Just-In-Time compilation of Haskell using PyPy and GHC

Knut Halvor Skrede

March 7, 2012

Abstract

The paper describes a system for using GHC as frontend and PyPy as backend for a Haskell JIT compiler. An intermediate language in JSON format based on GHC's Core language is described. The implemented parts are a serializer written in Haskell and a deserializer written in Python, in addition to some Haskell library functions. The project is meant to serve as a base for further development.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Project description

The project focuses on implementing a serializer from Haskell to an intermediate format using GHC, and a deserializer from the intermediate format to an interpreter using PyPy. And then to test the result on some simple Haskell programs. The hope is that this can serve as a base for future development into a full Haskell JIT compiler.

1.2 Motivation

The motivation behind the project is to see if a *strongly-typed non-strict purely-functional* programming language, Haskell, can benefit from just-in-time (JIT) compilation.

Some people seem to think that there is little to gain from using JIT compilation of strongly typed languages since they can be so heavily optimized at compile time. However, a JIT compiler has a lot more information to work with. Speedup in JIT compilers is achieved by feeding information observed at runtime into compilation and exploit it. For example, by locating variables that change very slowly, the JIT compiler can optimize multiple instances of the code, one for each variation of the variable. [2]

PyPy is a project that implements a meta-tracing JIT. The project defines a proper subset of Python called RPython. This language has the characteristics that it is possible to perform type-inference on it. Because of this, interpreters written in RPython can be translated to efficient C code. The idea is that interpreters can be written rapidly in RPython, and the interpreter implemented in RPython will benefit from PyPy's JIT compiler. To optimize the interpreter, the RPython toolchain accepts some compiler hints to determine what parts of the code to trace, and to define the static and dynamic parts of the interpreter "memory". [2]

GHC (Glasgow Haskell Compiler) is the most advanced compiler for the Haskell programming language. The GHC team has defined an external format for Core (Core is an intermediate format used by GHC) called *External Core*. GHC's front-end translates Haskell 98 (plus some extensions) into Core. This way it is possible to reuse GHC for parsing, desugaring and typechecking, while implementing the back-end separately. [7]

1.3 System description

The system will use GHC to generate the representation of the Core language and a simple Haskell program to serialize this representation into JSON. This entails using GHC for parsing, typechecking, desugaring and simplification. Simplification is an optimization step in GHC, performing small and simple transformations.

The resulting code will then be deserialized by the PyPy Haskell interpreter and executed. See figure 1 for a description of the implemented and to-be-implemented parts of the project.

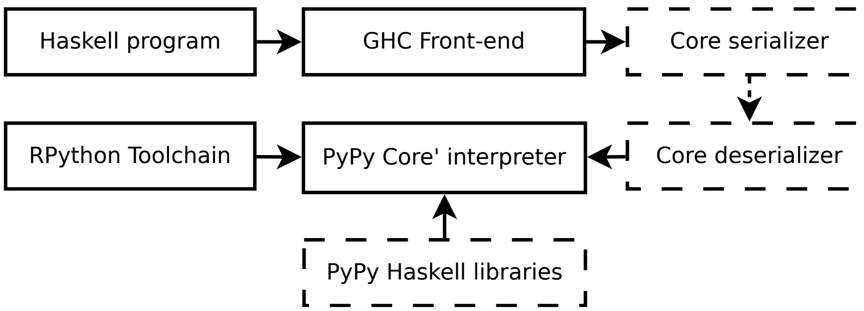

Figure 1: Project overview. Full border: implemented, Dotted border: unimplemented.

1.4 Structure of the paper

Following this introduction is a section discussing the background of the project. The background section contains some information about the tools being used, and why. Section 3 contains a description of the external language representation of the intermediate language used by GHC. Section 4 follows with the description of the intermediate format used in this project, which is a combination of JSON and External-core. Section 5 discusses the implementation of the system, the issues faced during development and some of the remaining issues. In section 6, example programs are passed through the pipeline with intermediate representations presented at each step. Section 7 discusses future work.

2 Background

2.1 PyPy

The PyPy project is basically two things; a toolchain for compiling RPython programs (written in Python), and an implementation of Python using these tools.

In this paper PyPy refers to former.

The basic concept of PyPy is to use a high-level language to allow for rapid development of interpreters for a variety of platforms. By implementing a compiler for RPython, interpreters for other languages can be written in RPython and compiled to any platform supported by the PyPy toolchain. Supported platforms include CLI and JVM. [1]

PyPy uses the meta-programming argument; if a VM (virtual machine) can be written at a level of abstraction high enough, then it should be possible to automatically translate this VM to other lower-level platforms. This is what PyPy does. [6]

2.1.1 RPython

RPython (Restricted Python) is a restricted proper subset of Python that it is possible to perform type inference on. This means that it can be translated to efficient C code, and it enables easy analysis as well as efficient compilation. This also means that RPython code can be run and debugged by Python interpreters, like CPython. [1]

2.1.2 Just-in-time compilation

The JIT compiler is the reason why PyPy is able to compete with other language implementations on speed. Or rather, it's meta-tracing JIT. The JIT is implemented for the RPython compiler, but through a set of compiler hints, it is able to trace the execution of the application interpreted by the RPython program.

2.1.3 Haskell-Python

Haskell-Python is an interpreter for a subset of the Haskell language, called Core'. We call it Core' (Core marked) here because it does not directly correspond to the Core language used by GHC (Glasgow Haskell Compiler). Haskell-Python is written in RPython, and is compiled into a JIT compiler using the RPython toolchain. Our goal for this project is to extend Haskell-Python to use GHC as a frontend for compilation of Haskell programs.

2.2 GHC

GHC (Glasgow Haskell Compiler) is a compiler for the Haskell programming language. *Haskell* is a *strongly-typed non-strict purely-functional* programming language, it will not be described in any detail here, since Haskell is not the language we focus on. See [3] for an introduction to Haskell.

Core is an intermediate language used by the Glasgow Haskell Compiler [4], and it is this language we wish to interpret. Core is a desugared version of Haskell, things like pattern matching and list comprehensions are transformed out to simpler constructs. [5]

In this project, GHC is used to generate an external representation of the Core language, reusing GHC for parsing, typechecking and desugaring.

2.2.1 Extcore

Extcore is a Haskell package for working with GHC's Core language. Among other things, it implements a parser for External-core, this is the part used from extcore in this project.

3 External-core

The Core language is an intermediate language used by GHC. It is the internal program representation in the compilers simplification phase. It appears to be a subset of Haskell, but with explicit type annotations in the style of the polymorphic lambda calculus (F_w). External-core is an external representation of Core generated from Haskell, by using GHC with a compiler flag (*-fext-core*). By using External-core, one may implement just parts of a Haskell compiler, using the remaining parts from GHC. For this project, GHC is used to generate External-core. This way, desugaring, type checking, pattern matching and overloading is performed. The remaining task is then to interpret the External-core representation. [7]

The GHC-generated External-core uses z-encoding for special characters in names (variables, constructors, ...), for a definition of the z-encoding, see appendix B. This is a good thing, as much of the functionality represented by these names has to be implemented in RPython, and special characters (like #, or +) is not allowed in Python names (function identifiers, class identifiers, ...).

Without using GHC to produce External-core, linking code into GHC would be an optional way of achieving this, which would be a difficult and large task. Or, the GHC API could be used to do the same task more cleanly. [7]

The initial starting point of this project was to use the GHC API, the reasoning was that External-core is not fully parenthesized and more tricky to parse. Thus, generating a fully parenthesized and more machine-readable format would make sense. However, it turned out that this too was a complicated task. As the internal datatypes of GHC representing Core does not match the description of External-core. The choice was then made to use GHC-generated External-core as the base for creating a new intermediate format that could be easily generated using available packages for manipulating External-core (*extcore* is used for this).

For a formal definition of External-core, see appendix A.

3.1 Informal semantics

3.1.1 Module

The first construct in the External-core representation of a program, is the *module*. This construct is represented by a *module identifier*, followed by a list of *type definitions* and *value definitions* respectively.

The *module* directly corresponds to Haskell source modules. The *module identifier* contains information of what package it belongs to, followed by its name. *Identifiers* that are defined at the top level of the module can be internal or external. External identifiers can be referenced from other modules in the program, internal identifiers can not. [7]

3.1.2 Type definition

A *type definition* can be an *algebraic-type*, or a *newtype* construct.

An *algebraic-type* is represented by the "%data" keyword, a *type constructor* and a *type binder*, followed by a list of *constructor definitions*. Each new *algebraic type* introduces a new *type constructor* and a set of one or more *constructor definitions*.

A *newtype* construct is represented by the "%newtype" keyword, two *type constructors*, a set of *type binders* and a *type*. [7]

3.1.3 Constructor definition

A *constructor definition* is represented by a *data constructor*, followed by a *type binder* and one or more *atomic types*.

3.1.4 Value definition

A *value definition* can be *recursive* or *non-recursive*. A *recursive value definition* is represented by the "%rec" keyword and one or more *non-recursive value definitions*. A *non-recursive value definition* is represented by a *variable identifier* followed by a *type* and an *expression*. [7]

3.1.5 Atomic expression

An *atomic expression* can be either of the following; a *variable*, *data constructor*, *literal* or a *nested expression*. [7]

3.1.6 Expression

An *expression* can be either of the followin; an *atomic expression*, *application*, (lambda) *abstraction*, *local definition*, *case expression*, *type coercion*, *expression note*, *external reference*, *dynamic external reference*, or an *external label*. See appendix A table 2 for their definitions. [7]

3.1.7 Argument

An *argument* can be either a *type argument* or a *value argument*. A *type argument* is simply an *atomic type*, and a *value argument* is an *atomic expression*.

Arguments are applied to *atomic expressions* to make *applications*. [7]

3.1.8 Case alternative

A *case alternative* can be either of the following; a *constructor alternative*, a *literal alternative* or a *default alternative*. [7]

3.1.9 Binder

A *binder* can be either a *type binder* or a *value binder*. It binds a *value* to a *type*, or a *type* to a *kind*. A *type binder* is represented by a "@" and a *type binder*. A *value binder* is simply represented as a *value binder*.

A *type binder* can either be a *type variable* (implicitly of kind *), or a *type variable* followed by a *kind* (explicitly kinded).

A *value binder* is a *variable* followed by a *type*. [7]

3.1.10 Literal

A *literal* can be either an *integer*, *rational*, *character* or *string*. See appendix A table 2 for their definitions. [7]

3.1.11 Atomic type

An *atomic type* can be either a *type variable*, *type constructor* or a *nested type*. [7]

3.1.12 Basic type

A *basic type* can be either of the following; an *atomic type*, *type application*, *transitive coercion*, *symmetric coercion*, *unsafe coercion*, *left coercion*, *right coercion*, or an *instantiation coercion.* [7]

3.1.13 Type

A *type* can be either a *basic type*, a *type abstraction* or a *arrow type construction.*

Types are built from *type constructors* and *type variables* using *type application* and *universal quantification.* There are a number of primitive *type constructors* defined in the "GHC.Prim" module. *algebraic-type* and *newtype* constructs introduce new *type constructors.* The *type constructors* are distinguished by name only. [7]

3.1.14 Atomic kind

An *atomic kind* can be either of the following; a *lifted kind* (*), *unlifted kind* (#), *open kind* (?), *equality kind*, or a *nested kind.* [7]

3.1.15 Kind

A *kind* can be either an *atomic kind* or an *arrow kind.* An *arrow kind* is an *atomic kind* followed by an arrow (->) and another *kind.* [7]

3.2 Evaluation of a program

A program is evaluated by reducing the expression "main:ZCMain.main" (note that qualified names in a module named m must have module-name m, this is the only exception) to *weak-head-normal-form* (WHNF), i.e. a primitive value, lambda abstraction, or fully applied data constructor. A heap is used to make sure evaluation is shared. The heap contains two types; a *thunk*, or a *WHNF*. A thunk is an unevaluated expression, also called a *suspension.* A *WHNF* is an evaluated expression, the result of evaluating a *thunk* is a *WHNF.* [7]

4 JSON representation of Core

4.1 JSON

JavaScript Object Notation (JSON) is a lightweight data interchange format.

Since a library for manipulating JSON is available for Haskell, this makes it a good choice for the project. In addition, it is easy to parse and the Haskell library contains a pretty-printer, making the result easier to inspect.

For a formal definition of JSON see appendix C

4.2 JSCore

In order to work with JSON and External-core, a format was defined that expresses the Core program in JSON notation. Most of the right hand side of the grammar evaluates to JSON Values. Even though the grammar is changed to support JSON, an effort was made to keep it similar to the original Core grammar for easy referencing. The size of the resulting files was not considered to be an issue. Other than the syntax, the JSCore language is the same as External-core.

For a formal definition of JSCore see appendix D.

5 Implementation

5.1 Tools and versions

The following tools and package versions was used in the implementation:

- GHC version 7.0.3
- extcore version 1.0.1
- PyPy current head branch (Last tested 09/11/2011)
- Haskell-Python interpreter: Interpreter of Core' written in RPython
- CPython 2.7: Used to test the Core' interpreter without it having to be correct RPython. (pypy-python could also have been used)
- Git was used as the projects version control system.

5.2 Pipeline

The project implements the following pipeline (see figure 3):

1. Serialize Haskell program:
   (a) Create External-core file from Haskell program using GHC.
   (b) Create JSCore from External-core using the extcore and JSON packages.

2. Deserialize JSCore:
    (a) Parse JSCore using the parsing tools available for PyPy
    (b) Build Core AST from resulting JSON datatype
3. Evaluate program:
    (a) Evaluation is done by the already implemented PyPy Core' interpreter, Haskell-Python. Additional functionality had to be built on top of this, mostly Haskell library functions.

5.3 Organization

The implementation is organized as represented by figure 2. The main folder (interpreter) contains the main program, and a program for generating dot[1] files, "makegraph.py" (used to create graphs of parsed JSCore files using graphviz[2]). The "haskell" folder contains the Haskell-Python Core' interpreter code, used by the main program for evaluation, and by the parser to generate the abstract-syntax-tree (AST). In addition to this, the subfolder "ghc" implements some simple functionality to be used by the test-programs. Among others, a very simple IO function for printing text to the terminal (putStrLn). These are categorised into packages corresponding to GHC packages, and libraries, corresponding to the libraries in the GHC packages. The RPython modules in these "packages" are loaded, and references to the attributes they contain are used during the creation of the AST. See figure 3 for a simple description of the pipeline.

The "core" folder contains a Haskell program for generating JSCore files from External-core files (core2js.hs), the JSCore parser (parser.py), and a simple datastructure representing Core modules (module.py).

The "tests" folder naturally contains tests, each test is in its own subfolders.

[1] A dot file is a file written in dot-language, used by the *dot* command line tool from the graphviz software package

[2] Graphviz is an open source software package for graph visualization

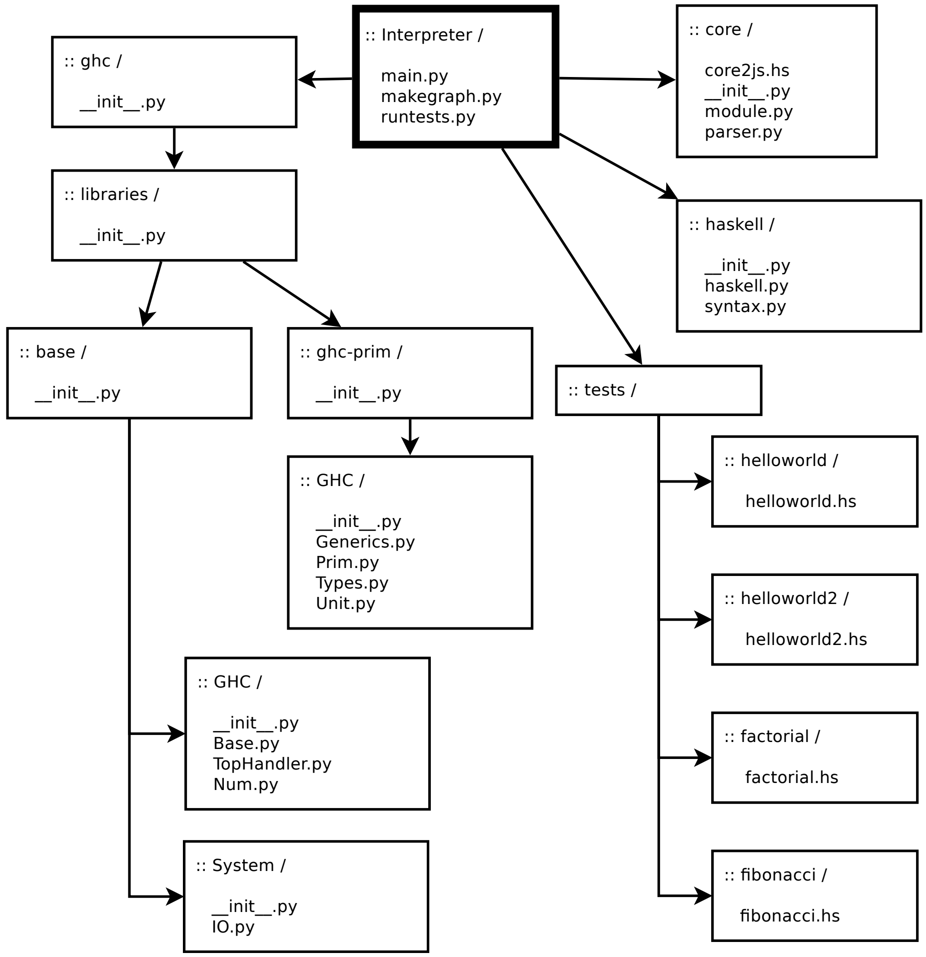

Figure 2: Code tree: The top of the boxes is the folder name, and the rest is the source files. Arrows represent subfolders.

5.4 Serializer

The serializer consists of two parts; GHC generating External-core, and a Haskell program to generate JSCore (core2js.hs).

External-core is easily generated by using a compiler flag:

```
ghc -fext-core {path-to-program}
```

The Haskell program generating JSCore uses the extcore packages. This package implements functionality for working with External-core. The result is a datastructure mapping directly to the External-core format defined in [7]. By traversing this structure, the program builds up a JSON tree using the Haskell JSON package. The result is a tree of JSON constructs (corresponding to the grammar defined in table 6), this is then pretty-printed and dumped to a file.

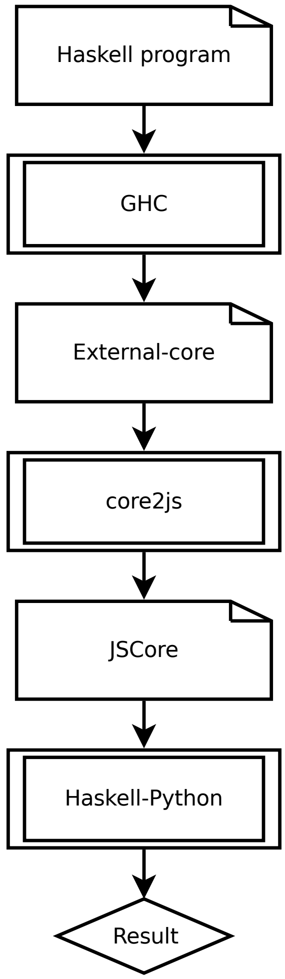

Figure 3: Pipeline implementation with core2js using extcore

5.5 Deserializer

The deserializer implements a JSON parser. The resulting datastructure is then traversed, recursively building up an AST using the constructs defined in the Haskell-Python interpreter.

PyPy implements a parser generator, this simply takes a grammar defined as a string, written in extended-backus-naur-form (EBNF), and generates a parser.

This parser is then used to create a JSON datastructure, as represented by table 4. The resulting datastructure is then traversed. By checking the contents of the JSON constructs with the actual JSCore format, the Core AST is built. External functionality is imported from the Python implementations of the Haskell libraries as it is encountered.

After this is done, we are left with a "module" object, corresponding to the initial Haskell module.

5.6 Haskell libraries

To make some simple test-cases work, some basic Haskell functionality had to be implemented. Some of this functionality was implemented already in the Haskell-Python Core' interpreter. The work done here was mostly to organize the functionality into modules corresponding to Haskell modules. The functionality implemented in these modules does however, not correspond to the Haskell implementations. This is left for future work, as this is a large task.

From figure 6 (representing a "hello world" program in JSCore), the expression "base:SystemziIO.putStrLn" (See appendix B for a definition of the z-encoding, used in this expression identifier) corresponds to the Haskell function "putStrLn", which is located in the Haskell module "System.IO". This is translated into a reference to the function "putStrLn" defined in the python module "ghc.libraries.base.System.IO". See figure 2.

5.7 Evaluation

In order to evaluate the Haskell programs correctly, the expression "main:ZCMain.main" is reduced to *WHNF*. However, to do this correctly would require a lot of the functionality used by GHC to be implemented. Specifically, "GHC.TopHandler.runMainIO". In order to implement this function a lot of other functionality would have to be implemented. The function is a wrapper around "main:Main.main", it catches uncaught exceptions and flushes stdout/stderr before exiting.

Currently, a simple hack is implemented; the function "runMainIO" simply returns the function "main:Main.main". The "main:Main.main" function is the entry-point to our program. This function is then evaluated when reducing "main:ZCMain.main" to WHNF.

Implementing this functionality in a manner similar to GHC is a goal for future development. Currently, the hacks suffice for testing. Simple Haskell functionality is implemented at a high level, such as the "putStrLn" function which is implemented as a simple "print" Python function.

5.8 Tests

Testing is done by a simple Python script. The test directory has a subdirectory for each test. This directory contains a Haskell program. The program is converted to External-core, then to JSCore, and then executed. If the program returns "0" it passes the test. The results of running these tests are given in figure 4.

```
--------------------------------
RESULTS
--------------------------------

factorial : Failure
fibonacci : Failure
helloworld : Success
helloworld2 : Failure
```

Figure 4: Result output from running tests (runtests.py)

The tests are supposed to be in the following increasing order of difficulty: helloworld, helloworld2, factorial and fibonacci. All the failures are due to the same problem; unimplemented primitives and library functions. See appendix E for test programs.

5.9 Issues

Very few of the initial plans regarding this project was actually realized in the implementation. The GHC API was intended to be used to generate JSCore, but this failed do to a lack of experience with Haskell and the GHC API (see figure 5 for a description of the initially intended pipeline). It was then discovered that a lot of the necessary code was already written in GHC (the code that generates the External-core files), however, this code was not exported in any way. It was also deeply embedded in the GHC code. After several unfruitful attempts at creating the JSCore format by using the GHC API and by reimplementing the GHC functions generating External-core. And some emails back and forth with the GHC team, these methods where abandoned.

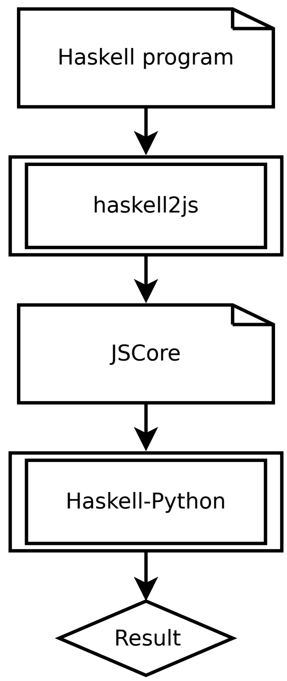

Figure 5: Pipeline with implementation of haskell2js using the GHC API (Not implemented)

The extcore package was then introduced, as it was thought to serve nicely for our purpose. This does however add an extra step to the generation of JSCore, having to generate External-core first. There is no way of working with External-core without parsing it from a file. See figure 3.

The reason for not using *extcore* from the beginning was that it was thought to not be well supported. As well as the seeming lack of support for the External-core format.

A large amount of time was spent trying to generate this intermediate format. And the greatest obstacle was that the tools being used was either incompatible, or lacking in documentation.

The method described here worked well for very trivial Haskell programs, however, it turned out that the extcore package was not able to parse any nontrivial External-core files generated by the GHC version used. It was thought that this was due to the fact that the extcore package was written for earlier versions of GHC. After unsuccessfully attempting to make the tools work with earlier versions (GHC 6.10 and 6.12 on linux and 6.10 on windows), it turned out that the *extcore* package implemented two parsers. One of which was outdated. This was not documented anywhere, but realized after e-mailing the package maintainer, who turned out to be very helpful.

6 Example

The example is a simple "hello world" program, as this was practically the only program that was able to pass through the entire pipeline. Following is the "hello world" program written in Haskell.

6.1 Haskell code

```
main = putStrLn "Hello, world!"
```

6.2 Converted to Core

After the program has passed through GHC and the External-core file has been generated, the program looks like this:

```
%module main:Main
  main:Main.main :: (ghczmprim:GHCziTypes.IO
                     ghczmprim:GHCziUnit.Z0T) =
    base:SystemziIO.putStrLn
    (base:GHCziBase.unpackCStringzh
     ("Hello, world!"::ghczmprim:GHCziPrim.Addrzh));
  main:ZCMain.main :: (ghczmprim:GHCziTypes.IO
                       ghczmprim:GHCziUnit.Z0T) =
    base:GHCziTopHandler.runMainIO @ ghczmprim:GHCziUnit.Z0T
    main:Main.main;
```

6.3 Converted to JSCore

In the next step it is parsed and dumped to JSCore:


```
{"%module": "main:Main", "tdefg": [],
 "vdefg": [{"qvar": "main:Main.main",
            "ty": {"bty": {"qtycon": "ghczmprim:GHCziTypes.IO"},
                   "aty": {"qtycon": "ghczmprim:GHCziUnit.Z0T"}},
            "exp": {"aexp": {"qvar": "base:SystemziIO.putStrLn"},
                    "args": {"aexp": {"aexp": {"qvar": "base:GHCziBase
                        .unpackCStringzh"},
                                  "args": {"aexp": {"lit": "Hello,
                                      world!"}}}}}},
           {"qvar": "main:ZCMain.main",
            "ty": {"bty": {"qtycon": "ghczmprim:GHCziTypes.IO"},
                   "aty": {"qtycon": "ghczmprim:GHCziUnit.Z0T"}},
            "exp": {"aexp": {"aexp": {"qvar": "base:GHCziTopHandler.
                runMainIO"},
                             "args": {"aty": {"qtycon": "ghczmprim:
                                 GHCziUnit.Z0T"}}},
                    "args": {"aexp": {"qvar": "main:Main.main"}}}}]}
```

6.4 JSCore graph

Using some libraries from PyPy we can generate a nice graph, directly corresponding to the datastructure generated from parsing. See figure 6. By simply traversing this datastructure we can generate the AST for the Core' interpreter (Haskell-Python).

6.5 Result

The program results as expected, outputting the string "Hello, world!".

7 Future work

7.1 Rewrite the deserializer to proper RPython

The deserializer is currently not written in proper RPython. Converting the code to RPython will allow it to be compiled to a JIT interpreter by the RPython toolchain. This should not be a very big task, but it requires understanding of the RPython coding style. There are also restrictions on how one may use the PyPy parser tools.

7.2 Implement basic Haskell libraries

Implementing Haskell libraries is necessary to run any Haskell program passed through GHC. One option may be to implement (or automatically generate from GHC code) Haskell primitive types, and to convert the Haskell base libraries to JSCore. Compiling GHC with the $-fext-core$ flag should be possible, and should generate External-core files for all library modules.

7.3 Linking functionality

Currently no thought has been given to the linking between multiple modules. This must be implemented for any non-trivial Haskell programs to function.

7.4 Test suite and benchmarking

A framework for testing and benchmarking should be set up. Allowing us to compare speed to other implementations, and progress of development. GHC uses a test framework relying on Python and GNU Make that should be looked into.

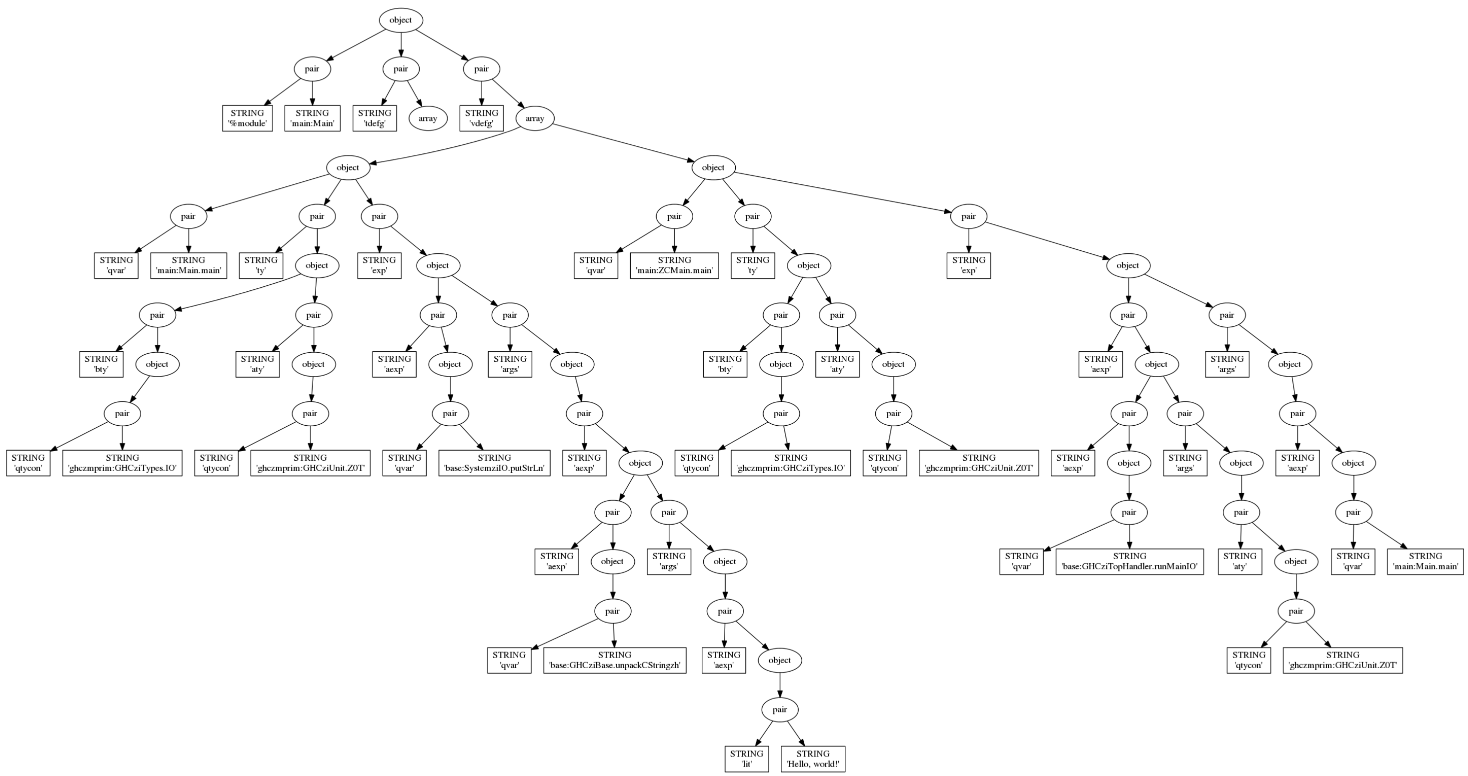

Figure 6: Example program translated to JSCore

7.5 Optimize JIT compiler for Haskell-Python

Further optimization of the JIT compiler for the Haskell-Python interpreter will be necessary to achieve good speed.

Appendices

A Formal definition of External-core

The following semantics is used to define the Core grammar, as seen in [7]:

[pat]	:	optional
{ pat }	:	zero or more repetitions
{ pat }$^+$	:	one or more repetitions
$pat_1 \| pat_2$	:	choice

Module

module	→	%module *mident* { *tdefg* ; } { *vdefg* ; }	

Type defn.

tdefg	→	%data *qtycon* { *tbind* } = { [*cdef* { ; *cdef* }] }	algebraic type
	|	%newtype *qtycon* *qtycon* {*tbind*} = *ty*	newtype

Constr. defn.

cdef	→	*qdcon* { @ *tbind* } { *aty* }$^+$	

Value defn.

vdefg	→	%rec { *vdef* { ; *vdef* } }	recursive
	|	*vdef*	non-recursive
vdef	→	*qvar* :: *ty* = *exp*	

Atomic expr.

aexp	→	*qvar*	variable
	|	*qdcon*	data constructor
	|	*lit*	literal
	|	(*exp*)	nested expr.

Expression

exp	→	*aexp*	atomic expr.
	|	*aexp* { *arg* }$^+$	application
	|	\ { *binder* } -> *exp*	abstraction
	|	%let *vdefg* %in *exp*	local definition
	|	%case (*aty*) *exp* %of *vbind* { *alt* { ; *alt* } }	case expr.
	|	%cast *exp* *aty*	type coercion
	|	%note " { *char* } " *exp*	expression note
	|	%external ccal " { *char* } " *aty*	external reference
	|	%dynexternal ccal *aty*	external reference (dynamic)
	|	%label " { *char* } "	external label

Argument

arg	→	@ *aty*	type argument
	|	*aexp*	value argument

Case alt

alt	$\rightarrow$	*qdcon* { @ *tbind* } { *vbind* } -> *exp*	constructor alternative
	$\mid$	*lit* -> *exp*	literal alternative
	$\mid$	%_ -> *exp*	default alternative

Binder

binder	$\rightarrow$	@ *tbind*	type binder
	$\mid$	*vbind*	value binder

Type binder

tbind	$\rightarrow$	*tyvar*	implicit of kind *
	$\mid$	(*tyvar* :: *kind*)	explicitly kinded

Value binder

vbind	$\rightarrow$	(*var* :: *ty*)	

Literal

lit	$\rightarrow$	([-] { *digit* }$^{+}$ :: *ty*)	integer
	$\mid$	([-] { *digit* }$^{+}$ % { *digit* }$^{+}$ :: *ty*)	rational
	$\mid$	(' *char* ' :: *ty*)	character
	$\mid$	(" { *char* } " :: *ty*)	string

Character

char	$\rightarrow$	Any ASCII character in range 0x20-0x7E except 0x22, 0x27, 0x5c	
	$\mid$	\x *hex* *hex*	ASCII code escape sequence
hex	$\rightarrow$	0 $\mid$ ... $\mid$ 9 $\mid$ a $\mid$ ... f	

Atomic type

aty	$\rightarrow$	*tyvar*	type variable
	$\mid$	*qtycon*	type constructor
	$\mid$	(*ty*)	nested type

Basic type

bty	$\rightarrow$	*aty*	atomic type
	$\mid$	*bty* *aty*	type application
	$\mid$	%trans *aty* *aty*	transitive coercion
	$\mid$	%sym *aty*	symmetric coercion
	$\mid$	%unsafe *aty* *aty*	unsafe coercion
	$\mid$	%left *aty*	left coercion
	$\mid$	%right *aty*	right coercion
	$\mid$	%inst *aty* *aty*	instantiation coercion

Type

ty	$\rightarrow$	*bty*	basic type
	$\mid$	%forall { *tbind* }$^{+}$. *ty*	type abstraction
	$\mid$	*bty* -> *ty*	arrow type construction

Atomic kind

akind	$\rightarrow$	*	lifted kind
	$\mid$	#	unlifted kind
	$\mid$	?	open kind
	$\mid$	*bty* :=: *bty*	equality kind
	$\mid$	(*kind*)	nested kind

Kind

kind	$\rightarrow$	*akind*	atomic kind
	$\mid$	*akind* -> *kind*	arrow kind

Identifier

mident	$\rightarrow$	*pname* : *uname*	module
tycon	$\rightarrow$	*uname*	type constr.
qtycon	$\rightarrow$	*mident* . *tycon*	qualified type constr.
tyvar	$\rightarrow$	*lname*	type variable
dcon	$\rightarrow$	*uname*	data constr.
qdcon	$\rightarrow$	*mident* . *dcon*	qualified data constr.
var	$\rightarrow$	*lname*	variable
qvar	$\rightarrow$	[*mident* .] *var*	optionally qualified variable

Name

lname	$\rightarrow$	*lower* { *namechar* }
uname	$\rightarrow$	*upper* { *namechar* }
pname	$\rightarrow$	$\{ \, namechar \, \}^{+}$
namechar	$\rightarrow$	*lower* | *upper* | *digit*
lower	$\rightarrow$	a|b|...|z|_
upper	$\rightarrow$	A|B|...|Z|
digit	$\rightarrow$	0|1|...|9

Table 2: Grammar for External-core

B Z-Encoding

Character	Code
Tuples:	
(##)	Z1H
()	Z0T
(,,,)	Z3T
Constructors:	
(	ZL
)	ZR
[	ZM
]	ZN
:	ZC
Z	ZZ
Variables:	
z	zz
&	za
—	zb
ˆ	zc
$	zd
=	ze
¿	zg
#	zh
.	zi
¡	zl
-	zm
!	zn
+	zp
’	zq
\	zr
/	zs
*	zt
_	zu
%	zv
c	znnnU

Table 3: GHC z-encoding

C Formal definition of JSON

Object

$$
\begin{array}{rcl}
\mathit{object} & \rightarrow & \{\ \} \\
 & | & \{\ \mathit{members}\ \} \\
\mathit{members} & \rightarrow & \mathit{pair} \\
 & | & \mathit{pair}\ ,\ \mathit{members} \\
\mathit{pair} & \rightarrow & \mathit{string}\ :\ \mathit{value}
\end{array}
$$

Array

$$
\begin{array}{rcl}
\mathit{array} & \rightarrow & [\ ] \\
 & | & [\ \mathit{elements}\ ] \\
\mathit{elements} & \rightarrow & \mathit{value} \\
 & | & \mathit{value}\ ,\ \mathit{elements}
\end{array}
$$

Value

$$
\begin{array}{rcl}
\mathit{value} & \rightarrow & \mathit{string} \\
 & | & \mathit{number} \\
 & | & \mathit{object} \\
 & | & \mathit{array} \\
 & | & \text{true} \\
 & | & \text{false} \\
 & | & \text{null}
\end{array}
$$

String

$$
\begin{array}{rcl}
\mathit{string} & \rightarrow & \text{""} \\
 & | & \text{"}\ \mathit{chars}\ \text{"} \\
\mathit{chars} & \rightarrow & \mathit{char} \\
 & | & \mathit{char}\ \mathit{chars} \\
\mathit{char} & \rightarrow & \text{any Unicode character except " } \\
 & & \text{or } \backslash \text{ or control characters:} \\
 & & \backslash\backslash \\
 & & \backslash/ \\
 & & \backslash b \\
 & & \backslash f \\
 & & \backslash n \\
 & & \backslash r \\
 & & \backslash t \\
 & & \backslash u\ \text{four-hex digits}
\end{array}
$$

Number

$$
\begin{array}{rcl}
\mathit{number} & \rightarrow & \mathit{int} \\
 & | & \mathit{int}\ \mathit{frac} \\
 & | & \mathit{int}\ \mathit{exp} \\
 & | & \mathit{int}\ \mathit{frac}\ \mathit{exp} \\
\mathit{int} & \rightarrow & \mathit{digit} \\
 & | & \mathit{digit}1-9\ \mathit{digits} \\
 & | & \text{-}\ \mathit{digit} \\
 & | & \text{-}\ \mathit{digit}1-9\ \mathit{digits} \\
\mathit{frac} & \rightarrow & .\ \mathit{digits} \\
\mathit{exp} & \rightarrow & \mathit{e}\ \mathit{digits} \\
\mathit{digits} & \rightarrow & \mathit{digit} \\
 & | & \mathit{digit}\ \mathit{digits} \\
\mathit{e} & \rightarrow & \text{e} \\
 & | & \text{e+} \\
 & | & \text{e-} \\
 & | & \text{E} \\
 & | & \text{E+} \\
 & | & \text{E-}
\end{array}
$$

Table 4: Grammar for JSON

D Formal definition of JSCore

[*pat*]	:	Zero or more repetitions of *pat* surrounded by [ ] and comma separated (A JSON Array).
[*pat*]$^+$	:	One or more repetitions of *pat* surrounded by [ ] and comma separated (A JSON Array).
{ *pat* }	:	Represents a JSON Object, *pat* is a JSON *members*.
$pat_1 \mid pat_2$	:	Choice.
$\|\|$ *pat* $\|\|$	:	Optional

JSCore grammar:

Module			
module	→	{ ”%module” : *mident* , ”tdefg” : [*tdefg*] , ”vdefg” : [*vdefg*] }	
Type defn.			
tdefg	→	{ ”%data” : *qtycon* , ”tbind” : [*tbind*], ”cdef” : [*cdef*] }	algebraic type
	$\mid$	{ ”%newtype” : *qtycon* , ”qtycon” : *qtycon* , ”tbind” : [*tbind*] , ”ty” : *ty* }	newtype
Constr. defn.			
cdef	→	{ ”qdcon” : *qdcon* , ”tbind” : [*tbind*] , ”aty” : [aty]$^+$ }	
Value defn.			
vdefg	→	{ ”%rec” : [*vdef*]$^+$ }	recursive
	$\mid$	*vdef*	non-recursive
vdef	→	{ ”qvar” : *qvar* , ”ty” : *ty* , ”exp” : *exp* }	
Atomic expr.			
aexp	→	{ ”qvar” : *qvar* }	variable
	$\mid$	{ ”qdcon” : *qdcon* }	data constructor
	$\mid$	{ ”lit” : *lit* }	literal
	$\mid$	{ ”exp” : *exp* }	nested expr.
Expression			
exp	→	*aexp*	atomic expr.
	$\mid$	{ ”aexp” : *aexp* , ”args” : [*arg*]$^+$ }	application
	$\mid$	{ ”lambda” : [*binder*] , ”exp” : *exp* }	abstraction
	$\mid$	{ ”%let” : *vdefg* , ”%in” : *exp* }	local definition
	$\mid$	{ ”%case” : *aty* , ”exp” : *exp* , ”%of” : *vbind*, ”alt” : [*alt*]$^+$ }	case expr.
	$\mid$	{ ”%cast” : *exp* , ”aty” : *aty* }	type coercion
	$\mid$	{ ”%note” : ” { *char* } ” , ”exp” : *exp* }	expression note
	$\mid$	{ ”%external ccal” : ” { *char* } ” , ”aty” : *aty* }	external reference
	$\mid$	{ ”%dynexternal ccal” : *aty* }	external reference (dynamic)
	$\mid$	{ ”%label” : ” { *char* } ” }	external label
Argument			
arg	→	{ ”aty” : *aty* }	type argument
	$\mid$	{ ”aexp” : *aexp* }	value argument
Case alt			
alt	→	{ ”qdcon” : *qdcon* , ”tbind” : [*tbind*] , ”vbind” : [*vbind*] , ”exp” : *exp* }	constructor alternative
	$\mid$	{ ”lit” : *lit* , ”exp” : *exp* }	literal alternative
	$\mid$	{ ”%_” : *exp* }	default alternative
Binder			

binder	$\rightarrow$	{ "tbind" : *tbind* }	type binder
	$\vert$	{ "vbind" : *vbind* }	value binder

Type binder

tbind	$\rightarrow$	{ "tyvar" : *tyvar* }	implicit of kind *
	$\vert$	{ "tyvar" : *tyvar* , "kind" : *kind* }	explicitly kinded

Value binder

vbind	$\rightarrow$	{ "var" : *var* , "ty" *ty* }	

Literal

lit	$\rightarrow$	*jsstring*	string
	$\vert$	*jsnumber*	number

JSON String

jsstring	$\rightarrow$	""
	$\vert$	" *jschars* "
jschars	$\rightarrow$	*jschar*
	$\vert$	*jschar jschars*
jschar	$\rightarrow$	any Unicode character except " or \ or control characters: $\backslash\backslash$ $\backslash/$ $\backslash b$ $\backslash f$ $\backslash n$ $\backslash r$ $\backslash t$ $\backslash u$ four-hex digits

JSON Number

jsnumber	$\rightarrow$	*jsint*
	$\vert$	*jsint jsfrac*
	$\vert$	*jsint jsexp*
	$\vert$	*jsint jsfrac jsexp*
jsint	$\rightarrow$	*jsdigit*
	$\vert$	$jsdigit1 - 9$ *jsdigits*
	$\vert$	- *jsdigit*
	$\vert$	- $jsdigit1 - 9$ *jsdigits*
jsfrac	$\rightarrow$	. *jsdigits*
jsexp	$\rightarrow$	*jse jsdigits*
jsdigits	$\rightarrow$	*jsdigit*
	$\vert$	*jsdigit jsdigits*
jse	$\rightarrow$	e
	$\vert$	e+
	$\vert$	e-
	$\vert$	E
	$\vert$	E+
	$\vert$	E-

Atomic type

aty	$\rightarrow$	{ "tyvar" : *tyvar* }	type variable
	$\vert$	{ "qtycon" : *qtycon* }	type constructor
	$\vert$	{ "ty" : *ty* }	nested type

Basic type

bty	$\rightarrow$	*aty*	atomic type
	$\vert$	{ "bty" : *bty* , "aty" , *aty* }	type application
	$\vert$	{ "%trans" : *aty* , "aty" : *aty* }	transitive coercion
	$\vert$	{ "%sym" : *aty* }	symmetric coercion

	$\mid$	{ "%unsafe" : *aty* , "aty" : *aty* }	unsafe coercion
	$\mid$	{ "%left" : *aty* }	left coercion
	$\mid$	{ "%right" : *aty* }	right coercion
	$\mid$	{ "%inst" : *aty* , "aty" : *aty* }	instantiation coercion

Type

ty	→	*bty*	basic type
	$\mid$	{ "%forall" : [*tbind*]$^+$, "ty" : *ty* }	type abstraction
	$\mid$	{ "bty" *bty* , "ty" : *ty* }	arrow type construction

Atomic kind

akind	→	*	lifted kind
	$\mid$	#	unlifted kind
	$\mid$	?	open kind
	$\mid$	{ "bty" : *bty* , "bty" : *bty* }	equality kind
	$\mid$	{ "kind" : *kind* }	nested kind

Kind

kind	→	{ "akind" : *akind* }	atomic kind
	$\mid$	{ "akind" : *akind* , "kind" : *kind* }	arrow kind

Identifier

mident	→	" *pname* : *uname* "	module
tycon	→	" *uname* "	type constr.
qtycon	→	" *mident* . *tycon* "	qualified type constr.
tyvar	→	" *lname* "	type variable
dcon	→	" *uname* "	data constr.
qdcon	→	" *mident* . *dcon* "	qualified data constr.
var	→	" *lname* "	variable
qvar	→	" ‖ *mident* . ‖ *var* "	optionally qualified variable

Name

lname	→	*lower* { *namechar* }
uname	→	*upper* { *namechar* }
pname	→	{ *namechar* }$^+$
namechar	→	*lower* $\mid$ *upper* $\mid$ *digit*
lower	→	a $\mid$ b $\mid$ ... $\mid$ z $\mid$ _
upper	→	A $\mid$ B $\mid$ ... $\mid$ Z $\mid$
digit	→	0 $\mid$ 1 $\mid$ ... $\mid$ 9

Table 6: Grammar for JSCore

E Test programs

E.1 helloworld

```
main = putStrLn "Hello, world!"
```

E.2 helloworld2

```
main = putStrLn ("Hello, " ++ "world!")
```

E.3 factorial

```
module Main where

main = do
    let n = 10
    putStrLn $ "Factorial 10 is: " ++ (show $ fac n)

fac :: Int -> Int
fac 0 = 1
fac n = n*fac (n-1)
```

E.4 fibonacci

```
module Main where

main = do
    let n = 10
    putStrLn $ "The 10. fibonacci number is: " ++ (show $ fib n)

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

References

[1] D. Ancona, M. Ancona, A. Cuni, and N.D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64. ACM, 2007.

[2] C.F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. 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*, page 9. ACM, 2011.

[3] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M.M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, et al. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[4] S.L.P. Jones, C. Hall, K. Hammond, J. Cordy, H. Kevin, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. 1992.

[5] S.P. Jones and A. Santos. Compilation by transformation in the Glasgow Haskell Compiler. *Functional Programming, Glasgow*, pages 184–204, 1994.

[6] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 944–953. ACM, 2006.

[7] A. Tolmach and T. Chevalier. the GHC Team. An external representation for the GHC Core language, 2010.