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

Knut Halvor Skrede

December 10, 2011

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

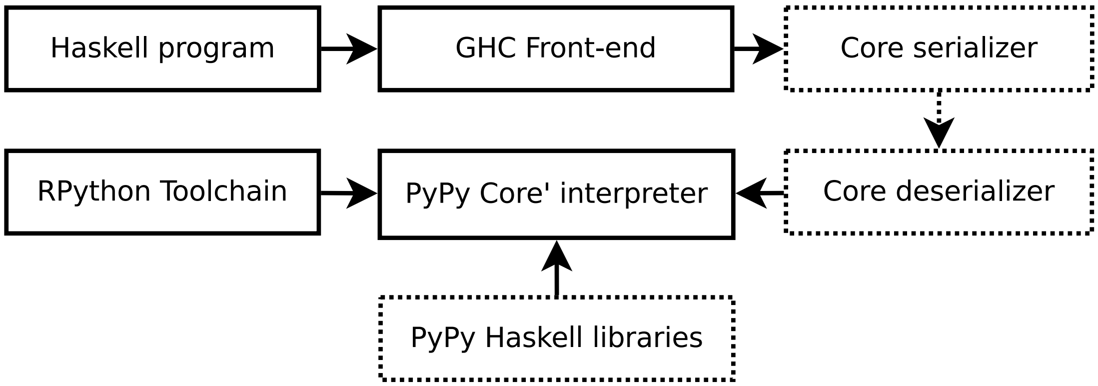

Figure 1: Project overview. Full border: implemented, Dotted border: unimplemented.

1 Introduction

1.1 Project description

The project is to implement a serializer from Haskell to an intermediate format using GHC, and a deserializer from the intermediate format to an interpreter usin 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 statically typed languages since they can be so heavily optimized at compile time. However, a JIT compiler has a lot more information to work with.

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. 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".

GHC (Glasgow Haskell Compiler) is the most advanced compiler for the Haskell programming language. The GHC team has defined an external format of Haskell 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. [6]

1.3 System description

The system will use the GHC Haskell library 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.

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.

1.4 Structure of the paper

Follwing this introduction is a section (Section 3: Background) discussing the background of the project. The background section contains some information about the tools beeing used, and why. Section 3 (External-core) contains a description of the external language representation of the intermediate language used by GHC. Section 4 (JSON representation of Core) follows with the description of the intermediate format used in this project, which is a combination of JSON and external-core. Section 5 (Implementation) discusses the implementation of the system, the issues faced during development and some of the remaining issues. In section 6 (Examples), example programs is passed through the pipeline with intermediate representations presented at each step. Section 7 (Future Work) discusses work to be done in the future.

2 Background

2.1 PyPy

2.1.1 A quick overview

The PyPy project is basically two things:

1. The RPython toolchain, written in Python; a set of compiler tools for programs written in RPython.
2. 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.

2.1.2 RPython

RPython is a restricted proper subset of Python, this enables easy analyzis as well as efficient compilation. This also means that RPython code can be run and debugged by Python interpreters, like CPython.

2.1.3 Translation toolchain

2.1.4 Just-in-time compilation

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

An introduction to virtual machine (VM) construction with python [5].

2.1.5 Haskell-Python

Haskell-Python is a interpreter for a subset of the Haskell language, called Core'. We call it Core' here because it does not directly correspond to the Core language used by GHC (the 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 start extending Haskell-Python to use GHC as a frontend for compilation of Haskell programs.

2.2 GHC

Haskell is a *strongly-typed non-strict purely-functional* programming language, it will not be described in detail here, since Haskell is not the language we focus on. See [1] for an introduction to Haskell.

Core is an intermediate language used by the Glasgow Haskell Compiler [3], 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. [4]

3 External-core

The Core language is an intermediate language used by GHC. It is the internal program representation in the compilers simplification phase. External-core is an external representation of Core generated by using a compiler flag. 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 remaing task is then to interpret the external-core representation. [6]

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. [6]

The initial starting point of this project was to use the GHC API, the reason 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 turened out that this too was a complicated task. As the internal core datatypes of GHC 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.

3.1 External-core language definition

The following semantics is used to define the Core grammar, as seen in [6]:

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

	$\mid$	%dynexternal ccal *aty*	external reference (dynamic)
	$\mid$	%label " { *char* } "	external label

Argument

arg	$\rightarrow$	@ *aty*	type argument
	$\mid$	*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	→	*akind*	atomic kind
	|	*akind* -> *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* | *upper* | *digit*
lower	→	a|b|...|z|_
upper	→	A|B|...|Z|
digit	→	0|1|...|9

3.2 Evaluation of a program

A program is evaluated by reducing the expression "main:ZCMain.main" 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* [6]

4 JSON representation of Core

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.

4.1 Formal definition of JSON

Object

object	→	{ }
	|	{ *members* }
members	→	*pair*
	|	*pair* , *members*
pair	→	*string* : *value*

Array

array	→	[ ]
	|	[*elements*]
elements	→	*value*
	|	*value* , *elements*

Value

value	→	*string*
	|	*number*
	|	*object*
	|	*array*

	$\mid$	true
	$\mid$	false
	$\mid$	null

String

string	$\rightarrow$	""
	$\mid$	" *chars* "
chars	$\rightarrow$	*char*
	$\mid$	*char chars*
char	$\rightarrow$	any Unicode character except " or \ or control characters:
		\\
		\/
		$\backslash b$
		$\backslash f$
		$\backslash n$
		$\backslash r$
		$\backslash t$
		$\backslash u$ four-hex digits

Number

number	$\rightarrow$	*int*
	$\mid$	*int frac*
	$\mid$	*int exp*
	$\mid$	*int frac exp*
int	$\rightarrow$	*digit*
	$\mid$	$digit1 - 9\ digits$
	$\mid$	- *digit*
	$\mid$	- $digit1 - 9\ digits$
frac	$\rightarrow$	. *digits*
exp	$\rightarrow$	*e digits*
digits	$\rightarrow$	*digit*
	$\mid$	*digit digits*
e	$\rightarrow$	e
	$\mid$	e+
	$\mid$	e-
	$\mid$	E
	$\mid$	E+
	$\mid$	E-

Table 3: Grammar for JSON

4.2 JSON representation of Core

In order to work with JSON and external-core, a format was define 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.

The following definitions was used to describe the grammar:

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

JSON Core grammar:

Module

module	$\rightarrow$	{ "%module" : *mident* , "tdefg" : [*tdefg*] , "vdefg" : [*vdefg*] }	

Type defn.

tdefg	$\rightarrow$	{ "%data" : *qtycon* , "tbind" : [*tbind*], "cdef" : [*cdef*] }	algebraic type
	$\mid$	{ "%newtype" : *qtycon* , "qtycon" : *qtycon* , "tbind" : [*tbind*] , "ty" : *ty* }	newtype

Constr. defn.

cdef	$\rightarrow$	{ ”qdcon” : *qdcon* , ”tbind” : [*tbind*] , ”aty” : [aty]$^+$ }	

Value defn.

vdefg	$\rightarrow$	{ ”%rec” : [*vdef*]$^+$ }	recursive
	$\mid$	*vdef*	non-recursive
vdef	$\rightarrow$	{ ”qvar” : *qvar* , ”ty” : *ty* , ”exp” : *exp* }	

Atomic expr.

aexp	$\rightarrow$	*qvar*	variable
	$\mid$	*qdcon*	data constructor
	$\mid$	*lit*	literal
	$\mid$	{ ”exp” : *exp* }	nested expr.

Expression

exp	$\rightarrow$	*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	$\rightarrow$	{ ”aty” : *aty* }	type argument
	$\mid$	{ ”aexp” : *aexp* }	value argument

Case alt

alt	$\rightarrow$	{ ”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
	$\mid$	{ ”vbind” : *vbind* }	value binder

Type binder

tbind	$\rightarrow$	{ ”tyvar” : *tyvar* }	implicit of kind *
	$\mid$	{ ”tyvar” : *tyvar* , ”kind” : *kind* }	explicitly kinded

Value binder

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

Literal

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

JSON String

jsstring	$\rightarrow$	””	
	$\mid$	” *jschars* ”	
jschars	$\rightarrow$	*jschar*	
	$\mid$	*jschar jschars*	
jschar	$\rightarrow$	any Unicode character except ” or \ or control characters:	
		\\	
		\/	

\b
\f
\n
\r
\t
\u four-hex digits

JSON Number

jsnumber	→	*jsint*
	|	*jsint jsfrac*
	|	*jsint jsexp*
	|	*jsint jsfrac jsexp*
jsint	→	*jsdigit*
	|	*jsdigit*1 − 9 *jsdigits*
	|	- *jsdigit*
	|	- *jsdigit*1 − 9 *jsdigits*
jsfrac	→	. *jsdigits*
jsexp	→	*jse jsdigits*
jsdigits	→	*jsdigit*
	|	*jsdigit jsdigits*
jse	→	e
	|	e+
	|	e-
	|	E
	|	E+
	|	E-

Atomic type

aty	→	{ "tyvar" : *tyvar* }	type variable
	|	{ "qtycon" : *qtycon* }	type constructor
	|	{ "ty" : *ty* }	nested type

Basic type

bty	→	*aty*	atomic type
	|	{ "bty" : *bty* , "aty" , *aty* }	type application
	|	{ "%trans" : *aty* , "aty" : *aty* }	transitive coercion
	|	{ "%sym" : *aty* }	symmetric coercion
	|	{ "%unsafe" : *aty* , "aty" : *aty* }	unsafe coercion
	|	{ "%left" : *aty* }	left coercion
	|	{ "%right" : *aty* }	right coercion
	|	{ "%inst" : *aty* , "aty" : *aty* }	instantiation coercion

Type

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

Atomic kind

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

Kind

kind	→	{ "akind" : *akind* }	atomic kind
	|	{ "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 5: Grammar for JSCore

5 Implementation

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 allready implemented PyPy Core' interpreter, Haskell-Python. Additional functionality had to be built on top of this, mostly Haskell library functions.

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
- Python 2.7: Used to test the Core' interpreter without it having to be correct RPython.

5.2 Organization

The implementation is organized as represented by figure 2. The main folder (interpreter) contains the main program, and a program for generating dot files, ”makegraph.py” (used to create graphs of parsed JSCore files using graphviz). The ”haskell” folder contains the PyPy 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 packages implements some simple functionality to used by the test-programs. Among others, a very simple IO function for printing text to the terminal (putStrLn). These packages are loaded and references to the functions they contain are used during the creation of the AST. See figure 3 for a brief description of the pipeline.

The ”core” folder contains Haskell program for generating JSCore files from external-core files, the JSCore parser, and a simple datastructure representing Haskell modules.

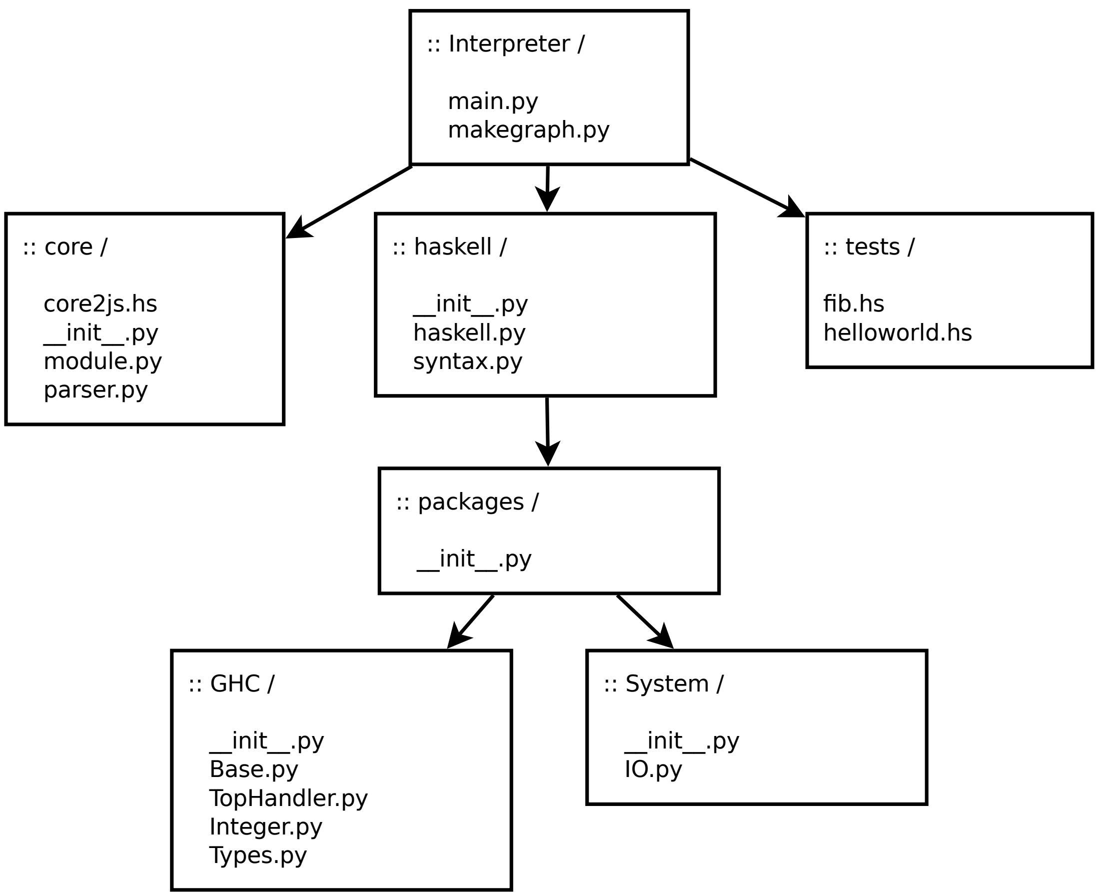

Figure 2: Code tree: The top of the boxes is the folder name, and the rest is the source files. Arrows represent subfolders.

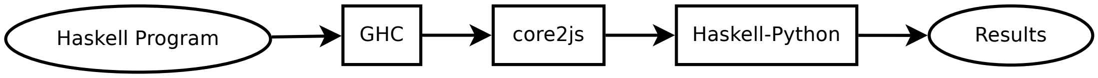

Figure 3: Pipeline implementation with core2js using extcore

5.3 Serializer

The serializer consists of two parts; GHC generating external-core, and a Haskell program to generate JSCore.

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 [6]. 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 5), this is then pretty-printed and dumped to a file.

5.4 Deserializer

The deserializer implements a JSON parser. The resulting datastructure is then traversed, 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 3. The resulting datastructure is then traversed, by checking the contents of the JSON constructs with the actual external-core format, the Core AST is built. External functionality is imported as it is encountered.

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

5.5 Haskell libraries

To make some simple test-cases work, some basic Haskell functionality had to be implemented. Some of this functionality was implemented allready 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 5 (representing a "hello world" program in JSCore), the atomic expression "base:SystemziIO.putStrLn" 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 located in "haskell/packages/System/IO.py". See figure 2.

5.6 Evaluation

In order to evaluate the Haskell programs correctly, the expression "main:ZCMain.main" would have to be reduced to *WHNF*. However, this would require alot of the functionality used by GHC to be implemented. Specifically, "GHC.TopHandler.runMainIO()". In order to implement this function alot 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. Implementing this is a goal for further development, but currently a simple hack is to only evaluate the expression "main:Main.main". This way, simple programs can be tested by implementing the necessary functionality at a high level, such as the "putStrLn" function is implemented as a simple "print" function in python.

5.7 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

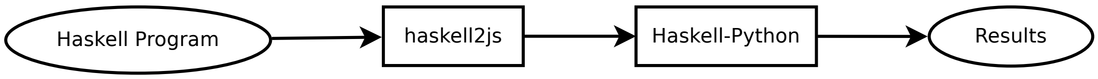

Figure 4: Pipeline with implementation of haskell2js using the GHC API

figure 4 for a description of the initially intended pipeline). It was then discovered that alot of the necessary code was allready written in GHC (the code that generates the external-core files), however, this code was not exported in any way. It was also deeply embeded in the GHC code. After several attempts at creating the JSCore format, and some emails back and forth with the GHC team, these methods where abandoned.

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 very well supported, as the external-core format seems to be changing. This also turned out to be the case. However, GHC also changes rapidly, and an implementation using the GHC API may not work for very long either. A version linking to the GHC executable would most likely be the worst choice.

A large amount of time was spent trying to generate this intermediate format.

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 GHC versions later than 6.10.

6 Examples

6.1 Example 1: hello world

This example program 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:

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

6.1.1 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;
```

... TODO: Explain this representation in more detail.

6.1.2 Converted to JSCore

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

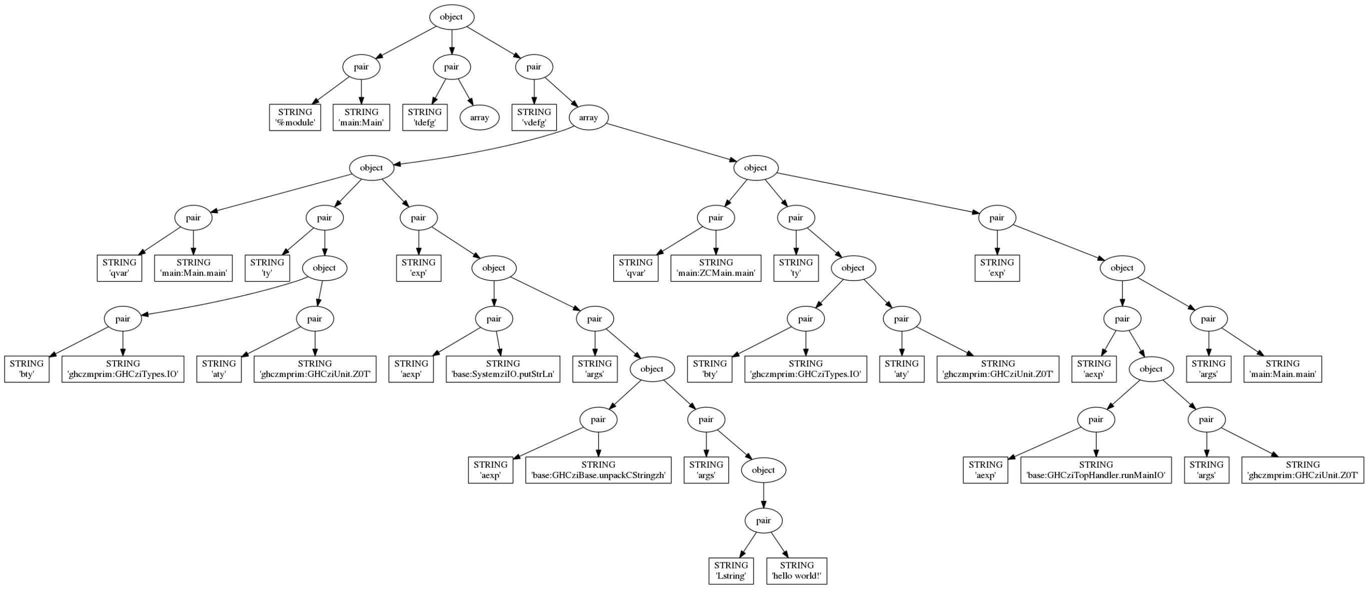

Figure 5: Example program translated to JSON

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

6.1.3 JSCore graph

Using the parsing libraries of pypy we can generate a nice graph from the result, see figure 5. By simply traversing this datastructure we can generate the AST for the Core interpreter (Haskell-Python).

6.1.4 Result

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

6.2 Example 2: fibonacci

The following program is a simple fibonacci program.

```
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)
```

6.2.1 Converted to Core

As we can see, the simple hello world program becomes more complex when translated to Core by ghc.

```
%module main:Main
  %rec
  {fibrbu :: integerzmgmp:GHCziIntegerziType.Integer ->
             integerzmgmp:GHCziIntegerziType.Integer =
     \ (dsdno::integerzmgmp:GHCziIntegerziType.Integer) ->
         %case integerzmgmp:GHCziIntegerziType.Integer (base:GHCziClasses.zeze
                                                        @ integerzmgmp:
                                                            GHCziIntegerziType.Integer
                                                        base:GHCziClasses.
                                                            zdfEqInteger dsdno
                                                        (integerzmgmp:GHCziInteger.
                                                            smallInteger
                                                          (0::ghczmprim:GHCziPrim.
                                                              Intzh)))
         %of (wildB1::ghczmprim:GHCziBool.Bool)
           {ghczmprim:GHCziBool.False ->
              %case integerzmgmp:GHCziIntegerziType.Integer (base:GHCziClasses.zeze
                                                             @ integerzmgmp:
                                                                 GHCziIntegerziType.
                                                                 Integer
                                                             base:GHCziClasses.
                                                                 zdfEqInteger dsdno
                                                             (integerzmgmp:
                                                                 GHCziInteger.
                                                                 smallInteger
                                                               (1::ghczmprim:GHCziPrim
                                                                  .Intzh)))
              %of (wild1X8::ghczmprim:GHCziBool.Bool)
                {ghczmprim:GHCziBool.False ->
                   base:GHCziNum.zp @ integerzmgmp:GHCziIntegerziType.Integer
                   base:GHCziNum.zdfNumInteger
                   (fibrbu
                    (base:GHCziNum.zm @ integerzmgmp:GHCziIntegerziType.Integer
                     base:GHCziNum.zdfNumInteger dsdno
                     (integerzmgmp:GHCziInteger.smallInteger
                      (1::ghczmprim:GHCziPrim.Intzh))))
                   (fibrbu
                    (base:GHCziNum.zm @ integerzmgmp:GHCziIntegerziType.Integer
                     base:GHCziNum.zdfNumInteger dsdno
                     (integerzmgmp:GHCziInteger.smallInteger
                      (2::ghczmprim:GHCziPrim.Intzh))));
                 ghczmprim:GHCziBool.True ->
                   integerzmgmp:GHCziInteger.smallInteger
                   (1::ghczmprim:GHCziPrim.Intzh)};
            ghczmprim:GHCziBool.True ->
              integerzmgmp:GHCziInteger.smallInteger
              (0::ghczmprim:GHCziPrim.Intzh)}};
  main:Main.main :: (ghczmprim:GHCziTypes.IO
                     ghczmprim:GHCziUnit.Z0T) =
```

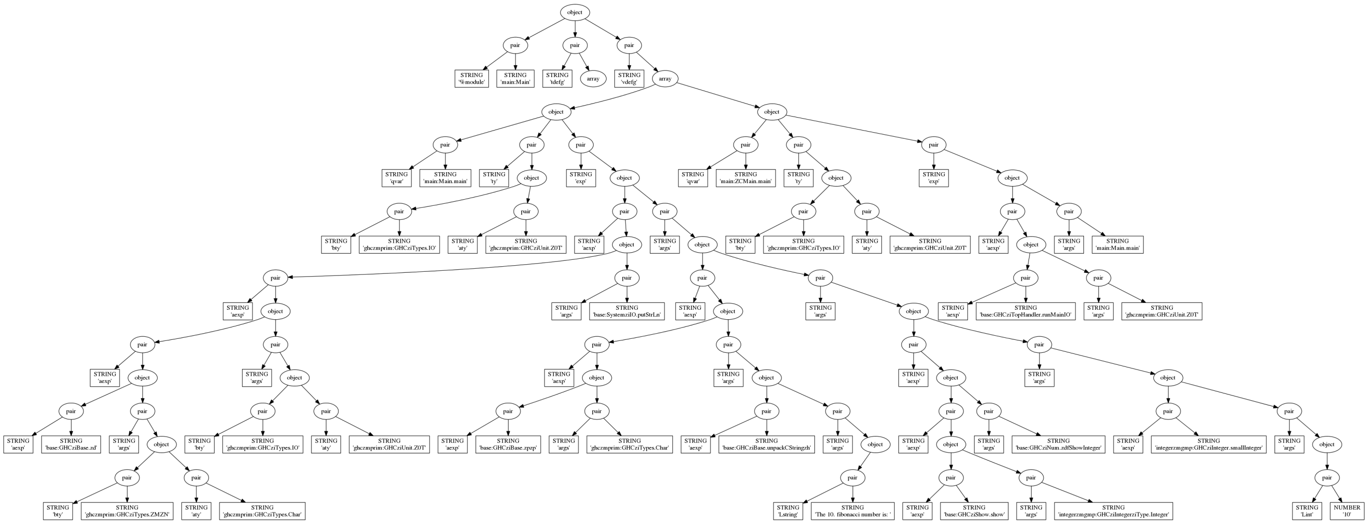

Figure 6: Example program 2 translated to JSON

```
    base:GHCziBase.zd
    @ ((ghczmprim:GHCziTypes.ZMZN ghczmprim:GHCziTypes.Char))
    @ ((ghczmprim:GHCziTypes.IO ghczmprim:GHCziUnit.Z0T))
    base:SystemziIO.putStrLn
    (base:GHCziBase.zpzp @ ghczmprim:GHCziTypes.Char
      (base:GHCziBase.unpackCStringzh
        ("The 10. fibonacci number is: "::ghczmprim:GHCziPrim.Addrzh))
      (base:GHCziBase.zd @ integerzmgmp:GHCziIntegerziType.Integer
       @ ((ghczmprim:GHCziTypes.ZMZN ghczmprim:GHCziTypes.Char))
       (base:GHCziShow.show @ integerzmgmp:GHCziIntegerziType.Integer
         base:GHCziNum.zdfShowInteger)
       (fibrbu
         (integerzmgmp:GHCziInteger.smallInteger
           (10::ghczmprim:GHCziPrim.Intzh))))));
main:ZCMain.main :: (ghczmprim:GHCziTypes.IO
                         ghczmprim:GHCziUnit.Z0T) =
    base:GHCziTopHandler.runMainIO @ ghczmprim:GHCziUnit.Z0T
    main:Main.main;
```

6.2.2 Converted to JSCore

And translated to JSCore by our serializer:

```
"\"fib.hcr\" (line 2, column 4):\u000aunexpected \"r\"\u000aexpecting \"data\" or \"
    newtype\""
```

6.2.3 JSCore graph

Using the parsing libraries of pypy we can generate a nice graph from the result, see figure 6

By simply traversing this datastructure we can generate the AST for the Core interpreter.

6.2.4 Result

7 Future work

7.1 Getting programs into Haskell-Python from GHC

This project focused mainly on this task, which was thought to be simple. However, due to a combination of reasons, this turned out not to be the case. Mainly, inexperience; with Haskell, the GHC API, and functional languages in general. However, a lot of experience was gained during this project, and future development will benefit from this.

The work involved in this will be revisiting the different possibilities to achieve our goal. These options are:

- Write an external-core parser directly in RPython, using files generated by GHC.
- Include development of extcore as a part of the project. This most likely a bad idea, as this seems to be no easier than the alternatives.
- Use the GHC API to generate JSCore, this is also nontrivial.
- Create functionality to be linked into the GHC executable, in order to generate the representation of JSCore. Also nontrivial.
- Simply use the version of GHC matching the version of extcore. This will not be a good idea for further development of the project, but may be a simple solution to get a prototype working quickly.
- Implement a Haskell pipeline in RPython, including parsing, typechecking and desugaring. This will be a very large task.

7.2 Rewrite the deserializer to proper RPython

The deserializer is currently not written in proper RPython. Converting the code to RPython will alow 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.

7.3 Implement base 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.

References

[1] 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.

[2] S.L.P. Jones. Implementing lazy functional languages on stock hardware: the spineless tagless g-machine. *Journal of functional programming*, 2(2):127–202, 1992.

[3] 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.

[4] S.P. Jones and A. Santos. Compilation by transformation in the glasgow haskell compiler. *Functional Programming, Glasgow*, pages 184–204, 1994.

[5] 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.

[6] A. Tolmach and T. Chevalier. the ghc team. an external representation for the ghc core language, 2010.