

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

Listing A.1: Addition benchmark: Haskell source code

```haskell
main = putStrLn (show (fib 33))

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

Listing A.2: Fibonacci benchmark: Haskell source code

```haskell
main = putStrLn (show (length' (fromto' 500000)))

fromto' :: Int -> [Int]
fromto' 0 = []
fromto' i = i : fromto' (i - 1)

length' :: [Int] -> Int
length' [] = 0
length' (x:xs) = 1 + length' xs
```

Listing A.3: Length benchmark: Haskell source code

```haskell
main = putStrLn (show (test 20000000))

test :: Int -> Int
test 1 = 1
test x = test ((math1 x) + (math2 x))

math1 :: Int -> Int
math1 x = math3 ((10 * x) `div` (5 * 2))

math2 :: Int -> Int
math2 y = y + 1 - y

math3 :: Int -> Int
math3 z = z - 2
```

Listing A.4: Math benchmark: Haskell source code

```haskell
main = putStrLn (show (length' (replicate 500000 9)))

length' :: [Int] -> Int
length' [] = 0
length' (x:xs) = 1 + length' xs
```

Listing A.5: Replicate benchmark: Haskell source code

```haskell
main = putStrLn (show (length (fromto' 500000)))

fromto' :: Int -> [Int]
fromto' 0 = []
fromto' i = i : fromto' (i - 1)
```

Listing A.6: RPy-Length benchmark: Haskell source code

APPENDIX B

RPython trace logs

This section contains RPython JIT trace logs. Listing B.1 is an example of PyHaskell trace log that has not been simplified, and the remaining trace logs in this section have all been simplified as explained in section 7.1 on page 39. Listing B.1 is not a real trace log, but a selection of lines from the raw, original trace logs of the addition and naive Fibonacci benchmarks.

Listing B.2 and Listing B.3 show the trace log for the addition benchmark described in section 8.1 on page 47. These traces are created with PyHaskell revision `0152989a42b1` [11][1].

Listing B.4 and Listing B.5 are trace of the same benchmark, but after dictionary lookups have been removed with `@elidable` decorator hint, created with PyHaskell revision `3841970a6bdd` [11][1].

[1] PyHaskell created with RPython toolchain from PyPy 2.0 beta 1 (revision `07e08e9c885c`) [46]

```
[2e81e7f6ea1a] {jit-log-opt-loop
# Loop 0 (<Function object at 0x403122e8> ds1dr4 dsdr3 ds1dr4) : loop with 115 ops
[p0, p1]
+33: label(p0, p1, descr=TargetToken(1080639504))
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
+33: guard_nonnull_class(p1, 137970976, descr=<Guard2>) [p1, p0]
+54: p3 = getfield_gc_pure(p1,
        descr=<FieldP pyhaskell.interpreter.haskell.Substitution.inst_rhs 8>)
+57: guard_value(p3, ConstPtr(ptr4), descr=<Guard3>) [p1, p0, p3]
+72: p7 = getarrayitem_gc(p5, 0, descr=<ArrayP 4>)
+75: guard_class(p7, 137971028, descr=<Guard4>) [p0, p5, p7]
+88: p9 = getfield_gc(p7, descr=<FieldP pyhaskell.interpreter.haskell.Thunk.inst_application 8>)
+128: guard_nonnull_class(p14, 137972304, descr=<Guard7>) [p0, p5, p12, p14, p7]
debug_merge_point(0, 0, 'None')
+243: p30 = getfield_gc(ConstPtr(ptr29),
        descr=<FieldP pyhaskell.interpreter.module.CoreMod.inst_qvars 24>)
+249: i34 = call(ConstClass(ll_dict_lookup_trampoline__v88___simple_call__function_ll),
        p30, ConstPtr(ptr32), 360200661, descr=<Calli 4 rri EF=4>)
+281: guard_no_exception(descr=<Guard14>) [p27, p20, p18, i34, p30, None, None, None, p0, p12, p7]
+318: i40 = instance_ptr_eq(p18, p39)
+333: i43 = int_sub(i41, i42)
+335: i45 = int_eq(0, i43)
guard_false(i45, descr=<Guard17>) [p0, i43, None, None, None, None, p12, p7]
p47 = new_with_vtable(137970924)
+393: setfield_gc(p47, i43, descr=<FieldS pyhaskell.interpreter.primtype.Int.inst_value 8>)
setfield_gc(p7, p47, descr=<FieldP pyhaskell.interpreter.haskell.Thunk.inst_application 8>)
+485: i52 = int_is_true(i51)
+762: jump(p61, p73, p30, p38, descr=TargetToken(1080639552))
+775: --end of the loop--
[2e81e81d52ac] jit-log-opt-loop}
[309d5ce18a3] {jit-log-opt-bridge
# bridge out of Guard 12 with 79 ops
[p0, p1, p2, p3, p4, p5]
+6: guard_value(p0, ConstPtr(ptr6), descr=<Guard116>) [p0, p1, p5, p2, p4, p3]
p57 = new_array(2, descr=<ArrayP 4>)
+90: jump(p9, p5, i6, p3, p16, descr=TargetToken(-1223618496))
[309d8f63f93] jit-log-opt-bridge}
```

Listing B.1: Example of raw, not simplified PyHaskell JIT trace log

```
[df0be96bb30] {jit-log-opt-loop
# Loop 0 ((Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4) : loop with 115 ops
[p0, p1]
label(p0, p1, descr=TargetToken(-1223585776))
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
guard_nonnull_class(p1, 138375776, descr=<Guard2>) [p1, p0]
p3 = getfield_gc_pure(p1, descr=<FieldP Substitution.inst_rhs 8>)
guard_value(p3, ConstPtr(ptr4), descr=<Guard3>) [p1, p0, p3]
p5 = getfield_gc_pure(p1, descr=<FieldP Substitution.inst_subst 12>)
p7 = getarrayitem_gc(p5, 0, descr=<ArrayP 4>)
guard_class(p7, 138375840, descr=<Guard4>) [p0, p5, p7]
p9 = getfield_gc(p7, descr=<FieldP Thunk.inst_application 8>)
guard_nonnull_class(p9, 138377536, descr=<Guard5>) [p0, p5, p7, p9]
p12 = getarrayitem_gc(p5, 1, descr=<ArrayP 4>)
guard_class(p12, 138375840, descr=<Guard6>) [p0, p5, p12, p7]
p14 = getfield_gc(p12, descr=<FieldP Thunk.inst_application 8>)
guard_nonnull_class(p14, 138377536, descr=<Guard7>) [p0, p5, p12, p14, p7]
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
p16 = getfield_gc_pure(p9, descr=<FieldP Application.inst_function 8>)
guard_value(p16, ConstPtr(ptr17), descr=<Guard8>) [p16, p9, p0, p12, p7]
p18 = getfield_gc_pure(p9, descr=<FieldP Application3.inst_arg0 12>)
guard_class(p18, 138375712, descr=<Guard9>) [p18, p9, p0, p12, p7]
p20 = getfield_gc_pure(p9, descr=<FieldP Application3.inst_arg1 16>)
guard_class(p20, 138375712, descr=<Guard10>) [p20, p9, p18, p0, p12, p7]
p22 = getfield_gc_pure(p9, descr=<FieldP Application3.inst_arg2 20>)
guard_class(p22, 138376096, descr=<Guard11>) [p22, p9, p20, p18, p0, p12, p7]
debug_merge_point(0, 0, 'None')
p24 = getfield_gc_pure(p22, descr=<FieldP Application.inst_function 8>)
guard_value(p24, ConstPtr(ptr25), descr=<Guard12>) [p24, p22, p9, None, None, p0, p12, p7]
p27 = getfield_gc_pure(p22, descr=<FieldP Application1.inst_arg0 12>)
guard_class(p27, 138375712, descr=<Guard13>) [p22, p27, p9, None, None, p0, p12, p7]
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
p30 = getfield_gc(ConstPtr(ptr29), descr=<FieldP CoreMod.inst_qvars 24>)
i34 = call(ConstClass(ll_dict_lookup_trampoline__v32___simple_call__function_ll),
           p30, ConstPtr(ptr32), 360200661, descr=<Calli 4 rri EF=4>)
guard_no_exception(, descr=<Guard14>) [p27, p20, p18, i34, p30, None, None, None, p0, p12, p7]
i36 = int_and(i34, -2147483648)
i37 = int_is_true(i36)
guard_false(i37, descr=<Guard15>) [p27, p20, p18, i34, p30, None, None, None, p0, p12, p7]
p38 = getfield_gc(p30, descr=<FieldP dicttable.entries 12>)
p39 = getinteriorfield_gc(p38, i34, descr=<InteriorFieldDescr <FieldP dictentry.value 4>>)
i40 = instance_ptr_eq(p18, p39)
guard_true(i40, descr=<Guard16>) [p27, p20, None, None, None, p0, p12, p7]
debug_merge_point(0, 0, 'None')
i41 = getfield_gc_pure(p20, descr=<FieldS Int.inst_value 8>)
i42 = getfield_gc_pure(p27, descr=<FieldS Int.inst_value 8>)
i43 = int_sub(i41, i42)
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
i45 = int_eq(0, i43)
guard_false(i45, descr=<Guard17>) [p0, i43, None, None, None, None, p12, p7]
p47 = new_with_vtable(138375712)
setfield_gc(p47, i43, descr=<FieldS Int.inst_value 8>)
setfield_gc(p7, p47, descr=<FieldP Thunk.inst_application 8>)
p48 = getfield_gc(p12, descr=<FieldP Thunk.inst_application 8>)
guard_nonnull_class(p48, 138375712, descr=<Guard18>) [p48, p0, p12, p47, p7]
debug_merge_point(0, 0, '+ 1 (I# (_)) 1 (addr9Y (dsdr3)) - 1 dsdr3 (I# (_)) 1')
debug_merge_point(0, 0, 'None')
```

Listing B.2: Addition benchmark: trace before elidable, page 1

```
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '(Case (ds1dr4 dsdr3)) dsdr3 dsdr3')
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
label(p0, p48, p30, p38, descr=TargetToken(-1223585728))
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
i50 = call(ConstClass(ll_dict_lookup_trampoline__v32___simple_call__function_ll),
           p30, ConstPtr(ptr32), 360200661, descr=<Calli 4 rri EF=4>)
guard_no_exception(, descr=<Guard19>) [p48, i50, p30, p0]
i51 = int_and(i50, -2147483648)
i52 = int_is_true(i51)
guard_false(i52, descr=<Guard20>) [p48, i50, p30, p0]
p53 = getinteriorfield_gc(p38, i50, descr=<InteriorFieldDescr <FieldP dictentry.value 4>>)
i55 = instance_ptr_eq(ConstPtr(ptr54), p53)
guard_true(i55, descr=<Guard21>) [p48, p0]
debug_merge_point(0, 0, 'None')
i56 = getfield_gc_pure(p48, descr=<FieldS Int.inst_value 8>)
i58 = int_sub(i56, 1)
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
i59 = int_eq(0, i58)
guard_false(i59, descr=<Guard22>) [i58, p48, p0]
debug_merge_point(0, 0, '+ 1 (I# (_)) 1 (addr9Y (dsdr3)) - 1 dsdr3 (I# (_)) 1')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '(Case (ds1dr4 dsdr3)) dsdr3 dsdr3')
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
p61 = new_with_vtable(138375892)
p63 = new_with_vtable(138377536)
p65 = new_with_vtable(138376096)
setfield_gc(p63, ConstPtr(ptr66), descr=<FieldP Application.inst_function 8>)
p68 = new_with_vtable(138377536)
setfield_gc(p65, ConstPtr(ptr69), descr=<FieldP Application.inst_function 8>)
p71 = new_with_vtable(138376096)
setfield_gc(p68, ConstPtr(ptr17), descr=<FieldP Application.inst_function 8>)
setfield_gc(p71, ConstPtr(ptr72), descr=<FieldP Application1.inst_arg0 12>)
setfield_gc(p68, p71, descr=<FieldP Application3.inst_arg2 20>)
setfield_gc(p68, p48, descr=<FieldP Application3.inst_arg1 16>)
setfield_gc(p68, ConstPtr(ptr54), descr=<FieldP Application3.inst_arg0 12>)
p73 = new_with_vtable(138375712)
setfield_gc(p61, 2, descr=<FieldS CopyStackElement.inst_index 16>)
setfield_gc(p61, p0, descr=<FieldP StackElement.inst_next 8>)
setfield_gc(p71, ConstPtr(ptr25), descr=<FieldP Application.inst_function 8>)
setfield_gc(p65, p68, descr=<FieldP Application1.inst_arg0 12>)
setfield_gc(p63, p65, descr=<FieldP Application3.inst_arg2 20>)
setfield_gc(p63, ConstPtr(ptr75), descr=<FieldP Application3.inst_arg1 16>)
setfield_gc(p63, ConstPtr(ptr54), descr=<FieldP Application3.inst_arg0 12>)
setfield_gc(p61, p63, descr=<FieldP CopyStackElement.inst_application 12>)
setfield_gc(p73, i58, descr=<FieldS Int.inst_value 8>)
jump(p61, p73, p30, p38, descr=TargetToken(-1223585728))
--end of the loop--
[df0beada4b1] jit-log-opt-loop}
```

Listing B.3: Addition benchmark: trace before elidable, page 2

```
[e68b7659036] {jit-log-opt-loop
# Loop 0 ((Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4) : loop with 99 ops
[p0, p1]
label(p0, p1, descr=TargetToken(-1223131120))
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
guard_nonnull_class(p1, 138372064, descr=<Guard2>) [p1, p0]
p3 = getfield_gc_pure(p1, descr=<FieldP Substitution.inst_rhs 8>)
guard_value(p3, ConstPtr(ptr4), descr=<Guard3>) [p1, p0, p3]
p5 = getfield_gc_pure(p1, descr=<FieldP Substitution.inst_subst 12>)
p7 = getarrayitem_gc(p5, 0, descr=<ArrayP 4>)
guard_class(p7, 138372128, descr=<Guard4>) [p0, p5, p7]
p9 = getfield_gc(p7, descr=<FieldP Thunk.inst_application 8>)
guard_nonnull_class(p9, 138372576, descr=<Guard5>) [p0, p5, p7, p9]
p12 = getarrayitem_gc(p5, 1, descr=<ArrayP 4>)
guard_class(p12, 138372128, descr=<Guard6>) [p0, p5, p12, p7]
p14 = getfield_gc(p12, descr=<FieldP Thunk.inst_application 8>)
guard_nonnull_class(p14, 138372576, descr=<Guard7>) [p0, p5, p12, p14, p7]
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
p16 = getfield_gc_pure(p9, descr=<FieldP Application.inst_function 8>)
guard_value(p16, ConstPtr(ptr17), descr=<Guard8>) [p16, p9, p0, p12, p7]
p18 = getfield_gc_pure(p9, descr=<FieldP Application3.inst_arg0 12>)
guard_class(p18, 138371968, descr=<Guard9>) [p18, p9, p0, p12, p7]
p20 = getfield_gc_pure(p9, descr=<FieldP Application3.inst_arg1 16>)
guard_class(p20, 138371968, descr=<Guard10>) [p20, p9, p18, p0, p12, p7]
p22 = getfield_gc_pure(p9, descr=<FieldP Application3.inst_arg2 20>)
guard_class(p22, 138373088, descr=<Guard11>) [p22, p9, p20, p18, p0, p12, p7]
debug_merge_point(0, 0, 'None')
p24 = getfield_gc_pure(p22, descr=<FieldP Application.inst_function 8>)
guard_value(p24, ConstPtr(ptr25), descr=<Guard12>) [p24, p22, p9, None, None, p0, p12, p7]
p27 = getfield_gc_pure(p22, descr=<FieldP Application1.inst_arg0 12>)
guard_class(p27, 138371968, descr=<Guard13>) [p22, p27, p9, None, None, p0, p12, p7]
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
i30 = instance_ptr_eq(p18, ConstPtr(ptr29))
guard_true(i30, descr=<Guard14>) [p27, p20, None, None, None, p0, p12, p7]
debug_merge_point(0, 0, 'None')
i31 = getfield_gc_pure(p20, descr=<FieldS Int.inst_value 8>)
i32 = getfield_gc_pure(p27, descr=<FieldS Int.inst_value 8>)
i33 = int_sub(i31, i32)
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
i35 = int_eq(0, i33)
guard_false(i35, descr=<Guard15>) [p0, i33, None, None, None, None, p12, p7]
p37 = new_with_vtable(138371968)
setfield_gc(p37, i33, descr=<FieldS Int.inst_value 8>)
setfield_gc(p7, p37, descr=<FieldP Thunk.inst_application 8>)
p38 = getfield_gc(p12, descr=<FieldP Thunk.inst_application 8>)
guard_nonnull_class(p38, 138371968, descr=<Guard16>) [p38, p0, p12, p37, p7]
debug_merge_point(0, 0, '+ 1 (I# (_)) 1 (addr9Y (dsdr3)) - 1 dsdr3 (I# (_)) 1')
debug_merge_point(0, 0, 'None')
```

Listing B.4: Addition benchmark: trace after elidable, page 1

```
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '(Case (ds1dr4 dsdr3)) dsdr3 dsdr3')
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
label(p0, p38, descr=TargetToken(-1223131072))
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
i40 = getfield_gc_pure(p38, descr=<FieldS Int.inst_value 8>)
i42 = int_sub(i40, 1)
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
i43 = int_eq(0, i42)
guard_false(i43, descr=<Guard17>) [p38, p0, i42]
debug_merge_point(0, 0, '+ 1 (I# (_)) 1 (addr9Y (dsdr3)) - 1 dsdr3 (I# (_)) 1')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '_')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, 'None')
debug_merge_point(0, 0, '(Case (ds1dr4 dsdr3)) dsdr3 dsdr3')
debug_merge_point(0, 0, '(Case (0 dsdr3 ds1dr4) (_ dsdr3 ds1dr4)) ds1dr4 dsdr3 ds1dr4')
p45 = new_with_vtable(138372180)
p47 = new_with_vtable(138372576)
p49 = new_with_vtable(138373088)
setfield_gc(p47, ConstPtr(ptr50), descr=<FieldP Application.inst_function 8>)
p52 = new_with_vtable(138372576)
setfield_gc(p49, ConstPtr(ptr53), descr=<FieldP Application.inst_function 8>)
setfield_gc(p47, ConstPtr(ptr54), descr=<FieldP Application3.inst_arg1 16>)
setfield_gc(p47, ConstPtr(ptr29), descr=<FieldP Application3.inst_arg0 12>)
p56 = new_with_vtable(138373088)
setfield_gc(p52, ConstPtr(ptr17), descr=<FieldP Application.inst_function 8>)
setfield_gc(p56, ConstPtr(ptr57), descr=<FieldP Application1.inst_arg0 12>)
setfield_gc(p52, p56, descr=<FieldP Application3.inst_arg2 20>)
p58 = new_with_vtable(138371968)
setfield_gc(p45, p0, descr=<FieldP StackElement.inst_next 8>)
setfield_gc(p45, 2, descr=<FieldS CopyStackElement.inst_index 16>)
setfield_gc(p56, ConstPtr(ptr25), descr=<FieldP Application.inst_function 8>)
setfield_gc(p52, p38, descr=<FieldP Application3.inst_arg1 16>)
setfield_gc(p52, ConstPtr(ptr29), descr=<FieldP Application3.inst_arg0 12>)
setfield_gc(p49, p52, descr=<FieldP Application1.inst_arg0 12>)
setfield_gc(p47, p49, descr=<FieldP Application3.inst_arg2 20>)
setfield_gc(p45, p47, descr=<FieldP CopyStackElement.inst_application 12>)
setfield_gc(p58, i42, descr=<FieldS Int.inst_value 8>)
jump(p45, p58, descr=TargetToken(-1223131072))
--end of the loop--
[e68b77b8d02] jit-log-opt-loop}
```

Listing B.5: Addition benchmark: trace after elidable, page 2

APPENDIX C

External Core and JSCore examples

Listing C.1 is the Haskell source code of an example that show how the *extcore* package split up lambda-statements, and how `core2js`'s JSCore output is verbose. The example is a simple function that produces a list of a certain length where each element is a certain integer. The external Core output produced by GHC can be seen in Listing C.2, while Listing C.3 is the JSCore representation (which has been re-formatted to fit the page).

```
main = print (length (repeat' 10 5))

repeat' :: Int -> Int -> [Int]
repeat' 0 _ = []
repeat' n x = x : repeat' (n - 1) x
```

Listing C.1: Double lambda example: Haskell source code

```
%module main:Main
  %rec
  {repeat'r9Y :: ghc-prim:GHC.Types.Int ->
                 ghc-prim:GHC.Types.Int ->
                 (ghc-prim:GHC.Types.[] ghc-prim:GHC.Types.Int) =
      \ (dsdrE::ghc-prim:GHC.Types.Int)
        (ds1drF::ghc-prim:GHC.Types.Int) ->
          %case ((ghc-prim:GHC.Types.[] ghc-prim:GHC.Types.Int)) dsdrE
          %of (wildX6::ghc-prim:GHC.Types.Int)
            {ghc-prim:GHC.Types.I# (ds2drG::ghc-prim:GHC.Prim.Int#) ->
               %case ((ghc-prim:GHC.Types.[] ghc-prim:GHC.Types.Int)) ds2drG
               %of (ds3XrM::ghc-prim:GHC.Prim.Int#)
                 {%_ ->
                    ghc-prim:GHC.Types.: @ ghc-prim:GHC.Types.Int ds1drF
                    (repeat'r9Y
                     (base:GHC.Num.- @ ghc-prim:GHC.Types.Int
                      base:GHC.Num.$fNumInt wildX6
                      (ghc-prim:GHC.Types.I# (1::ghc-prim:GHC.Prim.Int#)))
                     ds1drF);
                  (0::ghc-prim:GHC.Prim.Int#) ->
                    ghc-prim:GHC.Types.[] @ ghc-prim:GHC.Types.Int}}};
  main:Main.main :: (ghc-prim:GHC.Types.IO
                     ghc-prim:GHC.Tuple.()) =
    base:System.IO.print @ ghc-prim:GHC.Types.Int
    base:GHC.Show.$fShowInt
    (base:GHC.List.length @ ghc-prim:GHC.Types.Int
     (repeat'r9Y
      (ghc-prim:GHC.Types.I# (10::ghc-prim:GHC.Prim.Int#))
      (ghc-prim:GHC.Types.I# (5::ghc-prim:GHC.Prim.Int#))));
  main::Main.main :: (ghc-prim:GHC.Types.IO
                      ghc-prim:GHC.Tuple.()) =
    base:GHC.TopHandler.runMainIO @ ghc-prim:GHC.Tuple.()
    main:Main.main;
```

Listing C.2: Double lambda example: external Core output (Z-decoded)

```
{"%module": ["main:Main", ""], "tdef": [],
 "vdefg": [{"%rec": [{"qvar": "repeat'r9Y",
  "ty": {"bty": {"bty": {"qtycon": ["ghc-prim:GHC.Prim", "(->)"]},
                 "aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}},
         "aty": {"bty": {"bty": {"qtycon": ["ghc-prim:GHC.Prim", "(->)"]},
                         "aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}},
                 "aty": {"bty": {"qtycon": ["ghc-prim:GHC.Types", "[]"]},
                         "aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}}},
    "exp": {"lambda":
      {"vbind": {"var": "dsdrE", "ty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}},
      "exp": {"lambda":
        {"vbind": {"var": "ds1drF", "ty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}},
        "exp": {"%case": {"bty": {"qtycon": ["ghc-prim:GHC.Types", "[]"]},
                          "aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}},
          "exp": {"qvar": "dsdrE"},
          "%of": {"var": "wildX6", "ty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}},
          "alt": [{"qdcon": ["ghc-prim:GHC.Types", "I#"], "tyvar": [],
            "vbind": [{"var": "ds2drG", ty": {"qtycon": ["ghc-prim:GHC.Prim", "Int#"]}}],
            "exp": {"%case": {"bty": {"qtycon": ["ghc-prim:GHC.Types", "[]"]},
                              "aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}},
              "exp": {"qvar": "ds2drG"},
              "%of": {"var": "ds3XrM", "ty": {"qtycon": ["ghc-prim:GHC.Prim", "Int#"]}},
              "alt": [
                {"%_":{
                  "aexp":{"aexp":{"aexp": {"qdcon": ["ghc-prim:GHC.Types", ":"]},
                  "args": {"aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}},
                    "args": {"aexp": {"qvar": "ds1drF"}}},
                      "args": {"aexp": {"aexp": {"aexp": {"qvar": "repeat'r9Y"},
                        "args": {"aexp": {"aexp": {"aexp": {"aexp":
                          {"aexp": {"qvar": ["base:GHC.Num", "-"]},
                          "args": {"aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}},
                            "args": {"aexp": {"qvar": ["base:GHC.Num", "$fNumInt"]}}},
                              "args": {"aexp": {"qvar": "wildX6"}}},
                                "args": {"aexp": {"aexp":
                                  {"qdcon": ["ghc-prim:GHC.Types", "I#"]},
                                  "args": {"aexp": {"lit": {"int": 1},
                                  "ty": {"qtycon": ["ghc-prim:GHC.Prim", "Int#"]}}}}}}}}}},
                                    "args": {"aexp": {"qvar": "ds1drF"}}}}}},
                {"lit": {"lit": {"int": 0}, "ty": {"qtycon": ["ghc-prim:GHC.Prim", "Int#"]}},
                  "exp": {"aexp": {"qdcon": ["ghc-prim:GHC.Types", "[]"]},
                    "args": {"aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}}}]}}}}]}}}}]},

  {"qvar": ["main:Main", "main"],
    "ty": {"bty": {"qtycon": ["ghc-prim:GHC.Types", "IO"]},
    "aty": {"qtycon": ["ghc-prim:GHC.Tuple", "()"]}},
    "exp": {"aexp": {"aexp": {"aexp": {"qvar": ["base:System.IO", "print"]},
    "args": {"aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}},
      "args": {"aexp": {"qvar": ["base:GHC.Show", "$fShowInt"]}}},
        "args": {"aexp": {"aexp": {"aexp": {"qvar": ["base:GHC.List", "length"]},
          "args": {"aty": {"qtycon": ["ghc-prim:GHC.Types", "Int"]}}},
            "args": {"aexp": {"aexp": {"aexp": {"qvar": "repeat'r9Y"},
              "args": {"aexp": {"aexp": {"qdcon": ["ghc-prim:GHC.Types", "I#"]},
                "args": {"aexp": {"lit": {"int": 10},
                  "ty": {"qtycon": ["ghc-prim:GHC.Prim", "Int#"]}}}}},
                  "args": {"aexp": {"aexp": {"qdcon": ["ghc-prim:GHC.Types", "I#"]},
                    "args": {"aexp": {"lit": {"int": 5},
                    "ty": {"qtycon": ["ghc-prim:GHC.Prim", "Int#"]}}}}}}}}}},
  {"qvar": ["main::Main", "main"],
    "ty": {"bty": {"qtycon": ["ghc-prim:GHC.Types", "IO"]},
    "aty": {"qtycon": ["ghc-prim:GHC.Tuple", "()"]}},
    "exp": {"aexp": {"aexp": {"qvar": ["base:GHC.TopHandler", "runMainIO"]},
    "args": {"aty": {"qtycon": ["ghc-prim:GHC.Tuple", "()"]}}},
      "args": {"aexp": {"qvar": ["main:Main", "main"]}}}}]}
```

Listing C.3: Double lambda example: `core2js`'s JSCore output

APPENDIX D

Z-encoding

Table D.1 list the encoding rules for Z-encoding[1] by GHC when serializing Core into external Core. Tuples with # are unboxed. A char is encoded as a "z" followed by the char's hex code followed by an "U".

[1] Z-encoding rules are explained in more detail on Haskell Wiki:
`http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/SymbolNames`

Table D.1: Z-encoding

Characters	Code
Tuples:	
()	Z0T
(„)	Z3T
(# #)	Z1H
Constructors:	
(	ZL
)	ZR
[	ZM
]	ZN
:	ZC
Z	ZZ
Variables:	
z	zz
&	za
|	zb
^	zc
$	zd
=	ze
>	zg
#	zh
.	zi
<	zl
-	zm
!	zn
+	zp
'	zq
\	zr
/	zs
*	zt
_	zu
%	zv
char	znnnU

Bibliography

[1] 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.

[2] Håkan Ardö, Carl Friedrich Bolz, and Maciej Fijałkowski. Loop-Aware Optimizations in PyPy's Tracing JIT. Unpublished draft, `http://bitbucket.org/pypy/extradoc/src/a88377852aa3/talk/iwtc11/licm.pdf` [Online; accessed 26-10-2012], December 2011.

[3] John Aycock. A Brief History of Just-In-Time. *ACM Comput. Surv.*, 35(2): 97–113, June 2003. ISSN 0360-0300.

[4] Carl Friedrich Bolz. Controlling the Tracing of an Interpreter With Hints, Part 1: Controlling the Extent of Tracing. `http://morepypy.blogspot.com/2011/03/controlling-tracing-of-interpreter-with.html`, March 2011. [Online; accessed 28-12-2012].

[5] Carl Friedrich Bolz and Laurence Tratt. The Impact of Meta-Tracing on VM Design and Implementation. To appear in 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, Jan Christiansen, Knut Halvor Skrede, and Even Wiik Thomassen. Haskell-Python bitbucket.org project; source repository. `http://bitbucket.org/cfbolz/haskell-python/`, 2012. [Online; accessed 30-08-2012].

[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] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 10–18. ACM, 2007. ISBN 978-1-59593-690-5.

[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] Richard Eisenberg. System FC, as implemented in GHC. `http://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf?raw=true`, December 2012. [Online; accessed 07-12-2012].

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

[17] Sven Hager. Implementing the R Language Using RPython. Master's thesis, Heinrich-Heine-Universität Düsseldorf, Düsseldorf, Germany, October 2012.

[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] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in Haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272. ACM, 2010. ISBN 978-1-60558-794-3.

[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] Simon Marlow et al. Haskell 2010 Language Report. `http://www.haskell.org/onlinereport/haskell2010/`, April 2010. [Online; accessed 15-10-2012].

[24] 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.

[25] David M Peixotto. *Low-Level Haskell Code: Measurements and Optimization Techniques*. PhD thesis, Rice University, Houston, Texas, USA, 2012.

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

[27] Simon Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 327–337, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2.

[28] Simon Peyton Jones and James Fischer. Haskell Bug Tracker: Ticket 5844 - Panic on generating Core code. `http://hackage.haskell.org/trac/ghc/ticket/5844`, 2012. [Online; accessed 11-12-2012].

[29] 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.

[30] Simon Peyton Jones and Will Partain. Measuring the effectiveness of a simple strictness analyser. *Functional Programming, Glasgow*, pages 201–220, 1993.

[31] 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.

[32] Simon Peyton Jones and Andre Santos. Compilation by transformation in the Glasgow Haskell Compiler. *Functional Programming, Glasgow*, pages 184–204, 1994.

[33] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN international conference on Functional programming*, ICFP '96, pages 1–12, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7.

[34] Simon Peyton Jones, Simon Marlow, et al. GHC source code, git repository. `http://darcs.haskell.org/ghc.git/`, 2012. [Online; accessed 29-12-2012].

[35] 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 12-10-2012] `http://www.haskell.org/onlinereport/`.

[36] 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.

[37] Thomas Schilling. Re: Trace-based JIT of Haskell. Email to Even Wiik Thomassen. September 22, 2012.

[38] Thomas Schilling. Challenges for a Trace-Based Just-In-Time Compiler for Haskell. In *Implementation and Application of Functional Languages*, volume 7257 of *Lecture Notes in Computer Science*, pages 51–68. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-34406-0.

[39] David Schneider. Implementation of the Io language in RPython; source repository. `http://bitbucket.org/pypy/lang-io/`, 2009–2011. [Online; accessed 15-09-2012].

[40] David Schneider and Carl Friedrich Bolz. The Efficient Handling of Guards in the Design of RPython's Tracing JIT. In *6th workshop on virtual machines and intermediate languages, VMIL*, 2012.

[41] 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.

[42] 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 06-12-2012].

[43] Knut Halvor Skrede. Just-In-Time compilation of Haskell with PyPy and GHC. Master's thesis, Norwegian University of Science and Technology, Trondheim, Norway, June 2012.

[44] 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.

[45] The PyPy team. PyPy Homepage — the PyPy Speed Center. `http://speed.pypy.org/`, 2012. [Online; accessed 13-12-2012].

[46] The PyPy team. PyPy bitbucket.org project; source repository. `http://bitbucket.org/pypy/pypy/`, 2012. [Online; accessed 16-12-2012].

[47] David A. Terei and Manuel M.T. Chakravarty. An LLVM backend for GHC. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 109–120, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4.

[48] Andrew Tolmach, Tim Chevalier, et al. An External Representation for the GHC Core Language. `http://www.haskell.org/ghc/docs/7.4.1/core.pdf`, February 2012. [Online; accessed 01-10-2012].

[49] 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 12-10-2012].

[50] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: a compiler pearl. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, ICFP '12, pages 341–352. ACM, 2012. ISBN 978-1-4503-1054-3.

[51] Stephanie Weirich, Dimitrios Vytiniotis, Simon Peyton Jones, and Steve Zdancewic. Generative Type Abstraction and Type-Level Computation. In

Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '11, pages 227–240, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0.

[52] 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.

List of tables

List of listings

List of abbreviations

ABI application binary interface 56

API application programming interface 7, 9, 11, 21, 22, 54, 59

AST abstract syntax tree 7, 29, 51

CLR Common Language Runtime 16, 17

Cmm C minus minus 9, 27, 55, 56

DPH Data Parallel Haskell 56

GADT generalized algebraic data types 12, 20

GCC GNU Compiler Collection 9, 48

GHC Glasgow Haskell Compiler 2, 3, 5–7, 9, 11, 19–23, 25–27, 29, 30, 43, 47–49, 51, 53–57, 59, 69, 73

GHCi GHC's interactive environment 7

IR intermediate representation 9

JIT just-in-time 1–3, 13–18, 22, 25, 29, 33–35, 37–39, 41, 46–49, 51, 53, 55, 57–59, 63

JSON JavaScript Object Notation 2, 19, 20, 27, 54

JVM Java Virtual Machine 13, 16, 17

KLoC thousand lines of code 56

LINQ Language Integrated Query 5

NCG native code generator 9, 51, 55, 56, 59

SSA static single assignment 16, 34, 56

STG Spineless Tagless G-machine 7, 9, 55, 56

TNTC tables-next-to-code 9, 27, 48, 57

VM virtual machine 1–3, 14–19, 25–27, 33, 34, 41, 49, 53, 54, 57, 58