# MuPy: A First Client for the Mu Micro Virtual Machine

**(John) Jiabin Zhang**

A subthesis submitted in partial fulfillment of the degree of
Bachelor of Advanced Computing (Research & Development) at
The Department of Computer Science
Australian National University

October 2015

# Acknowledgements

Firstly I'd like to thank my supervisor, Steve Blackburn, for giving me this opportunity to work on this exciting project. Having a lot of responsibilities on his shoulders, he has been quite busy. Yet he still sacrificed a lot of his time to give me directions for my research, encourage me by reminding me the value of my work, and sacrifice extra time for meetings and giving last minute feedback for my thesis.

I also thank the Mu research team and the fellow research students under the supervision of Steve. I thank Kunshan, for taking an interest in me and my work. You have kindly given me a lot of help and support whenever I ask of you since the very beginning of the project. I thank Yi, Michael and Tony also for kindly supporting me throughout the way. Luke has given me a lot of encouragements when I had been worried or stressed, of which I am very thankful.

I have also received a great deal of support from my friends from the FOCUS (Fellowship of Christian University Students), all of who have been dear to me as brothers and sisters. I especially thank Micaiah, Elysia and Hendra for their friendship, their faith and love. You all have been with me through all sorts of difficulties and tough times. Words cannot express my gratefulness to all of you.

Lastly and above all, I thank my Lord Jesus Christ. Who has sent all these people into my lives; who has been with me through joy and tears. At all time he as been merciful, faithful and graceful to me in provision. To Him be all the glory.

# Abstract

Much of the complexity in programming language implementation is made up of three fundamental concerns, namely concurrency, memory and hardware platform. Many of the programming languages today, especially managed languages, have broken features due to the inappropriate treatment of these three challenges. Micro Virtual Machines, a thin abstraction layer over these three concerns, was proposed to free up the language implementers of making compromises in language implementation to focus on the higher level features.

With the progress in the research of Mu, a concrete instance of Micro Virtual Machine, its hypothesis and design can now be tested. To test Mu's design and its ability to support non-trivial languages, I have embarked on the mission to develop an RPython compiler back-end targeting Mu. RPython, as a compilation framework, has been used to implement interpreters for managed languages, and has displayed promising results. Thus the development of an RPython language client is a strategically critical step in providing the power of Mu to many managed languages.

This thesis explores the unique challenges of bringing together Mu as a platform of great potential and RPython as a language client of high interest. Being a pioneering work in implementing language clients for Mu, this thesis also exposes some deficiencies in the design of Mu through some of the encountered issues, enabling Mu to extend its design with additional features to offer better support for client languages. The research of this thesis has also had milestone achievements. Having translated the essential features of RPython, the Mu back-end enables RPython to translate small scale programs and GC benchmark to Mu IR, thus testifying to the ability of Mu to support non-trivial languages.

In this thesis I will describe the various stages of the Mu back-end translation process, exploring the problems and issues in developing a language client for Mu with its distinct type system and design. I will also discuss the possible solutions to these issues, and how these issues have impacted on the design of Mu. I believe this work is strategically critical to the Micro Virtual Machine research team, the PyPy/RPython development team, and many other language implementers who may be benefitted from implementing their languages on top of the Mu Micro Virtual Machine.

# Contents

Chapter 1

# Introduction

The core of my work is in developing a language client for the Mu Micro Virtual Machine, in order to test its claims and hypothesis. This chapter provides an overview to the problem and my contributions.

## 1.1 Problem statement

### 1.1.1 Mu and Micro Virtual Machine

Generally, programming language implementations can be done in two ways, either being monolithic (building everything from the ground up), or building on an existing Virtual Machine (VM). Either way comes with benefits and difficulties. Building on top of a VM gives a higher level of abstraction over the machine architecture, which provides good support for compiler backend, memory and concurrency. However the existing VMs are usually heavy-weight, featuring high level optimisations and extensive libraries. They also tend to be tailored to the abstraction of the language they support, leaving them unsuitable for new programming languages [Castanos et al. 2012; Wang et al. 2015].

Micro Virtual Machines ($\mu$VMs), a concept proposed by Wang, Lin, Blackburn, Norrish and Hosking [Wang et al. 2015], is aimed to provide language implementers a robust and light-weight platform to build their language. There are three main concerns, as identified by Wang *et al.*, in monolithic language implementation: hardware (compiler backend), memory (garbage collection), and concurrency (scheduler and memory model). $\mu$VM aims to abstract over these three concerns, providing a platform for language implementers to freely focus on higher level language design. Different from many existing VMs like the JVM, CLR and LLVM, $\mu$VM aims to be minimal and light-weight, designed to support the development of new languages, and also natively support Garbage Collection (GC) and concurrency. It also supports cross-language reuse of demanding implementation details, enabling it to be a suitable platform for many different languages. Being light-weight also allows $\mu$VM to have the potential to be formally verified, making it suitable for security-critical applications as well.

### 1.1.2 The need for testing

The concept of Micro Virtual Machines has great potential in benefitting the booming programming language ecosystem today. It frees the designers of future languages from making compromises in design due to the challenges in language implementation.

However, these claims, along with the concept of $\mu$VM, must be tested and verified. Wang *et al.* has embarked on a mission to develop a concrete implementation of a $\mu$VM, called Mu. Major progress has been made on the specification and reference implementation of Mu [Micro VM ]. With this progress, the design of Mu in supporting a non-trivial modern managed language can now be tested.

## 1.2 Contributions

The claim and design of Mu can only be tested through building a concrete implementation of a modern managed language on top of it. No language client has been built on Mu yet. Consequently, no one has yet confirmed the benefits and difficulties of developing a language client for Mu. This thesis thus serves as pioneering research in this area.

Through building an RPython client for Mu, I have tested the design of Mu to support managed languages. While Mu offers largely compatible program structure, type system and features such as exception catching which all simplify the translation process, Mu also has deficiencies in its design, which raise issues in supporting parts of the RPython language. These findings have motivated Wang *et al.* to reconsider the design of Mu and include additional features to the most recent specification. These additional features provide solutions to the encountered issues, and strengthens the ability of Mu to support modern managed languages.

### 1.2.1 RPython and MuPy language client

#### 1.2.1.1 RPython as the language of choice

RPython is a restricted subset of Python that is compiled into C [**?**]. The restrictions are primarily on the dynamic typing feature of Python. Being statically typed, RPython programs can be analysed and optimised at compile time. More than simply being a thin layer over C, RPython has built-in garbage collectors and several high-level datatypes, making the development of language implementations much easier. In addition, the highlight of RPython as an interpreter framework is the ability to automatically generate a high performance meta-tracing Just-In-Time (JIT) compiler for the language interpreter. RPython thus provides an efficient way of implementing high performance managed language implementations. More details about the translation process of RPython will be discussed in Chapter 3.

Being a powerful compilation framework to generate a high performance interpreter, alternative implementations of many popular languages have been made using RPython. These include Python (PyPy), Prolog (Pyrolog), Erlang (Pyrlang), Racket

(Pycket), JavaScript (PyJS), Haskell (PyHaskell) and so on. Thus having RPython as a language client for Mu is a strategically critical step in porting many modern managed languages on top of it.

#### 1.2.1.2 The MuPy project

Seeing its strategic value, I have been motivated to develop a Mu back-end for RPython in addition to its existing C back-end. I name this project MuPy, capturing both the platform and language.

The RPython framework turns an RPython program into control flow graphs, infers the types of variables and functions, performs various optimisations and transformations, and then translates them into C code. The goal of this project is to 'redirect' the translation process so that instead of generating C code, the Mu back-end will instead generate Mu Intermediate Representation (IR) code.

I break down this translation process into three stages.

**Graph transformation** alters the structure of the RPython control flow graphs to match the structure of function definitions in Mu. This involves adding branching and return instructions, inserting `PHI` instructions at join points, creating transitional blocks for links that have the same source and destination blocks, and transforming the exceptions in an alternate way that utilises the exception handling support from Mu.

**Type mapping (MuTyping)** converts the types in the flow graph to the types in the Mu type system, along with mapping instructions. It also identifies the initialised global heap objects that are to be stored in Mu global cells.

**Mu IR code generation** turns the flow graphs into the Mu IR program bundle. An initialisation routine for the objects to be stored in global cells is also generated.

This thesis focuses on graph transformation and type mapping, since Mu IR code generation is relatively straightforward.

To load the translated program bundle, I developed a minimal language client. The client contains a launcher that loads the bundle program, constructs an RPython list of command-line argument strings, and executes the bundle code. I also implemented the print output in this language client using the trap mechanism.

The Mu back-end for RPython and the language client has had reasonable success in translating small scale programs. It was able to translate and execute successfully the GC benchmark [Ellis et al. ].

### 1.2.2 Impact on the research of Mu

This project of building the RPython language client has also had many positive contributions to the research and implementation of Mu, as I have been working closely with the Mu research group. Some of the additional features in the most recent version of Mu's specification are motivated by the issues encountered in the MuPy project.

One of the problems encountered is the initialisation of heap objects. RPython represents these objects as constants in the flow graph and does not provide explicit routines to initialise them. The intuitive approach of creating an initialisation routine as the entry point for the program bundle, as I have currently taken, has drawbacks such as significantly increasing the size of the bundle code. Another approach of specifying the layout of these objects using a language has motivated the Mu research team to include a Heap Allocation Initialisation Language (HAIL) in the most recent specification [Mu Spec ].

RPython also contains some direct memory access operations and compiler intrinsics. The compromised treatment for these elements due to the lack of support has caused significant performance cost. This has led to discussions on the design of Mu and the proposal of the native interface extension. This feature is eventually included in the most recent specification, enabling the client languages to perform lightweight C library calls.

The MuPy project thus has motivated the inclusion of these additional features capable of bringing about new possibilities in implementing client languages.

## 1.3 Thesis outline

Chapter 2 presents background on language implementation, Micro Virtual Machines and RPython. Chapter 3 provides details about the RPython translation that are necessary for the implementation detail of the RPython language back-end. Chapter 4 describes the transformation of structural elements in the control flow graphs, along with the discussion on exception transformation strategies. Chapter 5 discusses, with detail, the mapping of type system and operations, the global constants and the problem of initialisation, and the problem of raw memory access operations. Chapter 6 presents the MuPy language client, containing the loader and treatment of print output. The finale Chapter 7 concludes my thesis, identifying the future work and direction of research, and describing how my contributions testifies to the ability of Mu supporting modern managed languages.

Chapter 2

# Background and Related Work

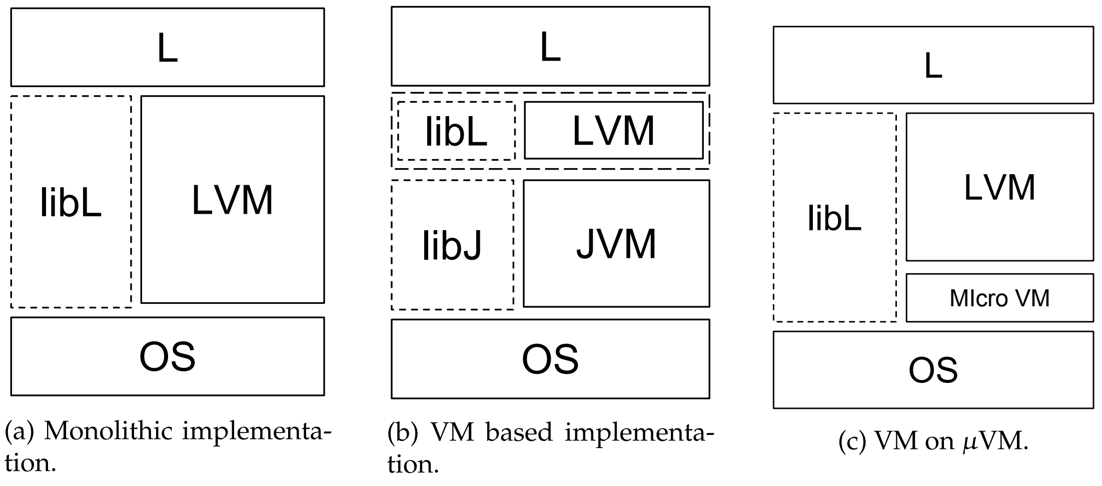
Virtual machines are widely used for language implementations. They provide abstractions and support over many desirable features such as GC, JIT, threading, portability etc. that simplify the development for new languages. This chapter provides a brief introduction to the background of the use of virtual machines in programming language implementation, the motivation and design of the Mu Micro Virtual Machine, and the PyPy project with the underlying RPython compilation framework.

Section 2.1 briefly presents the monolithic and virtual machine based programming language implementation strategies and some of the issues involved. Section 2.2 introduces the Mu Micro Virtual Machine, with its motivation, design features and potential. Section 2.3 gives a general overview of the RPython compilation framework, the languages implemented using RPython, and existing back-ends. Finally, 2.4 gives a summary to the chapter, emphasising value of having a Mu back-end for RPython.

## 2.1 Programming Language Implementation Strategies

Programming languages play an essential role in the computing industry. Nowadays, programming languages have become highly abstract and rich in features. While offering the programmers higher grounds to express program logic and algorithms, modern languages have also increased the complexity of implementation. The compilation process of Haskell, for example, involves a complex pipeline of procedures to lower the abstractions [Marlow and Peyton-Jones ].

There are generally two approaches to language implementation: monolithic (Figure 2.1a) and building on existing platforms (Figure 2.1b) [Wang et al. 2015]. In the monolithic approach, the language designer needs to implement every detail of the language (*e.g*. CPython, Haskell, Javascript etc.). All of the abstractions and mechanisms (*e.g*. GC, concurrency, back-end architecture etc.) need to be brought down to a lower level language. Naive solutions to these details can be easy, yet prune to limitations, having low performance, and broken features due to inappropriate treatments. Naive reference counting garbage collection strategy, for example, though easy to implement, has well-known performance limitations and the inability to collect cycles; and the inappropriate treatment of concurrency in the early stages of CPython has led to the infamous Global Interpreter Lock (GIL) to be baked into the language [Behrens



(a) Monolithic implementation.

(b) VM based implementation.

(c) VM on $\mu$VM.

Figure 2.1: Language implementation approaches.

2008].

Another approach is building the language implementation on existing platforms, usually virtual machines (VMs). Examples include Jython, Scala and C#. VMs (*e.g.* JVM, CLR etc.) raise the abstraction of the target machine, providing support for many low level details. Many VMs provide a highly optimised garbage collector, support for concurrency, machine independent intermediate code, and extensive libraries. Jython for example, by mapping Python threads to Java threads, has got rid of GIL [Juneau et al. 2010]. Thus these VMs can provide great assistance in simplifying the compilation process, and prevent compromised implementation decisions being baked into the design. This approach also has draw-backs, such as the fact that many of these platforms are usually large, containing far more than the need of a client language. They are also tailored towards the language they support, and can be a poor match to new languages [Castanos et al. 2012; Wang et al. 2015].

There are also compiler infrastructures such LLVM that provide reusable components for building compilers [Lattner and Adve 2004]. LLVM provides support for generating an intermediate representation, a great deal of optimisations, and compilation down to the machine level. However it is weak on the support for GC and JIT compilation due to its focus on C, making it not a highly desirable tool for modern dynamic languages that relies heavily on run-time JIT compilation [Wang et al. 2015].

In recent years RPython has emerged as an attractive compilation framework for developing language interpreters [RPython Doc ]. Containing implementations of GC and a high performance meta-tracing JIT, RPython covers many lower level details, and allows high performance language interpreters to be developed using a subset of Python language with high level abstractions. However RPython's support for threading is weak and almost non-existant. Language implementations written in RPython need to implement their own concurrency strategies.

## 2.2 Mu and Micro Virtual Machine

### 2.2.1 Motivation & Proposal

Motivated by the complexity of language implementation, and the lack of good supporting substrates, Wang *et al.* proposed the concept of Micro Virtual Machines ($\mu$VMs).

$\mu$VMs attack the three fundamental concerns in language implementation, memory (GC), concurrency, and back-end architecture (portability and JIT), by providing a thin abstraction layer over them (Figure 2.1c). A $\mu$VM offers language implementers a unifying substrate, on which other virtual machines and language clients can be built; and gives them both the access to state of the art foundations and maximum liberty to implement language-specific semantics.

### 2.2.2 Mu

Mu is a concrete instance of a $\mu$VM. It provides well defined a intermediate representation and a client interface that aim to support a diverse range of language clients. Being explicitly minimal and light-weight, the design of Mu also aims to be formally verifiable and a trusted substrate for other VMs..

Mu adopts a client-server model, allowing the flexibility of multiple language clients running on multiple and possibly remote Mu servers. A language client can communicate to a Mu server through the client interface to manipulate the state of Mu and handle asynchronous events from Mu. Mu accepts the Mu intermediate representation (Mu IR) code. Thus a language client is responsible for compiling the program in source language to Mu IR in the unit of code bundles, and deliver them to a Mu server to be executed.

The type system of Mu is simple, providing only universal and fundamental machine abstractions. Apart from varying bit-widths integer types, two floating point types and various memory reference types, it also contains fixed width struct and array types, variable sized hybrid types, and vector types fro SIMD instructions. The operations in Mu take into account the interaction between GC and concurrency, and provides support for many desirable features such as exception catching, light weight context switching, swap-stack operation and traps. In the recent specification, Mu also includes a Mu Native Interface (MuNI), that allows the client language to perform arbitrary yet unsafe C calls (the clients are trusted). Thus Mu allows a wide range of possibilities for client language implementation.

Mu is still undergoing active research. More detail about the design and the most recent specification of Mu can be found online [Mu Spec ].

## 2.3 PyPy and RPython

RPython is a compilation and support framework for developing interpreters of dynamic languages. It allows language interpreters to be written in a subset of Python (RPython), with high-level abstractions and few dependencies to lower level details.

The RPython compilation framework produces a concrete virtual machine by inserting lower level details during the translation process. The framework is also highly customisable, enabling new translator back-ends to be written for different platforms.

The most attractive feature of RPython is the ability to generate JIT compilers in a language independent way. This allows any interpreters written in RPython to automatically have a powerful meta-tracing JIT, adding high performance to implementations with zero cost [Bolz and Tratt 2015].

### 2.3.1 Language implementations using RPython

The automatic generation of a powerful meta-tracing JIT and the ease of language implementation through high level abstractions have made RPython very attractive in the community of language developers. Implementations of many languages have been developed using RPython. These include Python (PyPy [Rigo and Pedroni 2006]), Racket (Pycket [Bauman et al. 2015]), Erlang (Pyrlang [Huang et al. ]), Javascript (PyJS [Zalewski ]), R (Rapydo [Hager ]), PHP (HappyJIT [Homescu and Şuhan 2011]), and Haskell (PyHaskell [Thomassen and Hetland ]). A simple teaching language Simple Object Machine (SOM) has also been implemented at the time of writing [Marr and Ducasse 2015].

Thus it is highly desirable and strategically critical to develop a language client for RPython. When fully implemented, it makes available for Mu all of other languages that are written in RPython. A core part of the goal is developing a back-end targeting Mu for RPython.

### 2.3.2 Existing RPython back-ends

RPython currently only has C as the default back-end. LLVM back-end has also been attempted yet failed many times. It is also argued, in the RPython documentation, that having LLVM as a back-end is pointless, since the C code can be compiled using LLVM [RPython Doc ].

Javascript back-end has also been attempted by compiling the generated C code using Emscripten [Emscripten ; PyPy.js ]. This allows PyPy interpreter to run inside the web browser.

The goal of a Mu back-end thus opens up another possibility for RPython.

## 2.4 Summary

Programming language implementations have become more complex as the level of abstraction gets higher. Compared to the monolithic approach, the use of VMs in language implementation frees the language implementers from the challenges of lower level details, allowing them to focus on higher level language features. However many existing VMs are heavy-weight and large, offering far more than the need of a language client. And the focus on their supporting language make them a poor match for new languages.

$\mu$VMs provide a thin layer of abstraction over just three core concerns in language implementation, namely concurrency, garbage collection and compiler backend. Being minimal and light-weight, yet maintaining maximum flexibility, $\mu$VMs aim to be a desirable, unifying substrate on which language implementations and other virtual machines can be built.

To test the design and claims of $\mu$VMs, RPython is taken as the language of choice. RPython is a powerful compilation framework offering both useful high level language abstractions for developing interpreters, and the ability to automatically generate a high performance meta-tracing JIT. This has attracted developers to implement many other programming languages using RPython and have achieved promising performance. Thus developing an RPython language client is a strategically critical step in bringing many language implementations onto Mu.

Chapter 3

# RPython Translation Process

Being a back-end to the RPython compilation framework, MuPy intercepts the default translation and redirects it to the Mu back-end translation tasks. It is thus important to understand the RPython translation process, and the kind of data structures MuPy will be accepting as input. This chapter aims to provide an overview to the RPython translation process necessary for understanding the MuPy project. More detailed information can be found on RPython online documentation [**?**].

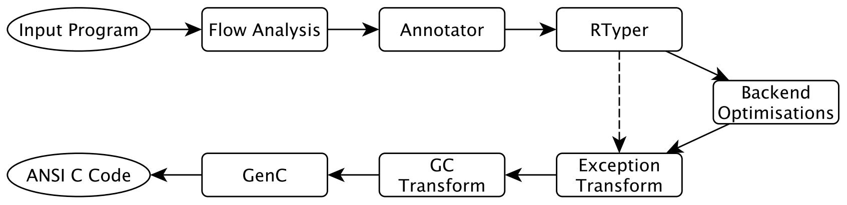
Section 3.1 gives a general overview of the RPython translation process. Section 3.2 gives more detail on the flow graph representation of the program. The translation of the flow graph representation is the main theme of Chapter 4. Section 3.3 provides an overview of the type inference process that is at the core of RPython. The mapping of the inferred type representation is discussed in Chapter 5. Section 3.4 gives an overview of the back-end optimisations and code generation process. Finally, Section 3.5 provides a summary to this chapter.

## 3.1 Overview

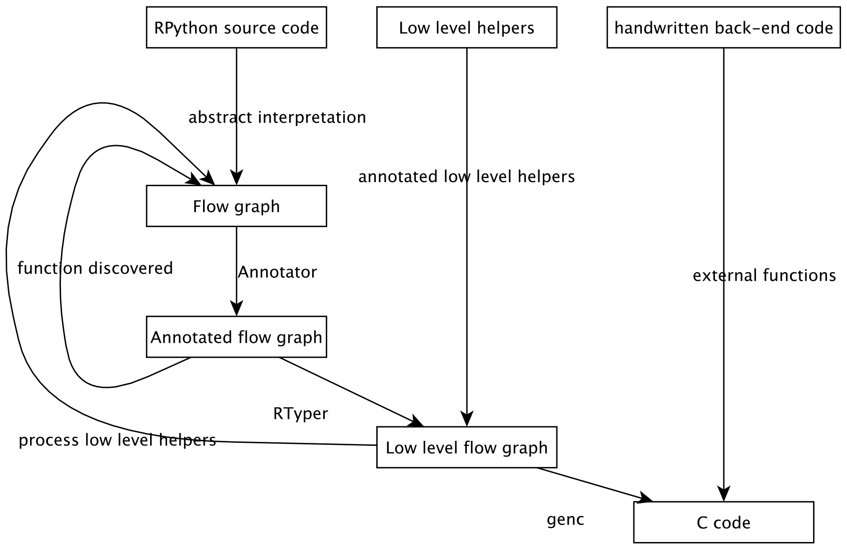
RPython is a restricted subset of Python that is statically typed. As a language, it has many features that are very helpful in implementing language interpreters. It offers many high level abstract data types such as classes, lists and dictionaries; as well as useful language features such as hierarchical exception handling, garbage collected memory and JIT. Thus the task of the translation process, and in general of the RPython compilation framework, is to break down these high level abstractions and language features to the level of the target language (*e.g.*. C).

As shown in Figure 3.1[RPython Doc ], the translation process consists of several stages.

The input program source code is first converted into control flow graphs by abstract interpretation and flow analysis. Then, the annotator does the type inference, deducing the general type information of variables. During this process, RPython discovers extra functions that are called by the input source code. These functions are also annotated and included in the set of graphs. At the RTyper stage, the general type information is specialised using the Low Level Type System (LLTS) that closely resembles C. The general operations on these variables and objects are also specialised. The



(a) Components of the translation process.



(b) Overview of translation process.

Figure 3.1: Illustrations for RPython translation process.

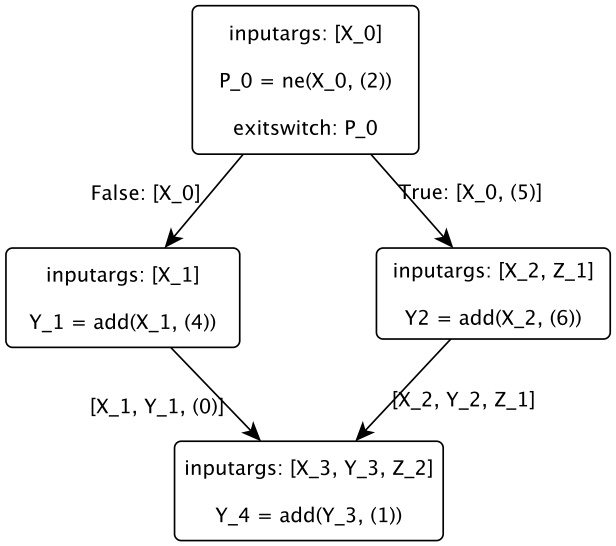


Figure 3.2: Example CFG.

low level helpers contain implementations of various data types and their methods. These helper functions go through the same process of being transformed into flow graphs and having their type specialised.

The next step is the optional back-end optimisations. These optimisations include inlining, no-op removal etc. Exception and GC transformations are also applied to include the C compatible implementations for these high level language features. Then the flow graphs are translated into C code, and it can be compiled using other C compilers into a binary that is able to be executed on the specific platform. Other C source code in the C back-end that implements various features using external functions are also included during the compilation.

Thus the translation process gradually breaks down the high-level data types, operations and program structures to bridge the semantic gap between RPython and C.

Note that the RPython compilation framework is written in a meta-circular manner using Python. This allows the translation process to use high level language features offered by Python. Thus a normal Python interpreter is required to run the RPython translation engine.

## 3.2 Control Flow Graph Representation

The translation engine first imports the RPython source code, performs necessary initialisations for the module, and creates a bytecode representation for the program.

In flow analysis, an abstract interpreter takes the bytecode and performs abstract interpretation to produce the Control Flow Graph (CFG) intermediate representation. The abstract interpreter follows the bytecode instructions, and records the operations performed on Python objects into basic block structures. These basic blocks contains sequences of consecutive operations. Each operation in a block has a name, a list of arguments, and a result variable. Each block has a list of input arguments forming a closed scope. All of the variables referenced in the sequence of operations within a block, apart from the ones created by the operation results, are defined in the input arguments. Each block also has a list of outgoing links to other blocks. Variables and constants can be carried on links to assign values to the input arguments of the destination block. These links are the key to representing branching, loops, and exception handling control structures. The container class for these blocks and links is called `FunctionGraph`. Each instance of the class corresponds to the CFG representation of an RPython function. Figure 3.2 shows an example of these elements.

Each graph contains an entry block whose input arguments are the parameters of the function. From the entry block, all of the other blocks can be traced through the outgoing links. Each function also contains two unique blocks: a return block and an exception raising block. These two blocks have no operations and no outgoing links. The return block accepts one argument that holds the value to be returned by the function, indicating a single value return policy. The exception block accepts both the type and value object of the exception as arguments. The semantics indicate that both are thrown and stored in global memory, and special variables in reference to them are used in link arguments. More discussion on exception handling is presented in Chapter 4.

Each block has an exit switch variable (or some special constants such as `last_exception`, see Chapter 4) that expresses the conditional branching semantics. Its value is checked and compared against the `exitcase`s value of each outgoing link, and the link that has the matching `exitcase` is taken.

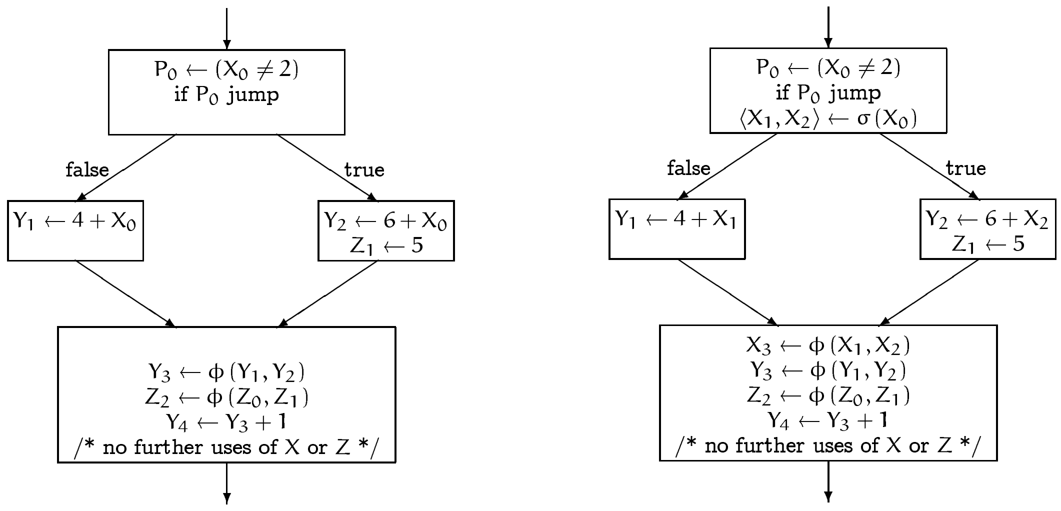
### 3.2.1 SSA & SSI

RPython's CFG has Single Static Information (SSI) form, which is an extension to the Single Static Assignment (SSA) form. Figure 3.3 shows a comparison between SSA and SSI form [Ananian and Rinard 1999].

In SSA form, each variable has exactly one definition point (*i.e.* an operation). At branched blocks the operations can refer to already defined variables, and they create alternate names that refer to the same entity in the code ($Y_1$ and $Y_2$ in Figure 3.3). At join points however, these alternate names must be unified using the $\phi$ functions to preserve the single-assignment property. Each $\phi$ function performs an assignment based on the control flow path taken to reach the joint point (*e.g.* $Y_3 \leftarrow \phi(Y_1, Y_2)$ operation in Figure 3.3).

SSI form, in addition to SSA, recognises that information is generated at branch points as well. Therefore SSI defines a $\sigma$ function that generates new names for splitting a variable at a branch point. This enables the construction of a one-to-one mapping between variable names and information about the variable at each point in the program. Such information can then be propagated to aid the analysis of the program.

In RPython, links can carry arguments of defined variables and constants, and the



$P_0 \leftarrow (X_0 \neq 2)$
if $P_0$ jump
false
true
$Y_1 \leftarrow 4 + X_0$
$Y_2 \leftarrow 6 + X_0$
$Z_1 \leftarrow 5$
$Y_3 \leftarrow \phi(Y_1, Y_2)$
$Z_2 \leftarrow \phi(Z_0, Z_1)$
$Y_4 \leftarrow Y_3 + 1$
/* no further uses of X or Z */
$P_0 \leftarrow (X_0 \neq 2)$
if $P_0$ jump
$\langle X_1, X_2 \rangle \leftarrow \sigma(X_0)$
false
true
$Y_1 \leftarrow 4 + X_1$
$Y_2 \leftarrow 6 + X_2$
$Z_1 \leftarrow 5$
$X_3 \leftarrow \phi(X_1, X_2)$
$Y_3 \leftarrow \phi(Y_1, Y_2)$
$Z_2 \leftarrow \phi(Z_0, Z_1)$
$Y_4 \leftarrow Y_3 + 1$
/* no further uses of X or Z */

Figure 3.3: Illustration of SSA (left) and SSI (right) representation. $\phi$ function joins different names into one, and $\sigma$ function splits one name into different names.

input arguments for each block define new names for the variables. Together they effectively implement both $\phi$ and $\sigma$ functions. In branched blocks, the variables in input arguments are given new names ($\sigma$ function). In joint blocks, the incoming variables are given new names ($\phi$ function).

## 3.3 Type annotation and transformation

RPython is a statically typed language. Though not explicitly specified, the types of the variables must be inferred at compile time. The task of type inference and specialisation is performed by the annotator and the RTyper.

### 3.3.1 Annotator

The annotator performs whole-program analysis, and gives a general type annotation to each variable in the flow graph. This annotation describes all of the possible Python objects that a particular variable can contain at run-time. As shown in Figure 3.4, the annotation types form a Python-like, hierarchical structure. With the root class being `SomeObject`, general type inference can identify integers, floats, strings, lists, tuples, dictionaries, class instances etc. When new evidence is discovered suggesting a new type for an already annotated variable, the annotator attempts to generalise the know annotation with the new information by effectively moving up the type hierarchy. Deducing general type information in this way is a key step in the translation process.

### 3.3.2 RTyper

As the general type information obtained from the annotator is still highly abstract, further specialisation to lower level representations is needed to aid the translation

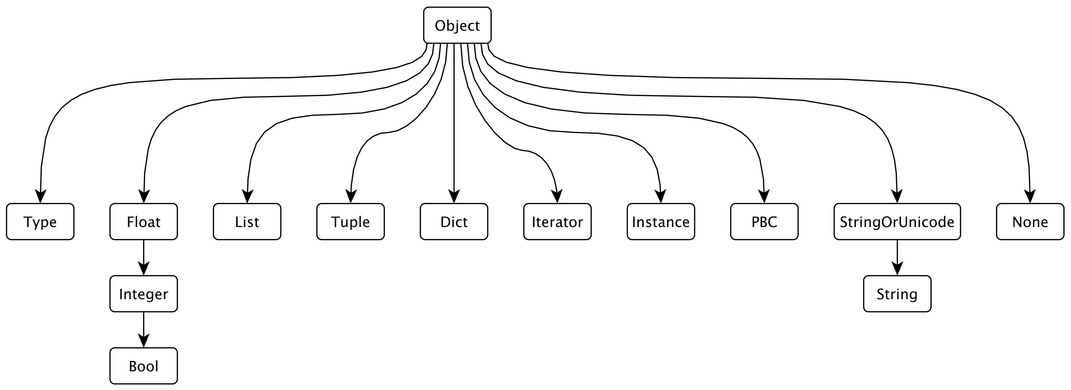


Figure 3.4: Simplified annotation type object hierarchy graph. The actual name of the type class is prefixed with `Some`, ie. `SomeString`.

into the default C back-end. This is the task performed by the RPython Typer (RTyper).

RTyper, similar to the annotator, considers each block of the flow graphs in turn, and performs analysis on each operation with its input arguments. It replaces each high level operation with one or more low level operations, and assigns the specialised low level type to the `concretetype` attribute of all the variables and constants in the flow graphs.

RTyper converts the Python-like high level type annotation to the C-like Low Level Type System (LLTS). In LLTS, there are primitive types such as `Signed` and `Unsigned` integers, `Float`, `Char`, `Bool` and `Void`; container types such as `Struct`, `Array`; and also other types such as `Pointer`, `Function` etc.

For operations, RTyper specialises high-level operations to LLTS operations based on the argument types. For example, `add` is translated into `int_add` if both operands are integers, or a `direct_call` to `ll_strconcat` for concatenating two strings. During this process, low level helper functions that implement method functions for RPython types in LLTS are pulled in to the collection of flow graphs. They are also annotated and specialised using LLTS.

LLTS also contains raw memory access types (*e.g.*. `Address`) and operations, and some compiler intrinsics (*e.g.*. `memcopy`). These elements are mainly used in the implementation of JIT and GC, but some are also used in the implementation of low level helpers (*e.g.* the use of `memcopy` in `ll_strconcat`). This has implications for the MuPy back-end, and is discussed in Chapter 5.

#### 3.3.2.1 Representation of classes and instances

RPython is object oriented, thus implementing the high level concept of classes and instances using low level structs and pointers is an essential part of the translation process. It also has implications for the hierarchical exception handling mechanism in RPython, since it depends heavily on the class inheritance feature. It is thus worth presenting the representation of classes and instances in RPython and LLTS.

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

class B(A):
  def __init__(self, a, b):
    A.__init__(self, a)
    self.b = b
```

(a)

```
struct object_vtable {
  RuntimeTypeInfo * rtti;
  Signed subclassrange_min;
  Signed subclassrange_max;
  RPyString * name; // A: "A", B: "B"
  struct object * instantiate();
}

struct A_vtable {
  struct object_vtable super;
}

struct B_vtable {
  struct A_vtable super;
}

struct object {
  struct object_vtable* typeptr;
}

struct A {
  struct object super;
  Signed inst_a;
}

struct B {
  struct A super;
  Signed inst_b;
}
```

(b)

Code 3.5: RPython LLTS class representation example. The classes defined in the RPython source code (3.5a), are translated through LLTS to the shown struct definitions in generated C code (3.5b).

Classes in LLTS are represented using virtual tables. RPython only allows single class inheritance. The root class `object` defines a virtual table that describes the information about a subclass. Every subclass embeds the virtual table of its super class in the field `super`. Thus it allows the virtual table be the header of any class structs. Through pointer casting to the root `object` class, the class information header can be accessed in every subclass.

The two fields worth noting are the `subclassrange_min` and `subclassrange_max`. They are two integers computed to reflect the class hierarchy. The two fields are used by the `ll_issubclass` function to check whether a class is a subclass of another class.

The instance struct of the root `object` class contains only a `typeptr` field. This field points to the class struct of a subclass, allowing an instance of the subclass to access its type information. Similar to the strategy used to describe the classes, the instance struct of a subclass embeds the instance struct of its super class as a header. Thus pointer casting is essential in accessing instance attributes across the class hierarchy.

Code 3.5 shows an example of this representation strategy.

Only custom defined classes are represented in this way. All the classes (*e.g.*. `StdOutBuffer`) and built-in types (strings, lists etc.) of RPython do not have a virtual table describing their class information.

An important assumption that RTyper makes is that all the variables in the flow graphs should only contain 'simple' values, ie. primitives and `Pointer`s. Thus all instances of container types are heap objects referenced by pointers. The implication of this is also discussed in Chapter 5.

## 3.4 Back-end optimisations and code generation

After type inference and specialisation, the level of abstraction in the flow graphs has been brought much closer to the default C back-end. Before generating C code, optional back-end optimisations and necessary exception and GC transformations are to be performed.

The optional back-end optimisations perform static analysis and simplification on the flow graphs. Inlining, for example, removes some small function calls and merges the blocks in the callee into the graph of the caller. It also attempts to merge a sequence of blocks that perform conditional branching on the same variable into one block that has multiple exits based on the value of the exit switch. This transformation is useful, since Mu offers a `SWITCH` instruction that branches to different code blocks depending on the value of the switch variable.

After the optional back-end optimisations, the C back-end performs exception and GC transformation, to include the implementation of exception handling and GC into the flow graphs. More details about the RPython's default exception transformation strategy is discussed in Chapter 4. The details of GC transform is not, however, included in the discussion of this thesis. This is because it is necessary to strip away any

GC operations from RPython, leaving it to Mu's built-in GC support.

After these transformations, the flow graphs are ready for C code generation. The generated C code is combined with other hand-coded C back-end support code, and compiled using a normal C compiler into machine executable code, thus reaching the end goal of the compilation process.

## 3.5 Summary

RPython, as a statically typed, restricted subset of Python, provides high level abstractions over data and data structures that are helpful in writing language interpreters. To bridge the semantic gap between RPython and the default back-end C, the translation process creates flow graph intermediate representation of the program and breaks down the abstractions through various stages. It starts by constructing control flow graphs from the RPython bytecode, then infers and annotates the types of the variables. To break down the abstraction of types and operations, RTyper specialises these elements using LLTS that closely resembles C. Optional back-end optimisations can then be performed on the typed graph to assist in code generation. The flow graphs are put through exception and GC transformations and code generator to produce a collection of C source files. These C files are compiled by normal C compilers, to produce an executable file.

Having had an overview of the whole translation process, the next few chapters introduce the stages in Mu back-end translation process that begins after the back-end optimisation stage.

Chapter 4

# Control Flow Graph Transform

As presented in Chapter 3, RPython source code is turned into Control Flow Graphs (CFGs). Mu IR and RPython CFG bear similar structures, that in both representations the body of function definitions are made up of blocks containing operations/instructions. These similarities make the translation intuitive and straightforward, yet the differences between the two representations need to be addressed. This chapter focuses on the process in bridging the difference between the two program structures, and also discusses the translation of the exception handling mechanism as an important task of this stage.

Section 4.1 gives a quick view of the Mu IR structure, highlighting the differences. Section 4.2 briefly describes some necessary transformation procedures. Section 4.3 explores the RPython exception handling mechanism and the difference in implementation strategies by C and Mu back-end.

## 4.1 Mu IR program structure

Mu IR code bundles are made up of top level definitions such as types, constants, global cells, and functions. Constants are values and global cells are spaces in the global memory. Each entity in Mu has a globally unique identifier. The global name (starts with '`@`') of a local variable (starts with '`%`') has the function name as the prefix. Code 4.1 shows an example Mu IR code bundle.

Similar to RPython CFGs, Mu functions are made up of named blocks containing instructions. However, instead of explicit links that represent argument carrying jumps, Mu terminates a block with explicit terminal instructions (*e.g.* branches, returns, instruction with exception clauses etc.). Thus one of the tasks in transforming the RPython CFG is to append terminal instructions to blocks.

Mu IR adopts SSA form instead of SSI. `PHI` instructions at the beginning of a block corresponds to the $\phi$ function of the SSA form at joint points. This also differs from RPython's implicit implementation of $\phi$ and $\sigma$ functions.

```
.typedef @i64 = int<64>     // 64-bit int
.typedef @double = double
.typedef @void = void
.typedef @refvoid = ref<@void>  // void reference

.const @i64_0 <@i64> = 0
.const @answer <@i64> = 42

.typedef @some_global_data_t = struct <@i64 @double @refvoid>
.global @some_global_data <@some_global_data_t> // a global cell contains a
    struct

.funcsig @BinaryFunc = @i64 (@i64 @i64) // @i64 -> @i64 -> i64
.funcdecl @square_sum <@BinaryFunc>     // declaration of an undefined
    function

.funcdef @gcd VERSION @gcd_v1 <@BinaryFunc> (%a0 %b0) {
    %entry:
        BRANCH %head
    %head:
        %a = PHI <@i64> { %entry: %a0; %body: %b; }
        %b = PHI <@i64> { %entry: %b0; %body: %b1; }
        %z = EQ <@i64> %b @i64_0
        BRANCH2 %z %exit %body
    %body:
        %b1 = SREM <@i64> %a %b
        BRANCH %head
    %exit:
        RET <@i64> %a
}
```

Code 4.1: Example Mu IR code bundle.

## 4.2 Transformation procedures

### 4.2.1 Necessary RPython back-end optimisations

RPython back-end optimisations are optional. However there are a few that are quite useful and necessary for the graph transformation stage of MuPy.

#### 4.2.1.1 No-op removal

RPython CFGs contain no-ops such as `same_as` (a simple assignment of an SSA variable to another). Such operations are redundant and cannot be translated since there is no corresponding Mu instruction. Removing such operations also involves renaming the subsequent uses of the result variable. It is thus convenient to take advantage of the default RPython no-op removal routine to perform the task.

#### 4.2.1.2 Exception raising operations to function call

LLTS explicitly distinguishes between operations that are guaranteed to succeed and operations that may fail with an exception.

Mu does provide exception support for instructions, but it doesn't quite match the expectation of RPython. For example LLTS has an operation `int_add_ovf` that raises an `OverflowError` when the addition result exceeds the length of `int` (32-bit). But according to the specification of Mu, addition overflow is silently dealt with by modulation according to the size of the operant type. Thus to correctly implement such a mechanism, the MuPy back-end compiler needs to generate code for overflow checking, this can be unnecessarily burdensome.

To simplify the translation process, this back-end optimisation can be used to take advantage of the existing implementation of such operations in RPython. This optimisation turns all of the exception raising operations to a function call to the implementation function. This implies that all of the exceptions will be raised from a `CALL` instruction, which may simplify some code analysis as well.

#### 4.2.1.3 SSI to SSA conversion

RPython also conveniently provides the conversion between SSA and SSI form. According to the documentation, the function `SSI_to_SSA` renames the variables in the flow graph in conformity to the SSA form. An alternative approach without using this optimisation involves inserting `PHI` instructions at the beginning of every block, thus emulating the renaming functionality of block input arguments. However, using the provided SSI to SSA conversion optimisation significantly reduces the amount of `PHI` instructions needed, resulting in a cleaner graph. Thus it is beneficial to utilise this optimisation.

### 4.2.2 MuPy graph transformation tasks

MuPy needs to implement graph transformation tasks that are not provided by RPython back-end optimisations.

#### 4.2.2.1 Forest pruning

Apart from the function graphs needed by the source program, RPython also includes many other function graphs. These functions aim to serve as the entry point for the executable target. These functions are not necessary, as the source program is loaded and initialised by the launcher (see Chapter 6). Together with the program entry point function, they form multiple roots when constructing a set of call tree from the function graphs. But in fact only the call trees formed by the program entry point is necessary. The other trees can be pruned by removing their roots.

The graph chopping algorithm takes the graph of the actual program entry point as the root, traverses the call tree and marks all of the function graphs that can be reached. All of the other function graphs that cannot be reached are thrown away. This thus produces a clean set of necessary graphs.

#### 4.2.2.2 Terminal instructions

The flow between blocks in Mu functions is directed by the explicit branching instructions. Though this differs from RPython's approach of using block exit links, it is generally straightforward to translate the exit links into branching and return instructions. There are cases where the exit switch is not a binary boolean variable, resulting in more than two exits (`if...elif` chain for a primitive type variable). Such cases can be translated using `SWITCH` instructions.

There are also cases where the exit switch is `last_exception`, denoting semantics of exception handling. The translation of exception handling is discussed in Section 4.3.

#### 4.2.2.3 Transitional blocks

Since links in RPython can carry arguments, thus it is possible for two links to have the same source and destination blocks yet carry different arguments, as shown in Figure 4.2a. The `PHI` instruction in Mu assigns the value to the result variable based on the incoming block. Thus in this case it cannot distinguish the different values that come from the same block.

To resolve the issue, MuPy adds empty transitional blocks between the source and destination blocks, thus distinguishing the different values by different incoming blocks, as shown in Figure 4.2b.

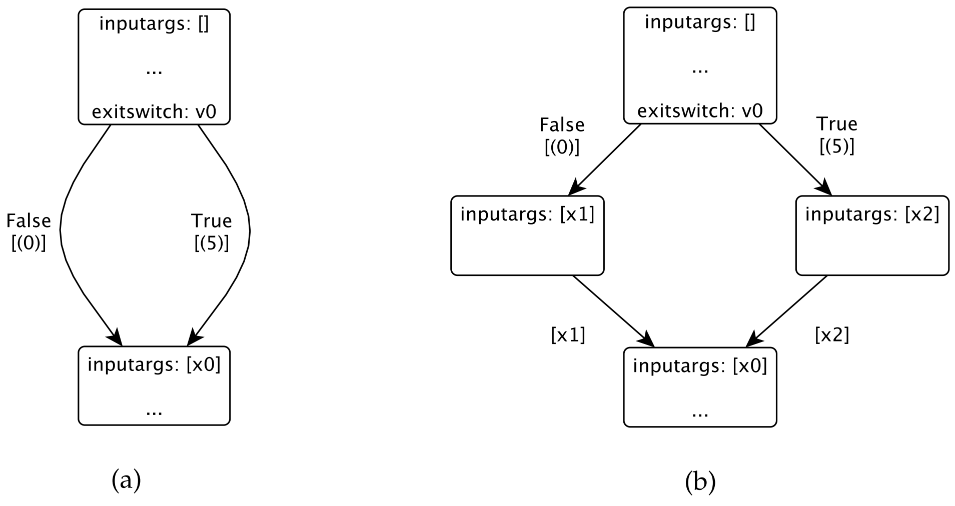


Figure 4.2: Adding transitional blocks to break down the argument carrying link semantics.

```
try:
    raise_error_1()
except MyError as e:
    print e.message
except IndexError:
    print "Caught"

class MyError(Exception):
    def __init__(self, msg):
        self.message = msg

def raise_error_1():
    raise MyError("1st msg")
```

Code 4.3: Example RPython code that uses exception handling.

## 4.3 Exception transform

RPython has a built-in hierarchical exception handling mechanism, using Python's `try except` construct and exception class hierarchy. This high level feature can give great assistance to language implementation. Code 4.3 shows an example RPython code that uses this feature.

### 4.3.1 Representing exception handling in a CFG

In RPython CFG, the exception handling semantics are expressed using a special exit switch constant and different exit cases for out going links.

The special exit switch constant `last_exception` denotes the exception occurrence at run-time by the last operation of the block. In each exit links of this block

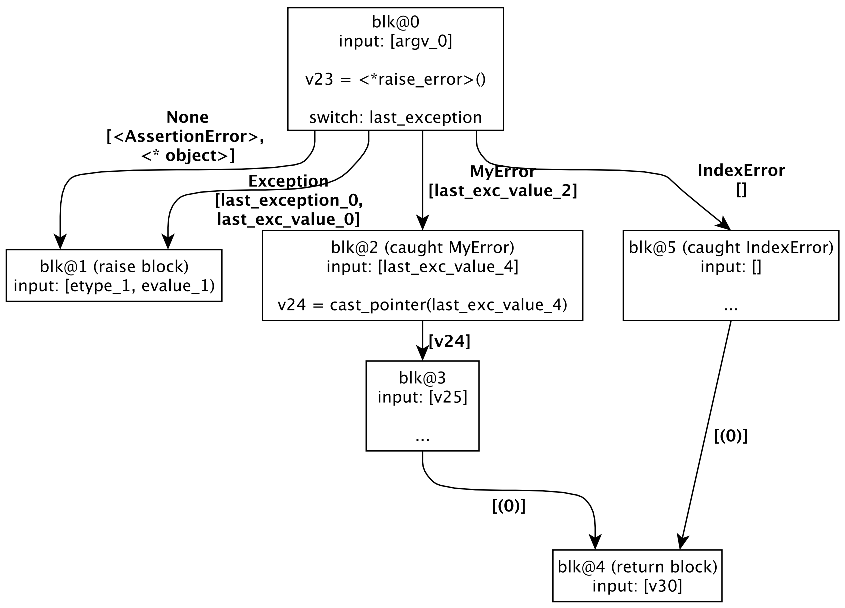


Figure 4.4: The corresponding CFG of Code 4.3 (some unrelated operations are neglected for simplicity). Note that in the non-exceptional case it attempts to raise an `AssertionError`. This is because Code 4.3 definitely raises an exception. Thus this non-exceptional case is indeed 'exceptional' if it indeed gets taken.

the `exitcase` attribute is set to an exception type class, except the first link, whose `exitcase` attribute is set to `None` to denote the non-exceptional path. The semantics of the representation is that if an exception has happend, the implementation should check through the links to find the matching exception path. To re-raise a thrown exception that is not explicitly caught by this block, a link with the `exitcase` set to the root exception type class `Exception` is appended at the end of the links, catching such an exception, and directs the flow to the raise block to be re-raised.

There are two special variables, `last_exception` and `last_exc_value` that appear as link arguments. They are assigned with the type class and value of the occurred exception at run-time.

Each graph has a unique raise block that accepts as input arguments the exception type class and value object. Though the block does not contain any operations, the exception raising semantics is supposed to be specialised depending on the back-end. Figure 4.4 illustrates these elements.

### 4.3.2 RPython exception transform

C does not support exception handling. Thus to specialise exception handling with the semantics of C, the default RPython exception transformation procedure needs to implement every detail. This includes the global exception data, function abnormal returns, exception occurrence checking, catching, type matching, exception re-raising

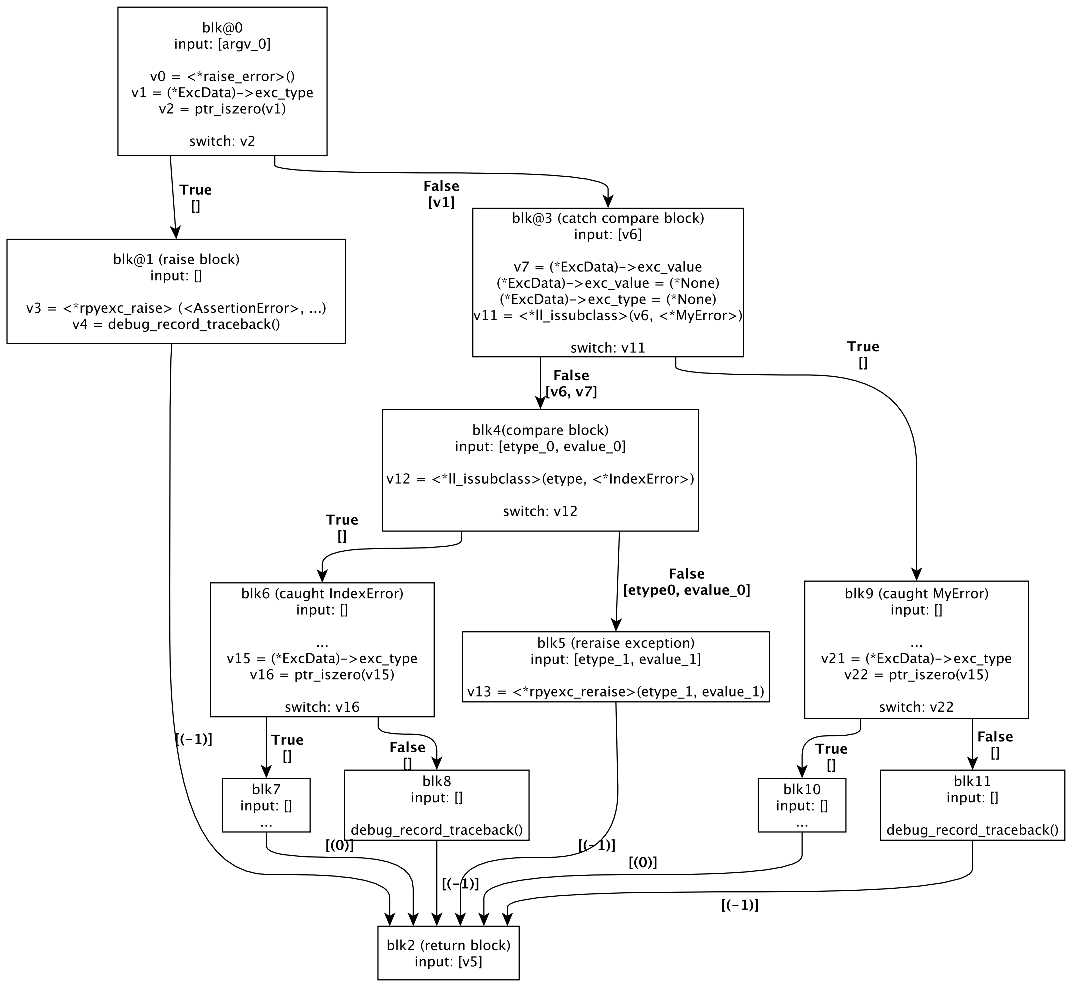


Figure 4.5: Illustration of RPython exception transformed CFG in Figure 4.4 (some of the operations have been reformatted to save space). `blk0` checks the occurrence of exception after calling an exception raising function. `blk3` retrieves the exception information and zeros the global struct. `blk3` also, along with `blk4`, checks the caught exception type, and performs the corresponding operation in `blk6` and `blk9` if a matching is found. `blk6` and `9` checks the occurrence of exception after the call to `rpython_print_item`, which may throw exceptions due to the call to the string concatenation function. If an exception has occurred, it records the debug trace and returns abnormally.

and stack unwinding.

When raising an exception in a callee, the default strategy packs the exception type and value into a global `ExcData` struct through the `rpyexc_raise` function. Then the callee function returns with an abnormal value to direct the flow back to the caller.

The caller function then checks the occurrence of the exception by checking exception type pointer stored in the global `ExcData` struct. If the pointer is non-zero (non-`NULL`), which means an exception has been raised, the caller branches to a series of blocks that retrieve the exception information and check the matching exception type using the `ll_issubclass` function.

This is a straightforward and intuitive approach to implement exception handling in C. Such an approach can be easily translated to Mu, but it will not take advantage of the exception catching feature that Mu provides, which is intended to support the exception handling feature of client languages.

### 4.3.3 MuPy exception transformation

#### 4.3.3.1 Exception handing in Mu

Mu provides its exception catching feature using two instructions, `THROW` and `LANDINGPAD`, and exception control clause in the some other instructions (such as `CALL`).

The `THROW` instruction takes a reference to any type, and throws it as the exception object. The `LANDINGPAD` instruction catches the thrown exception object as a void reference (since it does not know the exception object type) at the start of a exception handling block. The program can then cast the reference using the `REFCAST` instruction to uncover the exception information. Instructions such `CALL`, `NEW` etc. can have an optional exception clause that specify the normal and exception flow.

Therefore what Mu offers essentially is the throwing and catching of exception objects, the mechanism to interrupt program execution, normal and exceptional flows, and stack unwinding. Grounded on such support, the language client can then focus on the representation of exception objects and exception type hierarchy. This echoes the goal of Mu to hide the implementation details of low level mechanisms, thus freeing the language designers to focus on the higher level language features.

#### 4.3.3.2 Transformation strategy

Building on the support of exception catching from Mu, the tasks left for MuPy to do are to create instructions to initialise the exception object, and the blocks of catching and checking the matching exception types.

Since Mu can only throw one object reference as the exception object, a similar strategy of packing the exception type and value into a single struct (`MuPyExcData`) can be adopted. A pack block is created to allocate a struct of this type on the heap and pack the exception type and value, before passing the packed object reference to the exception raising site (*i.e.* the unique raise block in a flow graph). A `THROW` instruction is placed in the raise block to interrupt the program execution and throw the initialised `MuPyExcData` object.

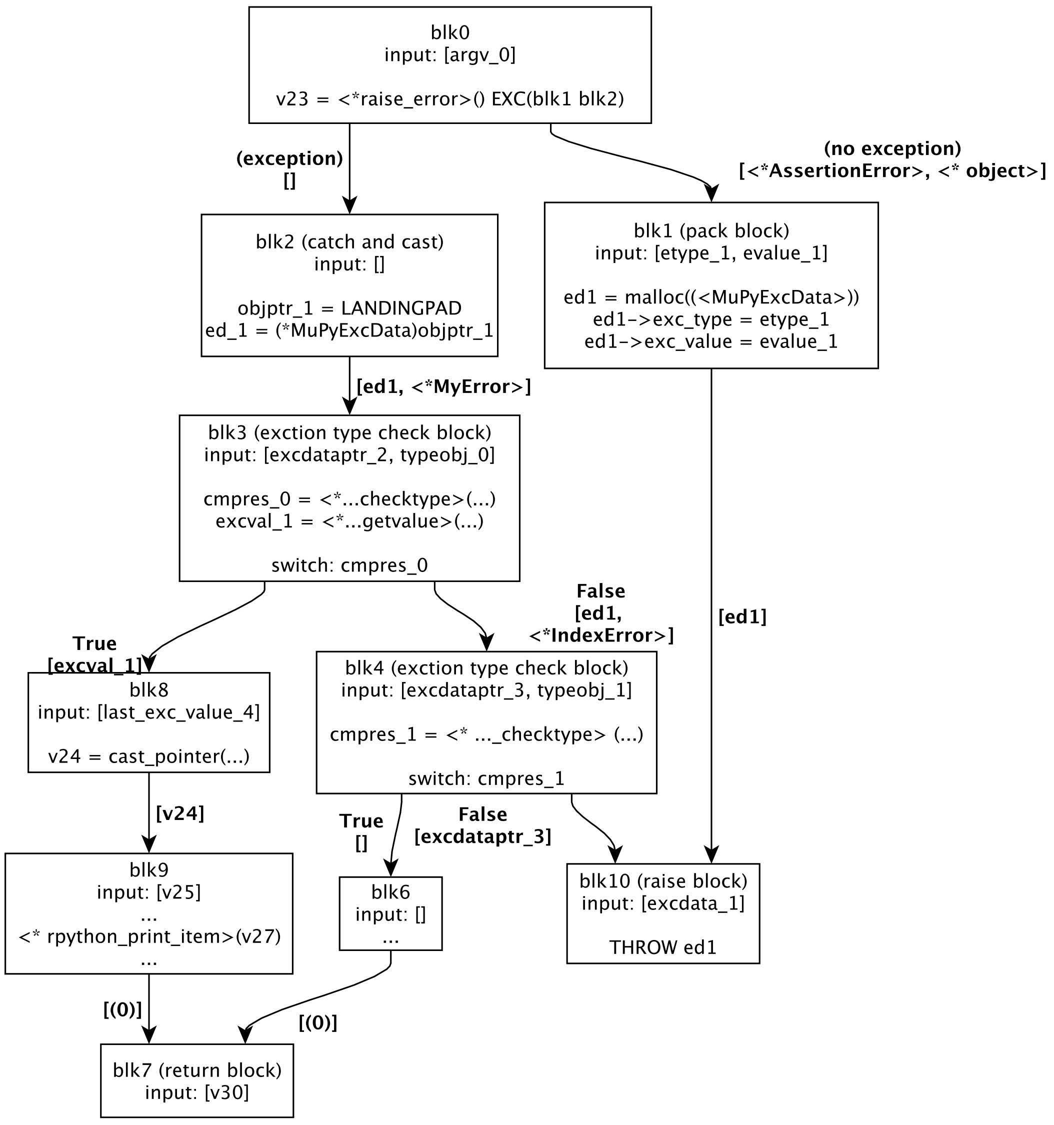

blk0
input: [argv_0]
v23 = <*raise_error>() EXC(blk1 blk2)
(exception)
[]
(no exception)
[<*AssertionError>, <* object>]
blk2 (catch and cast)
input: []
objptr_1 = LANDINGPAD
ed_1 = (*MuPyExcData)objptr_1
blk1 (pack block)
input: [etype_1, evalue_1]
ed1 = malloc((<MuPyExcData>))
ed1->exc_type = etype_1
ed1->exc_value = evalue_1
[ed1, <*MyError>]
blk3 (exction type check block)
input: [excdataptr_2, typeobj_0]
cmpres_0 = <*...checktype>(...)
excval_1 = <*...getvalue>(...)
switch: cmpres_0
True
[excval_1]
False
[ed1,
<*IndexError>]
[ed1]
blk8
input: [last_exc_value_4]
v24 = cast_pointer(...)
blk4 (exction type check block)
input: [excdataptr_3, typeobj_1]
cmpres_1 = <* ..._checktype> (...)
switch: cmpres_1
[v24]
True
[]
False
[excdataptr_3]
blk9
input: [v25]
...
<* rpython_print_item>(v27)
...
blk6
input: []
...
blk10 (raise block)
input: [excdata_1]
THROW ed1
[(0)]
[(0)]
blk7 (return block)
input: [v30]

Figure 4.6: CFG after MuPy exception transform. The `EXC` clause in `blk0` specifies that the normal and exceptional paths are directed to `blk1` and `blk2` respectively. `blk1` packs the exception type and value and pass the package to raise block (`blk10`). `blk2` catches the thrown exception with a `LANDINGPAD` instruction, and casts the exception object pointer into `MuPyExcData` struct reference. `blk3` and `blk4` checks the type of the caught exception. If the thrown exception fails to match, it is passed to `blk10` to be re-raised.

At the call site of the caller, MuPy defines an `EXC` clause to the `CALL` instruction to direct the exceptional flow to a series of exception catching and process blocks. MuPy catches the thrown exception using the `LANDINGPAD` instruction, and `REFCAST`s it to uncover the exception information. Then, like RPython, a series of blocks are created to check the exception type using the `ll_issubclass` function call. To re-raise the exception, the flow is directed to the raise block with the caught exception data, by passing the pack block. Figure 4.6 illustrates this strategy. [1]

In RPython, exception transformation is performed after the optional back-end optimisations as part of the C code generation stage. However MuPy puts the exception transformation before back-end optimisations. The reason for the decision is that the MuPy exception transformation generates helper functions; these functions, along with the other functions, need to pass through the same back-end optimisations, due to the reliance of the many convenient and necessary optimisations to simplify and clean the CFG. This decision also has draw backs, since the MuPy exception transformation breaks some rules about the flow graph model that are enforced by RPython. It could be left as a future work to explore this decision further.

## 4.4 Summary

The similarity in structure between RPython CFG and Mu program structure gives a good starting point for translation. The difference in the graph structures can also be easily bridged.

Many simplification, transformation and cleaning tasks can be done by utilising the RPython back-end optimisations. These include no-op removal, converting exception raising operations to function calls to relieve the compiler of implementation detail, and SSI to SSA transformation.

There are still other cleanning tasks to be done by MuPy. These include pruning out the graphs unreachable from the program entry point, leaving a clean set of only necessary function graphs.

One of the transformation tasks is inserting the transitional blocks. The links in RPython CFG contains additional semantics that allow two links to have the same source and target yet carrying different arguments. This cannot be directly translated to Mu according to the current IR specification. Inserting transitional blocks bridges the differences.

RPython exception transformation is implemented based on the assumption that the target language does not have exception handling and program interruption support. This causes the transformation process to implement every detail of hierarchical exception handling. However, by adopting Mu's exception catching mechanism, this can be done more easily and efficiently. Mu's support of exception catching removes

[1]Note that the call to `rpython_print_item` function in `blk9` does not generate checking on the exception occurrence. This is based on the assumption that when an exception clause is not specified to a `CALL` instruction and an exception is thrown down the call stack, Mu will automatically unwind the stack to search for an exception catch at a higher level. Such assumption does not hold generally for all exception raising instructions, but it holds for `CALL` instruction[Mu Spec ].

the need to implement the low level details, allowing the translation framework to implement higher level features such as exception objects and type hierarchies.

Having the graph transformed into the structure that fits well with the Mu function definition, the next chapter explores the translation for the rest of the elements in the graph, ie. types, operations and constants.

Chapter 5

# Mu Typer

One of the core tasks in the translation process is the translation of the type system. RPython uses the RTyper to translate the high level Python-like type annotations to the low level C-like Low Level Type System. Taking the idea from RTyper, MuPy uses the MuTyper to translate the Low Level Type System (LLTS) to the Mu Type System (MuTS). This is done in three subtasks: mapping the type system, translating the constant values and initialised heap objects, and the operations. This chapter describes these three subtasks, presents the issues encountered, and discusses the potential approaches, along with the impact that the discovery of these issues has brought on the research of Mu.

Section 5.1 describes the mapping of type system and discusses the differences. Section 5.2 describes the translation of values, discusses the problem of heap object initialisation, and how it has served as a motivation for Mu to include HAIL in the recent specification. Section 5.3 describes the specialisation of operations, and specifically discusses the problem of direct memory access operations and some compiler intrinsics, and how this problem has led to the MuNI feature being developed.

## 5.1 Mapping the type system

Just as RTyper specialises the Python-like type annotation to LLTS, MuTyper iterates through the variables and constants in the flow graphs, and 'specialises' their type recursively from LLTS to MuTS. Though LLTS is C-like, many elements in LLTS have a direct MuTS counterpart. A possible mapping between LLTS to MuTS is shown in Table 5.1.

### 5.1.1 Discussion and issues

#### 5.1.1.1 Variable sized `Structs`

In MuTS, the only variable sized type is `hybrid`, the size of all the other types are fixed into the type definition at compile time. Consequently also, `struct`s are not allowed to contain any `hybrid` fields. However, in LLTS, `Struct`s are allowed to contain one trailing `Array` field, thus can be made variable size. Such cases (*e.g.* strings in RPython) need to be translated into `hybrid`s. To map a variable sized `Struct` to

| LLTS | MuTS |
|---|---|
| `Signed` | `int<32>` |
| `Unsigned` | `int<32>` |
| `SignedLongLong` | `int<64>` |
| `UnsignedLongLong` | `int<64>` |
| `Char` | `int<8>` |
| `Bool` | `int<1>` |
| `Float` | `float` |
| `SingleFloat` | `float` |
| `LongFloat` | `double` |
| `Void` | `void` |
| `Struct` | `struct` |
| `FixedSizeArray` | `array` |
| `Array<T>` | `hybrid <int<64> T>` |
| `Ptr<T>` | `ref<T>` |
| `Ptr<FuncType>` | `funcref` |
| `Address` | `ptr` |

Table 5.1: Low Level Type System types and their corresponding Mu Type System types.

`hybrid`, MuPy takes out the non-array fields to form a header `struct`, and set it to be the fixed part of `hybrid`. An additional 64-bit integer (`int<64>`) field is added to the header to store the length information. Figure 5.1a and 5.1c illustrate an example of mapping the string type.

#### 5.1.1.2 `Ptr` and `Address`

Since LLTS is C-like, and GC is implemented separately, the only reference type is pointer (`Ptr`). This type is an object reference, it has, to some degree, liveness semantics. That is, it should always point to an alive object. Thus it should be translated into `ref` type in Mu, with the implied GC and liveness semantics. In the RPython's description of memory layout, there is a definition of the type `Address` as one of the `Primitives`. This type can be cased from `Ptr` using the `cast_ptr_to_adr` function. It does not have liveness semantics, it represents simply a memory address.

This type is useful for implementing GC, JIT and other features that require raw memory access. However, with the use of some compiler intrinsics (*e.g.* `raw_memcopy`, see Section 5.3.2), this type also appears in the flow graphs of essential language feature functions.

One of such use is in the implementation of the `copy_string_contents` function, used by the string concatenation function `ll_strconcat`. According to the documentation in the code, after casting the `Ptr` to `Address`, no GC operations are allowed to happen until the `raw_memcopy` operation finishes. This prevents the GC from moving the string in memory, invalidating the address value.

Mu used to not have a type that provides the same semantics. Essentially there was no type in Mu that prevents GC from moving the object. This problem led to the discussion of expanding MuTS to support such need by some language clients. In the most recent version of Mu specification, Mu Native Interface (MuNI) is introduced. It includes the type `ptr` that is isomorphic to the `Address` type, and the object pinning semantics that fixes the location of an object in the heap to prevent GC from moving it. This opens up new strategies in translating the raw memory access features in RPython. This issue and the impact of MuNI is further discussed in Section 5.3.2.

There also other types not used by the core of RPython, but used in different RPython library modules, such as `r_uint` for arithmetic. It would be part of the future work to research the translation of these types and their modules.

## 5.2 Global cells and initialsed heap objects

### 5.2.1 Constant to Global Cells

In RPython CFG, many initialised object instances (*e.g.* `StdOutBuffer`, strings), are represented as `Ptr` referenced `Constant` objects. This means that they are heap objects. Their content values are represented using a set of value types that reflect the structure of LLTS. These LLTS values are recursively mapped to a MuTS value model. Figure 5.1b and 5.1d illustrate the representation of a string reference in LLTS and MuTS value model respectively.

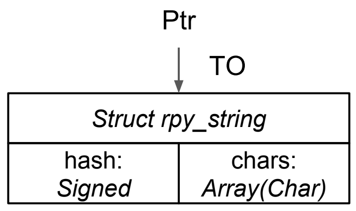
The concept of a constant in Mu differs from the concept of a 'constant' in RPython CFG. These heap objects are represented using `Constant` in the flow graph because their reference pointer value is assumed to be an initialised global value that can not be changed, though the objects they point to can be modified. However, in Mu, constants are values. They are not stored in the heap or the stack, but rather temporarily in registers. Though Mu allows `struct`s and `array`s (fixed sizes) to be constants, it does not allow references (`ref`) to be constants. The alternative would essentially render the heap object immortal, and defeats the purpose of garbage collection (especially in the case of strings.)

These global heap objects in RPython CFG best correspond to the global cells in Mu. These global cells are memory allocation units in the global memory. They are designed to be the counterparts of static or global variables in C/C++ [Mu Spec ], and thus fit well with heap objects.

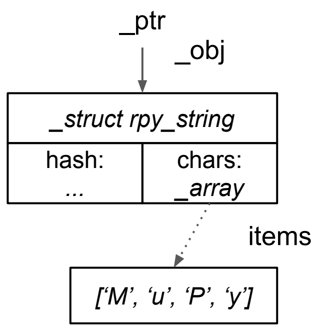
To access the content stored in a global cell, it is necessary to use the `LOAD` instruction to load it into an SSA variable. Thus MuPy inserts several `LOAD` instructions at the entry of the function that refers to these global heap objects. The heap object constants in the graph are also replaced with the SSA variable that contains the loaded content.

### 5.2.2 Heap object initialisation

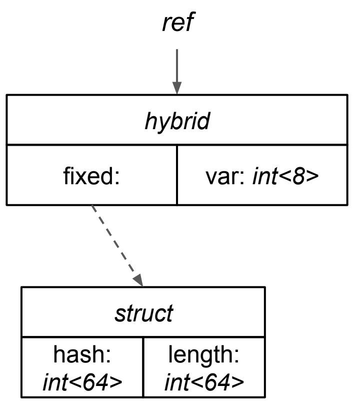
To store objects in global cells, they first need to be initialised. However the RPython CFG does not provide any routines that lay out the operations to initialise them. This



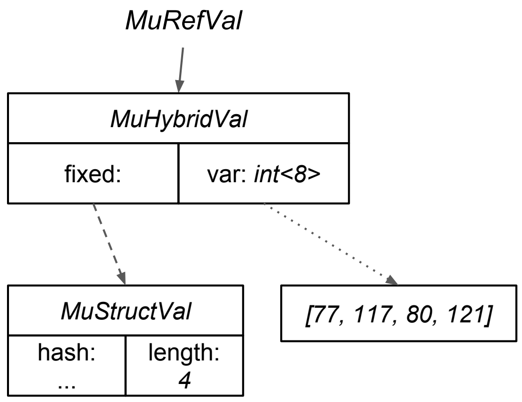
(a) The representation of string type in LLTS.



(b) The representation of the string "`MuPy`" in LLTS.



(c) Mapped string type in MuTS. MuPy puts the 'hash' field and an additional 'length' field together to form a header struct as the fixed part of the hybrid type. Each of the elements in variable part (allocated during run-time) has type `int<8>`.



(d) String object "`MuPy`" in Mu value model.

Figure 5.1: An example of the representation of the type and instance object of the string "`MuPy`" in LLTS and MuTS.

thus raises the issue of heap object initialisation. Mu used to not have any support for this task, all the initialisations need to be done using instructions.

There are two approaches to initialising heap objects and storing them in global cells. One is to generate a bundle entry point function that contains a list of instructions to initialise all the global memory cells. The bundle entry point function can then call the program entry function to start the program, and terminate the program with the `uvm.thread_exit` instruction after the call returns. This is the approach currently adopted by MuPy, taking advantage of the fact that all the information about the heap object is held in memory during the translation process. It also enables the bundle entry point to have top level control over the program. For example, top level exception handling blocks can be added to catch exceptions thrown in the program that is not explicitly caught, and process them accordingly. However, the drawback of this approach is that it significantly increases the size of the code bundle. Depending on the complexity and amount of the objects, the initialisation routine can occupy 20% $\sim$ 50% (line count) of the generated code. [1].

Another approach is to let the language client to handle this task. In addition to the IR code bundle, the compiler also generates another file that contains a list of global cell specifications. With such information, the MuPy language client launcher can construct the heap objects and store them into these global cells. However defining a language that describes the heap object layout and developing a parser for the language can be complex. Thus this approach is not taken by the current version of MuPy.

This need for heap object initialisation support motivated Mu to put forward a Heap Allocation and Initialisation Language (HAIL) in the most recent specification [Mu Spec ]. With HAIL, the difficulty of the second approach is been handled. Based on the known type definition, HAIL provides a standard, Mu-like way of allocating the memory for global cells and initialise their contents. It thus opens up a new and space-efficient way to be explored and integrated into MuPy in the future.

## 5.3 Mapping the instructions

Similar to RTyper, MuTyper also translates and specialises the LLTS operations. Each LLTS operation corresponds to one or more Mu instructions.

LLTS operations are categorised into:

- **numeric operations**: arithmetic on different numeric types, and their conversions;
- **pointer operations**: memory allocation and management, memory access via pointer references, and pointer casting;
- **address operations**: operations on the `Address` type, including some intrinsics;
- **JIT operations**: RPython JIT related operations;

[1]Measurements and estimation taken from the sample and testing programs during development.

- **GC operations**: used by different GCs;
- **JIT & GC interactions**: only used by some GCs;
- **weak reference operations**: operate on weak references in the LLTS memory model;
- **debugging operations**: debug information recording; and
- **miscellaneous operations**.

### 5.3.1 Translating essential operations

RPython, as a compilation framework, includes implementations of GC and a meta-tracing JIT. However since Mu already provides GC and JIT, MuPy seeks to strip away these elements from RPython, and focuses only on the language features. Thus the only operations that need to be translated are some of the numeric operations (the over flow checking operations have been converted into function calls using the RPython back-end optimisations), some of the pointer operations and some address operations.

The translation of numeric operations is generally straightforward. Some of the operations involves introducing extra constants (eg. `int_neg(x)` $\equiv$ `0 - x`). Since constants in Mu are referred to via globally unique identifiers, MuPy employs a global symbol table to keep track of the known mapping between LLTS constant values and Mu constants. MuPy can thus look up a known Mu constant by the desired constant type and value;

The LLTS pointer type, `Ptr`, is mapped to the general heap reference type `ref` in Mu. In Mu, access to the referenced object needs to be done using internal references. These internal references can be obtained from its general reference using the `GETIREF` instruction. Thus a context-free translation of the pointer operations causes all the resulting instruction sequences prefixed with `GETIREF` instructions. Compiler optimisations can be applied in the future to perform static analysis on the code to remove redundant `GETIREF` and other internal reference manipulation instructions, leaving only the essential ones, since these internal references only need to be obtained once.

Since the variable sized RPython `Struct` types are mapped to `hybrid` type, the operations that act on these `Struct`s also need to be translated differently. The operations that access the non-array fields of these `Struct`s (`getfield`, `setfield`), need to be translated into a sequence of instructions that access the fixed header part of the `hybrid` type; and the operations that access the array field (`getinteriorfield`, `setinteriorfield`) need to be translated into sequences of instructions that access the variable array part.

### 5.3.2 Raw memory access operations and Mu Native Interface

The address operations act on the `Address` types. They are used by the RPython JIT, GC implementations, and some RPython standard libraries. Some of the address operations are compiler intrinsics. Take `raw_memcopy` as an example, it is translated into the C `OP_RAW_MEMCOPY` macro. Combined with the source code in the C backend, this operation is directly mapped to the `memcpy` C standard library call.

Initially, these operations could not be translated, since the `Address` type used to not have an equivalent type in Mu. Though they mostly appear in JIT and GC modules, which are out of scope of MuPy, some of them do appear in implementation of essential elements such as strings to achieve highly efficient memory copying.

The strategy that MuPy currently implement is to rewrite functions such as `ll_strconcat`, and replace the use of these compiler intrinsics with functionally equivalent imitations (*e.g.* looping through the elements and copy them one by one). Compared to using the standard C library, this approach has significant performance cost from interpreting these high level instructions. Though it is likely that the Mu JIT will compile these hot loops into machine code to raise the performance at run time, it is still not ideal, and the cost of warming up the JIT still needs to be paid.

It is argued that Mu IR compiler can possibly observe such pattern and translate them in a more efficient way, even optimising them using `memcpy`, but it increases the complexity of the compiler. Another approach is to implement a 'Mu standard library' that provides the C Standard Library functions such as `memcpy` in Mu IR form. But this proposal has yet been put to implementation.

It was possible to use the `CCALL` instruction to call the C Standard Library functions. However it wasn't mature enough and did not support raw memory access types. To translate the compiler intrinsics into `CCALL` instruction, it was necessary to navigate through the other raw memory access operations that prepare the intrinsic call, and translate them without loosing functionality. This was an unnecessarily complex task, and is thus not desirable.

The issue of raw memory access raised the need for a native interface, and prompted the Mu research team to reconsider the design of Mu. The discussion touched on the possibility of adding a raw memory reference type `ptr`, and the semantics of object pinning. This eventually led to expansion of the design of Mu to include the Mu Native Interface (MuNI) in the most recent specification. MuNI is a light-weight unsafe interface through which Mu IR programs are able communicate with native programs [**?**]. This enables the language client to call the system libraries and libraries written in other languages, thus extends the power of Mu and opens up new possibilities.

Via MuNI, these compiler intrinsics can be directly translated into standard C library function calls, thus resulting in more efficient code. This will be an exciting research to be done in the future.

## 5.4 Summary

Many essential types in LLTS have a direct counterpart, thus the translation of these types is straightforward. `Struct`s in LLTS may contain a variable sized array field, in which case they are to be translated into `hybrid` type. Apart from the essential types, there are also types such as `Address` that involves direct memory address. At the time when the research was performed, there was no suitable corresponding types in Mu. But the newly introduced Mu Native Interface in the most recent specification opens up a new possibility. This can be further explored in the future.

Initialised heap objects are represented using `Constant`s in RPython CFG. They can not be translated as constants in Mu, but as global cells. These heap objects need to be initialised in the heap and stored in the global cells before the start of the program. Since RPython does not generate code to initialise these objects, this pokes the question of how these heap objects are to be initialised. The current approach is to create an initialisation routine as the bundle entry point and use the the Mu IR code to initialise these objects. However, being motivated by this issue, the HAIL language has been included in the most recent specification of Mu. It is left to the work in the future to utilise this feature.

Since Mu provides built-in GC and JIT, MuPy attempts to remove these components from RPython. Thus the instructions apart from the numeric, pointer and address operations can be ignored. Some RPython compiler intrinsics in the address operations are used in the implementation of some RPython language elements such as strings and standard libraries. These used to not being able to be directly translated. The current approach is to rewrite the helper functions to imitate the functionality of these intrinsics. But this suffers from significant performance cost from interpreting Mu IR code. This prompted the research of expanding the design of Mu to include MuNI, allowing the these intrinsics to be translated into C standard library calls. The utilisation of this feature is also left to future work.

After the MuTyper stage, the core translation tasks are completed. Code generation follows straightforwardly from the structure, and is thus omitted in this thesis. The next chapter introduces the language client and launcher for executing the translated program bundles.

Chapter 6

# MuPy Client

With the Mu back-end now being integrated into the RPython translation process, program targets written in RPython can now be translated into Mu IR bundles. For the Mu IR bundle to run on Mu, a language client is required as a thin layer on top of Mu to manage the execution environment and respond to the interaction events such as traps. This chapter explores some of the aspects in developing such a language client.

Section 6.1 introduces the launcher, the part of the language client that loads the program bundle and sets up the execution environment. Section 6.2 discusses the strategies of implementing I/O, focusing especially on the print output.

## 6.1 Launcher

The task of the launcher is to load the program bundle, initialise the execution environment (Mu instance, threads, stacks, trap handlers etc.) and launch the program execution. The launcher, as part of the language client, uses the client API to interact with Mu. The artefact is currently written in Scala, since the available implementation of Mu and the client interface is also written in Scala.

The task for the launcher is straightforward, and is generally the same for other language clients. However there are some language specific aspects that need to be addressed.

### 6.1.1 Command-line arguments

One aspect that is worth addressing is the initialisation of command-line arguments.

RPython, like Python and many other languages, treats the command arguments as a list of strings. However, due to the different object layout and representations, each language may represent this list object differently. RPython expects this list of string arguments to be a pointer to a variable sized struct containing the length and an array of strings in the heap. Figure 6.1 illustrates this type in LLTS. It is straightforward to deduce the corresponding MuTS type, or by simply inspecting the program bundle.

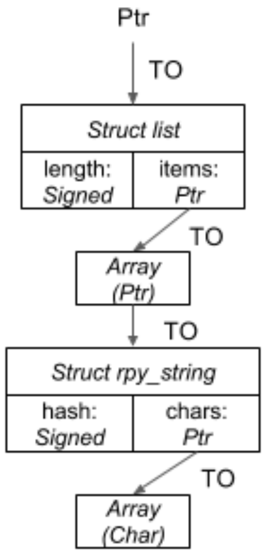


Figure 6.1: LLTS representation of the type of list of strings.

Scala takes the command-line arguments as an array of strings (`Array[String]`). It is relatively straightforward to construct the list of strings object using the client API.

### 6.1.2 Trap handler

Traps (`TRAP` and `WATCHPOINT`) pause the program execution and initiates a context switch back to the language client. A registered trap handler is used to handle these program traps. This mechanism can be used for debugging, introspection, and client-Mu interaction. The use of traps can be explored in future work, however at the moment this mechanism is mainly used to implement I/O.

## 6.2 I/O & print magic

### 6.2.1 Approaches to I/O

Implementing I/O can be a bit tricky, because the program needs to interact with the outside world.

RPython contains its own implementation of the `os` module that uses the RPython Foreign Function Interface (RFFI) allowing RPython code to call arbitrary C functions. In the `os` module, RPython redirects the `os.write` function to its own implementation, `os_write_llimpl`. This function is renamed to `ll_os.ll_os_write` in the `direct_call` operations as the callee argument. In the implementation of `rpython_print_newline` function, where the print output is to be made, RPython performs a call to `os.write` in the source code. This in turn uses the RFFI module to call the `write` system call, thus writing the buffer content to the desired file descriptor.

There are two general approaches to translate this implementation of print output. One is to do a system call from Mu IR, via MuNI and the `CCALL` instruction. Another is to use a trap to switch the context back to the language client, and let the language client handle the I/O with the operating system.

The benefit of the first approach is that it is closer to the RPython implementation, as both use some sort of native interface to perform a `write` system call. But there are also two main obstacles to this approach. One is that the adoption of this approach involves rewriting the RPython implementation of the `os` module (or providing an alternate version) that can be translated using MuNI. This could involve a lot of work. Another is that when the issue was first encountered, MuNI has not yet been put forward, and the `CCALL` instruction has not yet matured. But now as MuNI had been included in the recent specification, this path is open to be explored, as part of the future work.

The latter approach is slightly easier, and is the current adopted approach.

### 6.2.2 `TRAP` & print Magic

To implement the print output using the second approach, MuPy uses a 'print magic'.

Essentially, the goal is to insert a `TRAP` instruction when the print function is called. Simply removing the `os.write` call from `rpython_print_newline` and inserting an instruction to the function graph during the Mu back-end processing stages is a possible approach. However it suffers greatly when the inlining RPython back-end optimisation is turned on. Due to the short function body, the `rpython_print_newline` function graph is always inlined. Thus when the Mu back-end receives the control flow graphs as input, it is impossible to tell where to insert the trap instruction.

Alternatively MuPy can employ a 'magic trick'. Instead of removing the `os.write` call in the source code, during the graph transformation stage, Mu searches the graphs to find this call operation (*i.e.* `direct_call` operation with `ll_os.ll_os_write` being its callee), and replaces the name of the operation with `mu_magic_trap` and the callee with the magic string `"mu_printline"` [1]. This special magic operation can then be caught in the MuTyper stage, and directly translated to a call to the constructed function `__mu_printline__` that contains only a `TRAP` and a `RETVOID` instruction.

In the launcher, MuPy defines a trap handler that checks this trap instruction. When found, it will grab the string that is kept alive on the stack, retrieves the information and reconstruct string, and print it out using the JVM print method call.

The reason to convert the print magic operation into the call to `__mu_printline__` function instead of directly translating into a `TRAP` instruction is to unify the trap site. The trap handler in the launcher essentially matches the globally unique ID of the trap instruction. This global unique ID is made up of the function ID and the ID of the result variable. If inlining is turned on and the print trap is scattered in multiple

[1]The reason for changing not only the callee, but also the name of the operation to `mu_magic_trap` is primarily to accommodate for the graph chopping algorithm. In this way, when the chopping algorithm searches for `direct_call` operations, this operation is skipped.

places, it is difficult to correctly identify the these print traps from traps for other purposes.

## 6.3 Milestones achieved

The RPython client is now able to successfully translate many essential features of RPython, such as integer and floating point arithmetic, strings, classes and instances, exception handling and print output. The RPython adapted version of GC Benchmark [Ellis et al. ] (see Appendix A) that combines some of these elements has been successfully translated as a milestone. Thus Mu now has a complete working system that can run basic RPython programs.

## 6.4 Summary

The language client, in this case the MuPy client, is a thin layer between Mu and the RPython language. It contains a launcher that loads the MuPy bundle, constructs a RPython compatible representation of the list of strings for the command line arguments, and launches the bundle program based on the predefined bundle entry point function. It also contains trap handlers to respond to the I/O events.

Currently MuPy implements print output using the `TRAP` instruction, and lets the language client perform the printing. However, with the help of MuNI introduced in the recent specification, it is possible to translate the print output using a `write` system call. This is to be explored in the future.

Thus it has reached the end of the translation process. The next chapter outlines future work that can be done and concludes the thesis.

Chapter 7

# Conclusion

## 7.1 Future Work

### 7.1.1 Adopting the recent Mu specification

As presented in the previous chapters, the issues encountered during the development of MuPy have motivated Mu to extend its design with additional features such as HAIL and MuNI. These additional features open up new ways of translating the elements in RPython. Thus part of future development is to incorporate these into the MuPy translation process.

A part of the work would be to incorporate MuNI into existing RPython framework. This potentially involves rewriting the RFFI module, or developing an alternative MuNI module for RPython. All other modules that use RFFI will also need to be rewritten. This could cost much time and energy. More research needs to be conducted to investigate the RFFI module and then develop an appropriate strategy.

### 7.1.2 Language interpreters

The implementation of many languages have been made using RPython. These implementations are important goals for the MuPy project. The first goal would be to translate the RPython interpreter for SOM onto Mu. Then the other language implementations such as Pycket, RPython Haskell etc. are the next targets.

The RPython standard library contains modules that are useful for developing language interpreters. Thus an important goal before the successful translation of language interpreters, would be to successfully translate the standard library. This subgoal however, is still built on important language features, thus making the research into utilising the MuNI is of top priority.

### 7.1.3 Type specialisation from annotation

The current approach of type translation in MuPy depends heavily on the LLTS type specialisation by RTyper. Though most essential types and their operations can be translated, there are still imperfections such as the address types and operations, redundant GC and JIT operations, etc. Since LLTS is C-like, it is unaware of the different types in MuTS, and thus can not fully utilise them.

The possibility of specialising the types directly from the type annotations was considered. But the potential work that is involved could be huge, since the task would essentially involve building another type system (MuTS) apart from LLTS.

Thus more research needs to be done to evaluate the cost and benefit of such approach.

### 7.1.4 Extending the RPython language

RPython is a compilation framework. The current C backend limits some of the features and scopes of RPython. Mu offers many useful features such as threading, stacks, OSR, and possibly multiple return values in the future. Thus it is possible to extend the RPython language to utilise these features better.

One direction in which all this research is heading is to make MuPy a thin layer above Mu IR, combining both the high level abstraction offered by RPython and the powerful features offered by Mu. MuPy and Mu can thus become a favoured language and platform to develop language implementations.

## 7.2 Conclusion

To test the claims and concept of Micro Virtual Machines, a back-end for the RPython compilation framework targeting the Mu Micro Virtual Machine and a language client have been developed. This back-end is able to translate many essential features of RPython, such as integer and floating point arithmetic, strings, classes, exception handling etc., and has successfully translated the GC benchmark.

The translation process of Mu back-end transforms the RPython CFG representation of the program in conformity of the Mu program structure, and also utilises the exception catching mechanism offered by Mu to specialise the RPython exception handling feature. MuTyper translates the type of variables in LLTS to MuTS, along with the operations. Through these processes, the Mu back-end bridges the differences between the RPython intermediate representation and Mu IR.

There were many issues encountered throughout the development of the MuPy project, including the heap object initialisation, raw memory access types and instructions, among others. Some of these expose the imperfections of the design of Mu and have served as great contributions to the purpose of testing the platform. Thus these issues have motivated Mu to extend its capabilities and include multiple features that provide useful support for its language clients.

There are still many milestones ahead for MuPy motivated by the growing support from Mu. As the research of MuPy and Mu goes forward together, they will open up many new exciting possibilities for the implementation of programming languages.

In conclusion, through the development of the RPython language client for Mu, I have tested the design and claim of Mu and found it as a promising and desirable platform to support modern managed languages.

# Appendices

Appendix A

# GC Benchmark code in RPython

```
"""
RPython version of GC Benchmark.

The original benchmark was written by John Ellis and Pete Kovac of Post
    Communications.
It was then heavily modified by Hans Boehm, then at SGI.
"""


class Node:
    def __init__(self, l=None, r=None):
        self.left = l
        self.right = r


TREE_DEPTH_STRETCH = 18
TREE_DEPTH_LONGLIVED = 16
TREE_DEPTH_MIN = 4
TREE_DEPTH_MAX = 16
ARRAY_SIZE = 500000


def tree_size(i):
    return (1 << (i + 1)) - 1


def num_iters(i):
    return 2 * tree_size(TREE_DEPTH_STRETCH) / tree_size(i)


def populate(depth, node):
    if depth > 0:
        depth -= 1
        node.left = Node()
        node.right = Node()
        populate(depth, node.left)
        populate(depth, node.right)


def make_tree(depth):
    if depth <= 0:
        return Node()
    else:
        return Node(make_tree(depth - 1), make_tree(depth - 1))
```

```
def gcbench(argv):
    print "Garbage Collector Test"
    print " Stretching memory with a binary tree of depth", TREE_DEPTH_STRETCH

    # Strech the memory space quickly
    temp_tree = make_tree(TREE_DEPTH_STRETCH)
    temp_tree = None

    # Create a long lived object
    print " Creating a long-lived binary tree of depth", TREE_DEPTH_LONGLIVED
    ll_tree = Node()
    populate(TREE_DEPTH_LONGLIVED, ll_tree)

    # Create long-lived array, filling half of it
    print " Creating a long-lived array of", ARRAY_SIZE, "floating point numbers"
    array = [0.0] * ARRAY_SIZE
    for i in range(1, ARRAY_SIZE / 2):
        array[i] = 1.0 / i

    return 0


def target(*args):
    return gcbench, None
```

# Bibliography

ANANIAN, C. S. AND RINARD, M. 1999. Static single information form. Technical report, MASTER'S THESIS, MASSACHUSSETS INSTITUTE OF TECHNOLOGY. (p. 14)

BAUMAN, S., BOLZ, C. F., HIRSCHFELD, R., KIRILICHEV, V., PAPE, T., SIEK, J. G., AND TOBIN-HOCHSTADT, S. 2015. Pycket: A tracing jit for a functional language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015 (New York, NY, USA, 2015), pp. 22–34. ACM. (p. 8)

BEHRENS, S. 2008. Concurrency and python. (p. 5)

BOLZ, C. F. AND TRATT, L. 2015. The impact of meta-tracing on vm design and implementation. *SCICO*, 408–421. (p. 8)

CASTANOS, J., EDELSOHN, D., ISHIZAKI, K., NAGPURKAR, P., NAKATANI, T., OGASAWARA, T., AND WU, P. 2012. On the benefits and pitfalls of extending a statically typed language jit compiler for dynamic scripting languages. *SIGPLAN Not. 47*, 10 (Oct.), 195–212. (pp. 1, 6)

ELLIS, J., KOVAC, P., AND BOEHM, H. An artificial garbage collection benchmark. `http://hboehm.info/gc/gc_bench.html`. (pp. 3, 44)

Emscripten. Emscripten documentation. `http://kripken.github.io/emscripten-site/`. (p. 8)

HAGER, S. Implementing the r language using rpython. Master's thesis, Heinrich Heine University Düsseldorf. (p. 8)

HOMESCU, A. AND ŞUHAN, A. 2011. Happyjit: A tracing jit compiler for php. *SIGPLAN Not. 47*, 2 (Oct.), 25–36. (p. 8)

HUANG, R., MASUHARA, H., AND AOTANI, T. Pyrlang: A high performance erlang virtual machine based on rpython. `http://2015.splashcon.org/event/splash2015-posters-pyrlang-a-high-performance-erlang-virtual-machine-basec`
(p. 8)

JUNEAU, J., BAKER, J., NG, V., SOTO, L., AND WIERZBICKI, F. 2010. The definitive guide to jython. `http://www.jython.org/jythonbook/en/1.0/`. (p. 6)

LATTNER, C. AND ADVE, V. 2004. The LLVM Compiler Framework and Infrastructure Tutorial. In *LCPC'04 Mini Workshop on Compiler Research Infrastructures* (West Lafayette, Indiana, Sep 2004). (p. 6)

MARLOW, S. AND PEYTON-JONES, S. The glasgow haskell compiler. `http://www.aosabook.org/en/ghc.html`. (p. 5)

MARR, S. AND DUCASSE, S. 2015. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '15 (2015), pp. 821–839. (p. 8)

Micro VM. The micro virtual machine project. `http://microvm.github.io/`. (p. 2)

Mu Spec. Mu micro virtual machine online specification. `https://github.com/microvm/microvm-spec/wiki`. (pp. 4, 7, 30, 35, 37)

PyPy.js. Pypy.js: First steps. `https://www.rfk.id.au/blog/entry/pypy-js-first-steps/`.

RIGO, A. AND PEDRONI, S. 2006. 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 (New York, NY, USA, 2006), pp. 944–953. ACM. (p. 8)

RPython Doc. (pp. 6, 8, 11)

THOMASSEN, E. W. AND HETLAND, M. L. Trace-based just-in-time compiler for haskell with rpython. (p. 8)

WANG, K., LIN, Y., BLACKBURN, S. M., NORRISH, M., AND HOSKING, T. A. 2015. Draining the swamp: Micro virtual machines as a solid foundation for language development. In THOMAS BALL AND RASTISLAV BODIK AND SHRIRAM KRISHNAMURTHI AND BENJAMIN S. LERNER AND GREG MORRISETT Ed., *SNAPL* (Asilomar, California, may 2015), pp. 321–336. LIPICS. (pp. 1, 5, 6)

ZALEWSKI, S. A javascript interpreter in rpython. Master's thesis, Heinrich Heine University Düsseldorf. (p. 8)