Micro Virtual Machines: A Solid Foundation for Managed Language Implementation

Kunshan Wang

A thesis submitted for the degree of
Doctor of Philosophy
The Australian National University

September 2018

Except where otherwise indicated, this thesis is my own original work.

王坤山

Kunshan Wang

13 September 2018

to my parents and my grandparents

Acknowledgments

I would like to express my gratitude to the individuals and organisations that generously supported me during the course of my PhD.

First and foremost, I would like to thank my supervisor Prof. Steve Blackburn, and my advisors Prof. Antony Hosking and Dr. Michael Norrish, who supported me with their foresight, expertise, experience, guidance and patience. This thesis would not be possible without their continued support.

I would like to thank the Chinese government, the Australian National University and Data61 (formerly NICTA), who supported me financially. I would also like to thank my master supervisor Prof. Zhendong Niu, who guided me during my study in China and supported my study overseas.

I would like to thank my colleagues. Thank you Vivek Kumar, Ting Cao, Rifat Shahriyah, Xi Yang, Tiejun Gao, Yi Lin, Luke Angove, Javad Ebrahimian Amiri, and all other members of the Computer Systems Research Group. Your talent and friendliness made my study in the ANU enjoyable. Special thanks to Yin Yan, a visiting student, who accompanied me and restored my hope when I was overwhelmed by despair.

I would like to thank John Zhang, Nathan Young, Andrew Hall, who made contributions to the Mu project as Honours students or Summer scholars. It is my honour to work with you.

I would like to thank Prof. Eliot Moss at the University of Massachusetts, and all researchers who collaborated in the development of the Mu project. Your feedback and suggestions helped us make greater progress.

Finally I would like to thank my parents who supported me selflessly in my life and my study.

Abstract

Today new programming languages proliferate, but many of them suffer from poor performance and inscrutable semantics. We assert that the root of many of the performance and semantic problems of today's languages is that language implementation is extremely difficult. This thesis addresses the fundamental challenges of efficiently developing high-level managed languages.

Modern high-level languages provide abstractions over execution, memory management and concurrency. It requires enormous intellectual capability and engineering effort to properly manage these concerns. Lacking such resources, developers usually choose naïve implementation approaches in the early stages of language design, a strategy which too often has long-term consequences, hindering the future development of the language. Existing language development platforms have failed to provide the right level of abstraction, and forced implementers to reinvent low-level mechanisms in order to obtain performance.

My thesis is that the introduction of micro virtual machines will allow the development of higher-quality, high-performance managed languages.

The first contribution of this thesis is the design of Mu, with the specification of Mu as the main outcome. Mu is the first micro virtual machine, a robust, performant, and light-weight abstraction over just three concerns: execution, concurrency and garbage collection. Such a foundation attacks three of the most fundamental and challenging issues that face existing language designs and implementations, leaving the language implementers free to focus on the higher levels of their language design.

The second contribution is an in-depth analysis of on-stack replacement and its efficient implementation. This low-level mechanism underpins run-time feedback-directed optimisation, which is key to the efficient implementation of dynamic languages.

The third contribution is demonstrating the viability of Mu through RPython, a real-world non-trivial language implementation. We also did some preliminary research of GHC as a Mu client.

We have created the Mu specification and its reference implementation, both of which are open-source. We show that that Mu's on-stack replacement API can gracefully support dynamic languages such as JavaScript, and it is implementable on concrete hardware. Our RPython client has been able to translate and execute non-trivial RPython programs, and can run the RPySOM interpreter and the core of the PyPy interpreter.

With micro virtual machines providing a low-level substrate, language developers now have the option to build their next language on a micro virtual machine. We believe that the quality of programming languages will be improved as a result.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Statement

Micro virtual machines will allow managed languages to be developed more easily and with higher quality.

1.2 Problem Statement

Today's programming language landscape is littered with inefficient, hard-to-use but otherwise important languages. Despite their active user community and their application in critical settings, their performance overheads in many cases are best measured on a log scale, and their inscrutable semantics make software maintenance a nightmare.

Many of the performance and semantic issues that befall such languages can be traced to fundamental implementation challenges. PHP's confounding copy-on-write semantics [Tozawa et al., 2009] can be traced back to a bug report dating to 2002 [PHP, 2002], when the behaviour was first observed by a user. However, realising the challenges in fixing the semantics, the PHP developers chose not to fix the bug, but instead declared the semantics a language feature, and it has remained so to this day. The engineering challenge of implementing a garbage collector has led many languages to depend on naïve reference counting in their earliest implementations despite its well-known performance limitations and inability to collect cycles, as observed by Jibaja et al. [2011]. Similarly, the intellectual challenges in correctly implementing concurrency have led many programming languages to have weak, broken or absent models of concurrency; CPython's[1] use of the infamous global interpreter lock (GIL) [Behrens, 2008] is one such example.

The few high-performance language implementations that successfully fought through the challenges are invariably the result of non-trivial investments. It took twenty years for the Java Virtual Machine (JVM) to reach the maturity it has today, with heavy support from Sun, Oracle, and many researchers from academia.

[1] CPython is the official implementation of Python (https://www.python.org/). The source code is hosted at https://github.com/python/cpython. The letter 'C' indicates that it is implemented in C, and distinguishes it from other Python implementations, such as Jython, IronPython and PyPy.

Another example is the PyPy project [Rigo and Pedroni, 2006], which derives performance from advanced specialising meta-tracing technology. A series of successful publications are tribute to the substantial intellectual input underpinning PyPy's performance. Other VM projects, such as SpiderMonkey, V8, JavaScriptCore and HHVM, are the results of substantial investment from large organisations, such as Mozilla, Google, Apple and Facebook, each of which could justify the investment because of its impact on their bottom line.

Despite these challenges, the demand for high-quality languages remains high, while the complexity of systems increases, and new languages emerge to address the needs that arise in specialised domains. Many languages, such as R, are developed by domain experts rather than computer scientists. However, most development teams do not have access to the same intellectual resources as large companies, thus cannot afford building their next language implementation from scratch while still guaranteeing high quality.

This leads to a question: *Is there a platform for language developers where existing languages can be built with higher quality, and new languages can be designed and implemented properly?*

Existing Platforms Building a language on a well-established platform simplifies development and saves engineering effort.

LLVM is exactly this kind of platform. Created by Lattner and Adve [2004], LLVM is a popular compiler framework which has many built-in optimisers, supports many targets and provides a JIT compiler. However, LLVM was designed for non-garbage-collected languages, such as C, C++ and Objective-C. Consequently, LLVM does not provide any readily-usable garbage collectors for its clients, and adding GC support is difficult despite LLVM providing mechanisms intended to support GC. In addition, some concurrency primitives, such as the creation of threads, are still platform-dependent. An alternative to using LLVM as a language foundation is generating C source code directly. However, this approach also suffers from the lack of GC support.

The JVM is one of the most sophisticated VM platforms in the world, with high-performance implementations such as HotSpot and IBM's J9. Many high-level languages, such as Python, Ruby, Scala, Clojure etc., have been built on top of the JVM. However, because the JVM is specifically tailored to Java, it carries substantial dependencies, including object-oriented programming primitives and the comprehensive Java standard library, which may not be useful for other languages. Because it is designed for Java, there is generally a semantic gap between the JVM and other high-level language it might be used to support. Its JIT compiler is designed for Java, a static language, therefore dynamic languages will not run optimally unless specialisation is also supported, as shown from the experience of Castanos et al. [2012].

We identify concurrency, execution and garbage collection as the three major concerns that contribute to the complexities of language development.[2] Thus, we propose the concept of a micro virtual machine which only provides abstractions over these three concerns, but otherwise keeps the abstraction level as low as possible, so as to minimise unnecessary dependencies and semantic gaps. Micro virtual machines will serve as an ideal language-neutral platform for supporting managed languages.

1.3 Scope and Contributions

The aim of this thesis is to design a concrete micro virtual machine — Mu.

We provide a detailed description of the design of the Mu micro virtual machine, comparing and contrasting Mu with existing work. We show the principles that underpin our design decisions, and discuss the pros and cons of existing work and alternative designs. We describe the on-stack replacement API of Mu, and its implementation. We demonstrate the real-world applicability of Mu by using Mu as a backend for RPython.

This work serves as a *proof of existence* and a *proof of concept*, but not a proof of performance.[3] Because the micro virtual machine is a new concept without established prior work, designing a consistent system that involves all of the three major concerns is very difficult, and developing a high-performance implementation will be even more difficult.

Therefore, an explicit *non-goal* of this thesis is the efficient implementation of the Mu micro virtual machine itself, including instruction selection, register allocation, and the implementation of the garbage collector. The Mu micro virtual machine project is a joint effort of the Computer Systems Research Group at the ANU, where the focus of my work is the high-level VM design, while the efficient implementation of the Mu micro virtual machine is the concern of others. However, for every feature in the design, we will give reasons to show that it can be implemented efficiently.

Mu Design In Part I, we discuss the design of Mu. Mu is defined as a specification that allows multiple compliant implementations. We will discuss the design process which lead to the specification, including various issues we took into consideration, and compromises we made. The specification defines the Mu intermediate representation (IR) and its API. We use LLVM as the frame of reference for the IR design. As a micro virtual machine, it has a low-level type system and instruction set similar to that of LLVM. However, as an abstraction over memory, Mu also features GC-traced

[2]Geoffray et al. [2010] already identified concurrency, JIT compilation and garbage collection as the three major concerns when developing VMKit. We generalise JIT compilation into execution in general, because the cross-cutting concerns involving execution is not specific to JIT compilation, and our experience from the RPython client (see Section 9.1.4) shows that ahead-of-time compilation is also important. It is also noteworthy that VMKit's approach is to combine three existing software components, but our goal is to design and implement a minimalist platform that supports these concerns from first principles. See Section 3.2.3 for more details.

[3]Details of these terms are discussed in Page 17–20 of the book *Academic Careers for Experimental Computer Scientists and Engineers* authored by the National Research Council [1994].

object reference types and has heap object allocation as a primitive operation. To address run-time feedback-based optimisation that is required to efficiently implement dynamic languages, Mu provides APIs for trap handling, stack introspection, function redefinition and on-stack replacement, which conventional VMs seldom expose to the client. Mu provides threads as a primitive, a C++11-like memory model for synchronisation, and the SWAPSTACK operation [Dolan et al., 2013] to support coroutines and massively concurrent languages such as Erlang.

Stack Operations and OSR In Part II, we take a deeper look into the topic of on-stack replacement (OSR), a very important mechanism for JIT compiling. We introduce a versatile stack model which powers the SWAPSTACK operation, introspection and OSR. In a case study, we examine the well-known SpiderMonkey and V8 JavaScript engines, and learn that such crucial mechanisms are tricky to implement, but real-world virtual machines have to reinvent the wheel because they are developed from scratch. We demonstrate our Mu OSR API by constructing an experimental JavaScript client on the Mu reference implementation. We also show concretely how the Mu stack model can be natively implemented. System libraries already provide similar stack-related mechanisms, such as the stack unwinding metadata which supports C++ exception handling. However, we show that such mechanisms are insufficient and not portable, and this provides evidence supporting our thesis that a well-defined substrate such as Mu is much needed.

Supporting Real-world Languages Finally, in Part III, we show that Mu can support real-world non-trivial languages. We use RPython [Rigo and Pedroni, 2006] as a Mu client. By supporting this client, Mu will support all languages built upon RPython, including PyPy, RPySOM [Marr et al., 2014] and Pycket [Baumann et al., 2015]. RPython performs type inference to transform Python functions into a static language in the form of control flow graphs (CFG) similar to the level of the Mu IR. In the original RPython backend, it then injects exception handling and GC into the functions and generates C source code which the build system then compiles into an executable. But we replace the default C backend with a similar Mu IR backend. This work led to several refinements in the Mu design, including the concrete design of boot images and external linkage, addressing the challenges encountered in supporting RPython and, more generally, ahead-of-time compiling. We are able to run RPySOM and the core of the PyPy interpreter on the reference implementation. We also did some preliminary work on GHC. However, limited by time, we have not developed runnable Haskell clients, yet.

1.4 Thesis Outline

The body of this thesis is structured around the three contributions outlined above. Chapter 2 describes the difficulties in language implementation; Chapter 3 discusses related work that attempt to facilitate language development.

Part I, II and III comprise the main body of the thesis, covering the three key contributions. Part I provides a detailed description of the design of the Mu abstract machine, and its underpinning principles. Part II takes a deeper look into on-stack replacement and its implementation. Part III introduces our RPython client and our preliminary work on GHC, and shows Mu's real-world applicability.

Finally, Part IV concludes the thesis, summarising our contributions, and predicting the future when micro virtual machines are used as a standard tool for language implementation.

Chapter 2

Background

This chapter provides background information about the difficulties in managed language development. Section 2.1 provides an overview of the issue; Section 2.2 discusses the three major concerns that our micro virtual machine addresses, namely just-in-time (JIT) compilation, concurrency and garbage collection, and their cross-cutting concerns; Section 2.3 discusses the consequences of not properly handling these concerns, and summarises this chapter.

2.1 Difficulties in Managed Language Implementation

A large fraction of today's software is written in managed languages. The term 'managed languages' originates from the .NET Framework [Microsoft]. There is no precise definition, but such languages can be characterised as high-level languages that run on managed runtimes (i.e. virtual machines) which provide various abstractions, notably automatic memory management (i.e. garbage collection). Many modern languages, including Java, C#, Python, Ruby, JavaScript, PHP, etc., belong to this category. Some languages or implementations, such as RPython, GHC, and Go, are implemented by ahead-of-time (AoT) compilation and do not have any explicit virtual machines, but we still take these languages into consideration because of their use of garbage collection. Managed languages are economically important. For example, Facebook depends on servers running PHP for its core business of efficiently delivering hundreds of billions of page views a month, while Google depends on Java for its Android apps, and uses JavaScript to power its most widely used web applications including search and Gmail.

However, implementing such languages properly is difficult. As Geoffray et al. [2010] already identified in the VMKit project, JIT compilation, concurrency and garbage collection are the three major concerns that contribute to complexity. Each are technical minefields in their own right but when brought together in a language runtime, their respective complexities combine in very challenging ways. Each of these concerns is rich enough to support a well-developed research sub-community and rich literature of its own.

2.2 Three Major Concerns

In this section, we will take a deeper look at each of these concerns, as we must understand the challenges concretely in order to address them appropriately.

2.2.1 Just-in-time Compilation

In theory, all programming languages can be implemented by either compiling or interpreting. By traditional ahead-of-time (AoT) compiling, programs written in a high-level language are translated into executable machine code before execution, and the machine code is executed directly on the processor. Alternatively, by interpreting, an interpreter reads and decodes the program source code or byte code at run time, and executes on behalf of the program. Interpreting makes the program portable to any supported platforms without recompilation, but the 'interpretation loop' is a major performance bottleneck because the interpreter must decode every abstract syntax tree node or bytecode instruction before carrying out the corresponding operation.

Just-in-time (JIT) compilation is another approach to language implementation. The compiler is placed on the user's machine instead of the developer's, and the program is compiled into machine code on demand at run time. This gives the performance advantage of compilation while still allowing the program to be distributed in a machine-independent form. Some virtual machines, such as the HotSpot JVM, use a hybrid approach that supports both interpretation and JIT compilation.

However, JIT compilers are difficult to develop. Like ahead-of-time compilers, constructing JIT compilers requires deep knowledge of the machine architecture of each platform to be supported, while interpreters themselves can be simply developed in other high-level languages that are already portable. A JIT compiler must also take the overhead of compilation into consideration, as the overhead will be applied to the execution of the program, while ahead-of-time compilers are relatively unconstrained.

A unique challenge to JIT compilation is run-time *feedback-directed optimisation*. For responsiveness, language runtimes initially execute the program using interpreters or baseline JIT compilers, which have fast compilation speed, but generate suboptimal code. During initial execution, the language runtime detects frequently executed (hot) code, usually hot functions or hot loops, using timer-based (as in JikesRVM [Alpern et al., 2009]) or counter-based (as in SpiderMonkey [Mozilla] and V8 [Google]) profiling mechanisms. Various optimisations in the JIT compiler are then applied to the hot code only. Thus the language runtime needs to properly handle profiling and the transition between the user programs and the JIT compiler, which is not necessary for AoT compilers.

One important run-time optimisation is *specialisation*. This optimisation is crucial for dynamic languages because the main performance overhead of naïvely implemented dynamic languages is type checking values before each operation (such as adding, string concatenation, etc.), as revealed by the work of Castanos et al. [2012]. The specialiser performs type inference, and speculatively specialises the variable types into the actual value types observed at run time. For example, if a variable

```
1  THRESHOLD = 1000
2
3  def sum_between(low, high):
4      s = 0
5      i = low
6
7      hotness = 0
8      while i <= high:
9          s += i
10
11         hotness += 1
12         if hotness >= THRESHOLD:
13             optimise()
14
15     return s
```

Figure 2.1: Optimisation of a hot loop. Lines 11–13 emulate the counter-based profiling mechanism generated by the compiler to detect hot loops. Assume `low = 1` and `high = 1000000`. Since `low` and `high` are sufficiently far apart, optimisation will be triggered after the 1000th iteration. The optimiser may generate a newer version of the function `sum_between` which will be executed the next time it is called. However, after optimisation, there are still 999000 additional iterations of loops in the *unoptimised* version, which could be orders of magnitude slower than the optimised version. The runtime cannot simply restart the function, because the function may have other side effects that have already taken place. The correct solution is to replace the execution context (the stack frame) of the current `sum_between` activation with a new context (stack frame) for the optimised version, and let the program continue at the equivalent point of execution in the optimised version. In this way, the transition from the unoptimised version to the optimised version will take place immediately after optimisation. This mechanism is called 'on-stack replacement' (OSR).

is observed to always hold integers, then all operations on it will be specialised for integers without checking in the specialised version of the compiled code. This elides most of the type checking that slows down the execution, and greatly speeds up the execution of dynamic languages.

On-stack replacement (OSR) is an important low-level mechanism that supports feedback-directed optimisation. Consider the example in Figure 2.1. When optimisation happens at a loop inside a function, we want the optimised code to take effect immediately rather than having to wait for the unoptimised loop to end, which will be extremely slow. Initially developed for the Self VM [Hölzle et al., 1992; Hölzle and Ungar, 1994], the OSR mechanism replaces stack frames on the stack in order to transition the execution context to the optimised version. OSR can also be used during de-optimisation (invalidating speculatively specialised code) or during online code modification (usually used for debugging). However, OSR is very difficult to implement because it requires clear knowledge about the stack layout to decode stack frames at run time. This thesis dedicates Chapter 7 to discuss this topic in greater detail.

In summary, JIT compilation is not merely the emission of machine code at run time. It involves many advanced language implementation technique that work together to support run time optimisation.

2.2.2 Concurrency

As processor clock frequency stopped increasing exponentially, hardware vendors instead resorted to parallel hardware, such as multi-core processors, to improve performance. Such hardware potential cannot be fully utilised unless software is also designed with parallelism in mind. Meanwhile, concurrent programming models, such as multi-threading and the actor model, provide intuitive tools for the programmers to solve certain problems.

However, with the advent of parallel hardware, certain properties, such as sequential consistency, can no longer be taken for granted. *Sequential consistency* is the property that the result of the execution of concurrent threads is equivalent to an execution where all operations from all threads are executed in a specific sequential order. Unconditionally enforcing sequential consistency will significantly degrade performance. Counter-intuitive bugs manifest on parallel hardware. Memory operations may appear to be out of order, and may observe values that seem to come out of thin air.

For example, on architectures where the memory is read and written at 32-bit granularity, writing a 64-bit value may take two separate 32-bit store operations. Another thread may see an out-of-thin-air value with 32 bits being old and the other 32 bits being new. This phenomenon is known as *word tearing*.

Figure 2.2 shows another example where the execution of a multi-threaded program may not be equivalent to any sequentially consistent execution. This can be caused by either the reordering of memory accesses inside the processor, or optimisations performed by the compiler, or both.

Compiler transformations which were valid on single-threaded programs become erroneous in the face of concurrency. These phenomena force us to think carefully about support for concurrency at the programming language level, since pure library approaches cannot guarantee correctness of the resulting code, as pointed out by Boehm [2005].

The Java 1.5 memory model [Manson et al., 2005; Gosling et al., 2014] pioneered the attempt to specify the legal behaviours of multi-threaded programs, and the C++11 memory model [Boehm and Adve, 2008; ISO, 2012] continued such efforts for the C++ programming language. Designing a memory model is basically an attempt to achieve a compromise between the programmers, who desire certain guarantees in the programming model, and the hardware designers, who need to implement the parallel processors efficiently. The C++11 memory model took the approach of 'weak ordering' where certain atomic memory operations are explicitly labelled as synchronisation operations, which help guarantee the visibility between read and write operations across different threads, and where data-race-free programs are guaranteed to be sequentially consistent. This approach allows compilers to translate synchronisation operations into adequate machine instruction sequences, such as inserting memory fences before or after operations, while not having to

```java
int x = 0;
int y = 0;

void thread1() {
    x = 1;
    y = 2;
}

void thread2() {
    int yy = y;
    int xx = x;
    if (yy == 2) {
        assert(xx = 1); // may fail
    }
}
```

(a) Java Code

Execution	x=1 y=2 yy=y xx=x	x=1 yy=y y=2 xx=x	yy=y x=1 y=2 xx=x	x=1 yy=y xx=x y=2	yy=y x=1 xx=x y=2	yy=y xx=x x=1 y=2
Value of xx	1	1	1	1	1	0
Value of yy	2	0	0	0	0	0

(b) All Sequentially Consistent Executions and Their Results

Figure 2.2: Example of concurrent reads and writes without ordering. In the Java program in Figure 2.2(a), `x` and `y` are shared variables with initial values of 0. Thread 1 writes to `x` and then `y`, and thread 2 reads from the two variables in the opposite order. There is a chance that thread 2 may observe `yy == 2` and `xx == 0`. Figure 2.2(b) enumerated all possible sequentially consistent executions, and none of them may have `xx==0 && yy==2` as their results. This phenomenon can be caused in two places. Firstly, the CPU hardware may consider `x` and `y` as two independent memory locations, and reorder the two reads or the two writes to maximise memory throughput. Secondly, the compiler may also optimise the code, and emit machine code which accesses the memory in a different order from the program. In Java 1.5 or later, we can prevent this reordering by annotating the variable `y` with the keyword `volatile`, i.e. `volatile int y = 0;`. Reads and writes of volatile variables behave like implicit *memory fences* which forbid certain reordering of memory operations.

treat every memory operation as a synchronisation operation. This makes C++11 efficiently implementable on both strongly ordered architectures such as x86, as well as weakly ordered architectures such as ARM, POWER and Alpha. Newer programming languages, such as Go, also have their own memory models that define the semantics of multi-threaded programs.

Higher-level synchronisation mechanisms, such as mutex locks, condition variables, semaphores and message queues, can be implemented on top of atomic memory operations, such as compare-and-swap (CAS)[1], provided by the memory model. On modern systems such as GNU/Linux, these mechanisms are mostly implemented in user space for efficiency, with minimal assistance from the operating system kernel, such as the Futex system call.

The proper implementation of a memory model needs support from the JIT compiler, because the JIT compiler understands the instruction set of the platform, and the guarantees the machine instructions provide. For example, on x86, the plain `MOV` instruction has the ACQUIRE semantics when loading from memory, and has the RELEASE semantics when storing to memory; on AArch64, ordinary load and store instructions do not have such guarantees, but there are special instructions such as `LDAR` and `STLR` that provide the ACQUIRE and the RELEASE semantics, respectively. The JIT compiler is responsible for generating the correct instruction sequences to support the intended memory order on the concrete platform.

2.2.3 Garbage Collection

Garbage collection (GC) was introduced in LISP [McCarthy, 1960], and has become an integral part of modern programming languages. It is a manifestation of the principle of 'separation of concerns' in the way that it frees the programmers from having to manually deal with memory deallocation of every object they create.

Garbage collection consists of three key facets: a) object allocation, b) garbage identification, and c) garbage reclamation, as described by Blackburn and McKinley [2008]. Different garbage collection algorithms employ different approaches to handle each facet.

High-performance garbage collection algorithms are difficult to implement. Efficient garbage collectors must be co-designed with the compiler. For example, a *stack map* is a data structure to identify object references in stacks, and must be generated by the compiler, which knows the layout of stack frames. Stack maps are an integral part of exact GC, which can identify all object references in the entire virtual machine. Copying collectors, which bestow good locality and offer cheap *en masse* reclamation, depend on exact GC [Jibaja et al., 2011]. Another example, *write barriers* are code generated by the compiler around every write operation of reference fields. Generational collectors, which effectively manage high-mortality young objects, require write barriers to record references from the old generation to the nursery [Jibaja et al., 2011]. All of these advanced techniques are hard to implement, but state-of-the-art

[1] Also known as compare-exchange.

GC algorithms, such as Immix [Blackburn and McKinley, 2008], require many such techniques.

Efficient garbage collectors must also be co-designed with concurrency. If the garbage collector desires to make use of the parallel hardware resources, itself needs to implement parallel and concurrent scanning and collection algorithm, which will interact with the memory model. And the JIT compiler must also help the GC insert GC-safe points (yieldpoints) in the JIT-compiled machine code to perform handshakes between application threads and GC threads. Lin et al. [2015] provide an in-depth analysis of the challenges in the efficient implementation of yieldpoints.

Due to the difficulties in garbage collection implementation, language implementers often choose naïve garbage collection strategies in their initial VM design. Some implementations, such as Lua [Ierusalimschy et al., 1996], use naïve non-generational mark-sweep GC which does not perform as well as its generational counterpart; some implementations, such as CPython, use naïve reference counting[2] which performs worse than mark-sweep by 30% or more, as measured by Shahriyar et al. [2012]; others simply use conservative collectors, such as the off-the-shelf Boehm-Demers-Weiser garbage collector [Boehm and Weiser, 1988], which cannot support copying collectors. Using naïve GC algorithms degrades overall performance. What is worse, as Jibaja et al. [2011] pointed out in their paper, 'retrofitting support for high-performance collectors is typically very hard, if not impossible'. This is because fixing the collector alone is not enough. It is more important and more difficult to fix the compiler which is supposed to generate stack maps and write barriers. Although some language implementations, such as Mono [Mono], successfully migrated from conservative garbage collection to exact garbage collection, others, such as PHP, are stuck with naïve reference counting because its semantics depend on reference counting. Pyston [Pyston], a JIT-compiling Python implementation, attempted to use mark-sweep GC for better performance in its early versions, switched back to naïve RC in version 0.5, because Pyston intended to maintain compatibility with existing C extension modules written for the official CPython which depends on naïve RC, and there were cases where they 'wouldn't be able to support the applications in their current form' [Kevin Modzelewski, 2016].

2.3 Consequences and Summary

As shown in the previous section, many challenging issues, such as memory model, stack maps, write barriers and yieldpoints, arise when handling more than one of the three major concerns in the same system. Language implementers who are not prepared for such complexity often choose naïve implementation strategies to get their language up and running without initially worrying about performance. As we mentioned in Chapter 1, CPython uses a slow but easy-to-implement interpreter

[2]Not all reference-counting-based GC algorithms are naïve. Shahriyar et al. [2014] developed a high-performance Reference Counting Immix (RC Immix) algorithm which outperforms the tracing-based production collector in JikesRVM, namely Generational Immix. But the complexity of RC Immix cannot be overlooked.

as its sole execution engine, which avoided the complexity of JIT compiling; it uses a global interpreter lock (GIL) [Behrens, 2008] to prevent parallel execution because the interpreter is not designed for concurrent execution; and it uses a naïve reference counting GC algorithm which further slows down performance. At the time of writing, PyPy [Bolz et al., 2009] still has GIL in its mainline code [PyPyGIL], although there is ongoing work of eliminating the GIL using software transactional memory. The official Ruby [Ruby] implementation also has a 'global VM lock' which is similar to CPython's GIL.[3] PHP's confounding copy-on-write semantics also originates from the fact that it uses naïve reference counting. These early decisions get baked into the languages themselves, and hinder their long-term development. Now, many CPython native modules assume the presence of the GIL which makes its removal even more difficult. And so GIL-free Python implementation remains a research topic. The copy-on-write semantics [Tozawa et al., 2009] of PHP persists until today as a documented 'feature'.

These cross-cutting concerns exist because of the tightly coupled nature of JIT compilation, concurrency and GC. This is why we propose 'micro virtual machines' to address these concerns in *one* carefully designed system. The details of Mu design will be discussed in Part I. In the next chapter, we will look at existing systems that address the issue of language implementation.

[3]See `vmcore.h` in the Ruby source code.
URL: https://github.com/ruby/ruby/blob/e4600b87b5a13412fc8f46da22d4f224732e6769/vm_core.h#L561

Chapter 3

Related work

In the preceding chapter, we introduced some of the difficulties in language development. This chapter discusses related work that attempt to address these difficulties, and explains why those solutions are insufficient to achieve the goal of manged language development.

This chapter is structured around two fundamental ways to implement a language. Section 3.1 discusses monolithic language implementations; Section 3.2 discusses existing multi-language platforms for language development; Section 3.3 summarises this chapter.

3.1 Monolithic Language Implementations

One way to efficiently implement a managed language may be to implement the virtual machine from scratch. Many real-world language implementations, including HHVM [Adams et al., 2014], SpiderMonkey [Mozilla], V8 [Google] and LuaJIT [Pall], were developed this way. Compared to naïve implementations, these carefully developed implementations indeed significantly improved the performance of the language.

However, this approach has several problems. One problem is the lack of code reuse. Since such virtual machines are written for one language, their core components, such as the JIT compiler, cannot be reused for other language implementations. As a notable example, both SpiderMonkey and V8 are written in C++, and both implement the JavaScript programming language using similar techniques including JIT compilation, feedback-directed optimisation, specialisation, OSR and generational GC. Yet no code is shared between these two projects. Given that programming language implementation is difficult, such an approach can only be afforded by those with sufficient expertise and engineering power, usually large companies or organisations such as Mozilla, Google and Facebook. Another problem is that such implementations still tend to be naïve compared to mature systems. For example, despite the powerful tracing JIT compiler, LuaJIT [Pall] still uses a non-generational mark-sweep collector, and does not have any language-level threading support.[1]

[1]LuaJIT does have plans to implement more sophisticated garbage collectors in the future version LuaJIT 3.0 [LuaJITGC].

3.2 Multi-language Virtual Machines and Frameworks

An alternative to the monolithic approach is to build the language on a well-established virtual machine or compilation framework. In addition to micro virtual machines which we will discuss in the next chapter, there have been several existing language-neutral platforms that aimed to support the implementation of languages.

3.2.1 The Java Virtual Machine

The Java Virtual Machine (JVM) [Lindholm et al., 2014] was originally designed for the Java programming language, but its portable Java Bytecode, clearly specified behaviors and performance attracted a wide range of language implementations to be hosted on the JVM, including Jython [Jython], JRuby [JRuby], Scala [Scala] and X10 [Ebcioğlu et al., 2004].

This approach—reusing the existing JVM for new languages—raises several fundamental problems. The obvious one is the semantic gap between the new language and Java. The JVM implements many Java-specific semantics, including the Java object layout, thread model and object-oriented programming, which are irrelevant to other dissimilar languages. The JVM JIT compiler is tailored for Java, a static language. Repurposing such a JIT compiler for dynamic languages does not automatically deliver the hoped-for performance boost, unless appropriate language-specific optimisations, such as specialisation, are also added [Castanos et al., 2012]. Jython [Jython], for example, 'almost never exceeds CPython in performance, and is generally slower', according to the measurement from Bolz and Tratt [2015], because 'features that make use of Python's run-time customisability have no efficient implementation on a JVM'. On the other hand, useful low-level mechanisms, such as vector instructions, on-stack replacement, Swapstack and tagged references, are non-existent in the JVM specification. This omission makes it difficult to efficiently support other languages. For example, values in Lua, JavaScript or other dynamic languages, could be represented as tagged references which would elide heap allocations for numerical types. It would also be expensive to map the light-weight Erlang processes to Java threads which are usually mapped to native threads, and the Swapstack mechanism [Dolan et al., 2013] could be a better match.

Because JVM is designed to support the Java platform, it comes with a comprehensive Java standard library. Such a well-implemented library can be useful in practice, but it also introduces huge dependencies for other light-weight languages (such as Python) which usually have their own standard library more suitable for the language.

We learn from these unsuccessful attempts, and carefully design Mu to be a language-agnostic platform at a much lower level to avoid the semantic gap. We assume most optimisations will be done by a client above it, and expose many low-level mechanisms to the client in order to enable many advanced implementation techniques. Mu does not have a standard library, but lets the client decide what libraries to be bundled with the runtime.

3.2.2 LLVM

The Low Level Virtual Machine (LLVM) [Lattner and Adve, 2004] is a compiler framework including a collection of modular and reusable compilation and toolchain technologies. The LLVM compiler framework and its code representation (LLVM IR) together provide a combination of key capabilities that are important for language implementations.

However, LLVM is designed for *C-like languages*. Like C, the LLVM IR type system contains raw pointer types but not reference types, and LLVM does not provide a garbage collector. Instead it provides the `@gcread`[2] and `@gcwrite` intrinsic functions which are place holders for read and write barriers which the language front needs to implement and insert. LLVM also provides several mechanisms to identify references held on the stack. This kind of approach to GC is problematic. Remember that high-performance GC is extremely difficult to implement, therefore not providing a working garbage collector already imposes a considerable burden to language developers who are not GC experts. For stack roots, the `@gcroot` intrinsic function can tag `alloca` locations as GC roots, but it will force reference values to be held in the stack memory, which has performance problem. The `@stackmap` and the `@patchpoint` intrinsics can identify references in registers, but cannot update stack roots when the copying GC moves their referenced objects. The `@statepoint` intrinsic function handles object movement by creating a new SSA variable for the new address of each stack root after the statepoint. This exposes the change of object addresses to the language developers, forcing them to handle field access and reference equality tests[3] using knowledge about the specific GC. None of those stack root mechanisms are fully satisfactory. LLVM also does not help its client generate object maps for finding references inside heap objects,[4] forcing the client to consult the platform ABI for object layout, consequently compromising LLVM's abstraction over architecture. The LLVM client also has to implement yieldpoints, but Lin et al. [2015] showed that the efficient implementation involves carefully crafted machine instruction sequences and code patching, which are also architecture details. Overall, LLVM has made very little effort to address the cross-cutting concerns of GC and compilation, making it difficult to support high-performance GC on LLVM.

LLVM is also designed to be *maximal* instead of minimal. LLVM tries to minimise

[2]We abbreviated the names for simplicity. The full names of the intrinsic functions discussed in this paragraph are `@llvm.gcread`, `@llvm.gcwrite`, `@llvm.gcroot`, `@llvm.experimental.stackmap`, `@llvm.experimental.patchpoint` and `@llvm.experimental.gc.statepoint`.

[3]In concurrent copying GC, a process may simultaneously contain two references, one of which refers to an object in the 'from-space', while the other refers to the same object copied to the 'to-space'. Those two references must be considered equal even though they have different addresses, because they logically refer to the same object.

[4]According to the LLVM documentation, object maps (called 'type maps' in the LLVM documentation), alongside several other GC mechanisms such as object allocation and registering global roots and stack maps with the runtime, have been declared as non-goals of LLVM. One of the reasons is that object maps depend on a particular ABI, and not supporting object maps should allow 'integrating LLVM into an existing runtime' which already has its own ABI. They acknowledged that this may 'leave a lot of work for the developer of a novel language'. For more information, read the LLVM documentation on the goals and non-goals of GC with LLVM [LLVMGoals].

the work for language frontends and provides many ready-to-use optimisation passes inside LLVM. The provision of such optimisations also shows that LLVM is now a mature production project. However, just like the case of JVM, optimisations designed for one kind of language (such as C) are usually unhelpful for other languages which are very different (such as Python). Although LLVM provides many language-neutral optimisations, the most important optimisation, specialisation, is language-specific, and is not provided by LLVM. This can be observed from the LLVM-based Unladen Swallow project [UnladenSwallow, 2011] which failed to meet its goal of 5x speed improvement over the official CPython [CPython]. Unladen Swallow uses template compilation. It translates a Python byte code operation (for example, `BINARY_ADD`) into an LLVM-level `call` instruction which calls CPython's C API functions (for example, `PyNumber_Add`) that realises the same functionality. The implementation of each bytecode in CPython usually has a very long code path [Castanos et al., 2012] which usually tests the types and unboxes the operands. Stock optimisations in LLVM are unable to eliminate such long code paths, which are the real bottleneck, using specialisation as PyPy [Bolz et al., 2009] does. Therefore, although Unladen Swallow managed to eliminate the interpretation loop, it never achieved its desired speedup.

LLVM is the main reference according to which we design Mu. But we design Mu to explicitly support managed languages and handle garbage collection internally inside Mu. We also assume most optimisations are the obligation of the client rather than Mu.

3.2.3 VMKit

VMKit [Geoffray et al., 2010] is a common substrate for the development of high-level *managed runtime environments*, providing abstractions for concurrency, JIT-compiling and garbage collection. VMKit glues together three existing libraries: LLVM [Lattner and Adve, 2004] as the JIT compiler, MMTk [Blackburn et al., 2004] as the memory manager, and POSIX threads for threading. The VMKit developers built two clients, for the CLI and JVM respectively as a proof of concept.

As the name suggests, VMKit is not a self-contained virtual machine, but a toolkit that provides incomplete abstractions over certain features. For example, VMKit leaves the object layout to be implemented by the client. As a consequence, the client runtime, which is developed by the high-level language developer, must participate in object scanning and the identification of GC roots. However, object scanning is not a trivial job [Garner et al., 2011].

As a toolkit composed from three existing libraries, VMKit does not have well-defined semantics across the three major concerns, let alone formal verification. VMKit's solution to concurrency is the POSIX Threads library, but threads cannot be implemented as a library [Boehm, 2005], and require a carefully defined memory model involving both garbage collection and the JIT compiler. Built before the C++11 memory model was available, VMKit also implemented GC yieldpoints [Lin et al., 2015] without any atomic memory operations or synchronisation, which has undefined behaviour in C++11 or later.

Nonetheless, VMKit demonstrates that a toolkit that abstracts over the common key features can ease the burden of the development of language implementations, which is also part of the motivation for a micro virtual machine. Bearing out our design decisions, they have identified execution, concurrency and garbage collection as the three fundamental concerns.

3.2.4 Common Language Infrastructure

The Common Language Infrastructure (CLI) is Microsoft's counterpart to the JVM. Its Common Intermediate Language (CIL) is designed for several languages with similar level to VB.NET, C#, etc., but also hosts many different languages including Managed C++, F# and JavaScript. The CLI shares similar problems to the JVM in that it is monolithic and was designed for particular kinds of languages. For example, it has object-oriented primitives, such as inheritance and polymorphism, built into its Common Type System (CTS).

3.2.5 Truffle/Graal

Truffle and Graal [Würthinger et al., 2013] are reusable VM components for code execution developed by Oracle. Truffle is a language implementation framework, while Graal is a compilation infrastructure. Language developers implement a language by writing an abstract syntax tree (AST) interpreter using the Truffle API. The AST interpreter handles node rewriting which is essentially rules for specialising a language. Truffle optimiser implements dynamic *partial evaluation*, i.e. compilation with aggressive method inlining, on top of the API that the Graal compiler provides. When an AST is stabilised, i.e. the tree is invoked for a number of times without being rewritten, Truffle uses partial evaluation to compile the AST into the Graal compiler's high-level intermediate representation, and further optimisation is performed. The optimised code is executed on the Graal VM, a modification of the Java HotSpot VM that uses Graal as its dynamic compiler.

Truffle and Graal aim to provide a reusable code execution engine for implementations of object-oriented languages. This goal sits on a much higher level than Mu, and we consider it as a complement to the lower-level virtual machine such as Mu. A micro virtual machine will also benefit language frameworks like Truffle by providing abstraction over machine architectures. A Mu client can implement a partial evaluation framework like Truffle, and compile multiple languages to Mu IR.

3.2.6 PyPy/RPython

PyPy [Rigo and Pedroni, 2006] is a high-performance Python implementation. It is built on the RPython framework which supports more languages than just Python, including SOM, Racket, and Erlang. RPython is a meta-tracing framework. Language developers implement a language (such as Python) by writing an interpreter (such as PyPy) in RPython, a subset of the Python language. At run time, the JIT compiler performs meta-tracing, i.e. it records traces of operations performed by the interpreter

(PyPy) which interprets the high-level language (Python), and compiles the traces into machine code.

Meta-tracing (as used by Truffle/Graal) and partial evaluation (as used by PyPy/ RPython) are two alternative higher-level meta-compilation techniques for interpreters to make JIT compilation language-independent [Marr and Ducasse, 2015]. They complement lower-level VMs such as Mu.

Our research group has ported the RPython framework to Mu as a client, hoping that it will enable the many languages that are already implemented on RPython to run on Mu instead. Details of this effort are described in Chapter 9.

This work will not only benefit Mu, but may also benefit the PyPy project as well. Currently, the official RPython backend has to implement JIT compilers by writing machine code generators for many platforms. Because the backend translates RPython code to C code, it also has to implement exception handling and garbage collectors for the generated C code. To do this, every function call is followed by a check for pending exception, which introduces a cost. The GC finds roots from the C stack using shadow stacks or a deprecated compiler-specific 'assembler hackery' [PyPyGC], neither of which is satisfactory. Mu can provide RPython with a cross-platform compiler supporting 'zero-cost' exception handling [Itanium] and an exact garbage collector, and therefore ease the engineering of the RPython toolchain. Moreover, once ported, the abstraction provided by Mu means that RPython benefits from any improvement to Mu, such as the addition of new garbage collectors or new compiler targets.

3.3 Summary

Programming language implementation is difficult, and execution (especially JIT compilation), concurrency and garbage collection are the three major concerns that contribute to the complexity of language implementation. Their cross-cutting concerns, including memory model, stack maps, GC barriers and yieldpoints, often overwhelm people without expertise in computer systems. We also described various existing platforms, such as the JVM, LLVM, VMKit, etc., that have attempted to provide general platforms for language development.

Because of the tightly coupled nature of these concerns, we propose micro virtual machines — low-level substrates that are carefully designed to specifically address them. In the next part, we will introduce the design of Mu, our concrete micro virtual machine.

Part I

Mu: A Concrete Micro Virtual Machine

Chapter 4

Mu's High-level Design

In the preceding chapters, we discussed the challenges and the status quo of language implementation. In this chapter, we flesh out the high-level design of Mu, our concrete micro virtual machine. In the following two chapters, we will continue to discuss the Mu intermediate representation (IR) and the Mu client interface (API).

This chapter presents the high-level perspective of the Mu design. Section 4.1 introduces the design goals of Mu; Section 4.2 presents the principles that underpin the design of Mu; Section 4.3 presents the overall architecture of Mu; Section 4.4 introduces the Mu reference implementation; Section 4.5 summarises this chapter.

The work described in this part is presented in 'Draining the Swamp: Micro Virtual Machines as Solid Foundation for Language Development' [Wang et al., 2015].

4.1 Goals

We design Mu based on our goals as follows:

1. Mu provides abstraction over execution, concurrency and garbage collection.
2. We design Mu to facilitate high performance language implementations.
3. We aim to support diverse languages.
4. There shall be a formally verified Mu instance in the long term.

We design Mu with performance in mind. We make careful design decisions based on state-of-the-art implementation strategies, and avoid making flawed short-term decisions that may hinder the development in the long run. It is not our goal to release Mu early for production use.

We design Mu as a general platform for managed languages. Our focus is on languages most exposed to these concerns, namely dynamic managed languages. However, we are exploring clients for languages as diverse as Python, Haskell, Go, Lua and Erlang. It is not our goal to provide an implementation layer that will compete with mature, highly tuned runtimes such as the HotSpot JVM, which have benefited from enormous investment over a decade or more.

We also have the formal verification of Mu as one of our ultimate goals. Following the formal verification of the seL4 microkernel [Klein et al., 2009], a formally verified micro virtual machine will make a further step towards a fully verified system. It is important that although the formal verification of Mu is not part of this thesis, this goal shapes the design of Mu, and we sometimes have to make compromises for the ease of verification. Other members of our research group are actively working the formal verification of Mu.

4.2 Design Principles

To achieve our design goals, we have a number of principles which underpin the design of Mu.

1. Mu aims to be as *minimal* as practical; any feature or optimisation that can be deflected to the higher layers will be, provided that the three major concerns can be efficiently addressed.

2. Mu's client (the higher-level program that uses Mu. See Section 4.3) is *trusted*; improper use of Mu may result in undefined behaviours which may have arbitrary consequence, including crashing the entire system.

3. We use the LLVM intermediate representation (LLVM IR) [Lattner and Adve, 2004] as a *common frame of reference* for our own IR, deviating only where we find compelling cause to do so.

4. We *separate specification and implementation*; Mu is an open specification against which clients can program and which different instantiations may implement.

Minimalism is the number one principle. Because we aim to support a wide range of dissimilar languages, the level of the micro virtual machine must be kept as low as possible in order to avoid introducing intruding design decisions which are harmful to the high-level language implementation. The minimalist design also means the VM is easier to implement correctly, which will facilitate the creation of a formally verified implementation.

However, for the convenience of language developers, minimalism will be compensated for by *client libraries* that sit above Mu, implementing higher level features, conveniences, transformations, and optimisations common to more than one language. There can be different client libraries developed for different kinds of languages, such as object-oriented languages, functional languages, and so on. Mature projects, such as LLVM, already provide many optimisations ready for its clients to use. As of the time of writing, Mu does not have such libraries, yet. However, when such libraries are developed, they will be strictly client libraries, and excluded from the micro virtual machine itself.

The client is trusted. We fully trust the client, because the client has more knowledge than Mu about the concrete language. It understands all the requirements and constraints of the language semantics, such as array bounds checking, to enforce them correctly. Therefore, we trust the client to make the right decisions, and do not impose extraneous protection layers which may lead to unnecessary overhead.

Moreover, according to Castanos et al. [2012], the optimisations that have the greatest impact on the performance of programs are usually language-specific. It is true that there exist many language-neutral optimisations that work for many languages, such as common sub-expression elimination, loop-invariant code motion, dead code elimination, and many others provided by LLVM. Those optimisations may have the effect of doubling or tripling the throughput of programs. However, for higher-level languages, especially dynamic languages, the most important optimisations, such as specialisation, can achieve 10x or, in certain cases, over 100x performance gain over naive implementations [Castanos et al., 2012]. Such optimisations depend on intimate knowledge about the semantics of the concrete language. Therefore, we give the client the power and the responsibility for high-level optimisations. Mu trusts the client to make the right decisions, and assumes, for example, that the transformations made by the client's optimiser are valid. When Mu API functions are invoked, Mu loyally carries out operations according to the specification with no obligatory validations, especially during function redefinition and on-stack replacement which we will introduce in Chapter 6.

We use LLVM IR as a frame of reference for our IR. After LLVM's inception, it has developed into a production compilation framework known for high performance. It is not good practice to develop everything from scratch and abandon such a mature system. Therefore, we follow LLVM IR as closely as possible to design a performant virtual machine, deviating only when we have a compelling reason to do so. For example, Mu is designed for garbage-collected languages while LLVM is not, so the Mu type system has traced reference types which are absent in the LLVM type system.

It is worth noting that Mu is neither an extension to LLVM nor built upon LLVM. Mu is an independent project, and its design principles are very different from that of LLVM. LLVM IR was only used as a frame of reference for the design of Mu IR.

Mu is defined as a specification. This is the most important contribution of Mu. We define Mu as a specification for an abstract machine, which clearly defines the behaviour of Mu. Mu is not merely a collage of features. Although Mu incorporates many different ideas from related projects, including the static single information (SSI) form, garbage collection, the SWAPSTACK primitive, and the C++11 memory model which will be introduced later, the semantics of the Mu IR and the Mu API are carefully defined in one place. Therefore, unlike VMKit, the behaviour of Mu IR programs related to concurrency, execution, garbage collection, and their cross sections, can be reasoned about. This gives the client a dependable platform, and facilitates formal verification.

The specification allows many different compliant Mu implementations to coexist, such as a reference implementation, a high-performance implementation, and a formally verified implementation. The JVM is defined similarly [Lindholm et al., 2014], and that specification has allowed many implementations.

Mu is designed by a committee, but a rather small one. Our team at the Computer Systems research group at ANU holds weekly meetings to discuss the design, in order to ensure that the Mu design adheres to our design principles. We take input from the people working on different aspects of the project, including client development, high-performance Mu implementation, and formal verification. Sometimes, a decision may undergo hot debates, and compromises must be made, because a subtle change in one part of the system may have a chain of implications in other parts. Therefore, in this thesis, we will not only describe the details of the Mu design, but also the reasons behind the seemingly arbitrary design decisions.

4.3 Architecture

A Mu-based language system consists of Mu and a client, as illustrated in Figure 4.1. The *client* is a program which sits above Mu and implements concrete programming languages.

The Mu specification defines the *Mu intermediate representation* (IR) and the *Mu client interface* (API). The Mu IR is the low-level language accepted by and executed on Mu, while the Mu client interface defines the programming interface for client language runtimes. The client language runtime is responsible for (JIT-)compiling source code, bytecode, or recorded traces into Mu IR, which is delivered to Mu via the API. The Mu client interface specifies how the client may directly manipulate the state of Mu, including loading Mu IR code by sending messages to Mu, and how Mu-generated asynchronous events are handled by the client (i.e. trap handling).

Note that the separation between the micro virtual machine and the client is conceptual. Mu allows clients to be *metacircular*. Metacircular Mu clients are implemented in Mu IR, just like other metacircular virtual machines, such as JikesRVM [Alpern et al., 2009], which are implemented in their own languages. Such clients run on Mu, and control Mu using the Mu API from inside. Metacircular clients are usually built into a *boot image* (see Section 9.1.4) which contains both Mu and the client, as depicted in Figure 4.2.

The abstract state of an executing Mu instance comprises some number of execution engines (threads), execution contexts (stacks), and memory accessed via references or pointers. Mu's abstract threads are similar to (and may directly map to) native OS/hardware threads. Stacks contain frames, each containing the context of a function activation, including its current instruction and values of local variables. Memory consists of a garbage-collected heap, a static memory, and memory allocated on the stacks. The abstract state can be changed by executing Mu IR code directly or by invocation of operations by the client through the Mu client interface.

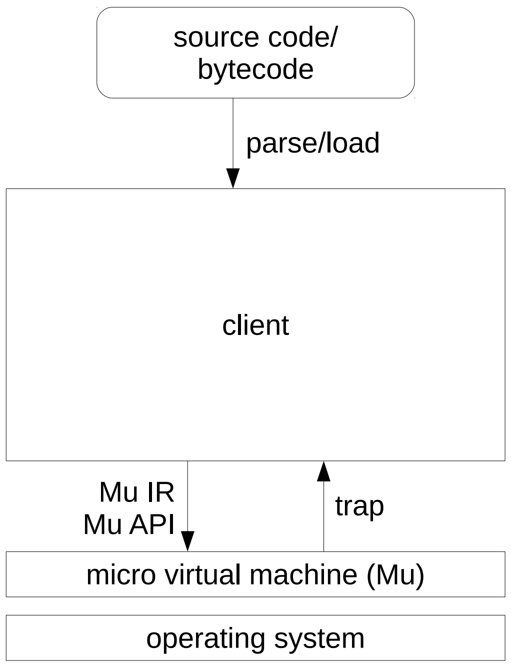
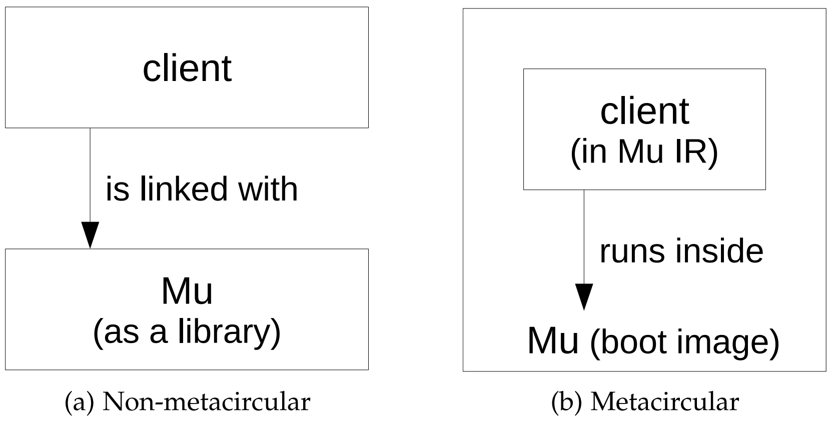


Figure 4.1: Mu architecture. A Mu-based language system consists of Mu and a *client* sitting above it. The client implements the concrete language. At run time, the client loads the source code or bytecode of the language, and translates into the Mu intermediate representation (IR). The client controls Mu via its API. Mu is responsible for the execution of Mu IR programs, usually via JIT compilation. During execution, Mu calls back to the client via *traps* (see Section 6.3.1) so that the client can handle events which cannot be handled by Mu, such as lazy code loading and run-time optimisation. As illustrated in the figure, the client is much bigger than the micro virtual machine, because Mu aims to be a thin abstraction layer, while the client handles all language-specific work and most optimisation.



(a) Non-metacircular (b) Metacircular

Figure 4.2: Non-metacircular vs metacircular clients. A non-metacircular client, as in Figure 4.2(a), uses Mu as a library which can be statically or dynamically linked with the client. A metacircular client, as in Figure 4.2(b), is written in Mu IR, and runs inside a Mu instance. Mu defines an API to build both Mu and Mu IR programs into a *boot image* (see Section 9.1.4), allowing Mu and a metacircular client to be implemented as one executable image.

4.4 Reference Implementation

Despite being a 'micro' virtual machine, the engineering effort required to build a high-performance Mu implementation is non-trivial, precisely because it implements three of the most vexing abstractions managed languages depend on. However, from the perspective of Mu design, we need a working Mu instance to show whether the design really works, and to expose design flaws as early in the design process as possible. We also need to let client developers evaluate the Mu design on a runnable Mu instance, and provide feedback. It was therefore essential to develop a reference implementation that was functionally correct and easy to maintain, even if performance was poor.

We developed the reference implementation, codenamed 'Holstein'[1] , in Scala. This implementation executes as an interpreter, uses green threads to execute multiple Mu threads, and has a simplified but realistic copying region-based garbage collector based on Immix [Blackburn and McKinley, 2008]. Although Scala is a JVM-based language, we avoided using the JVM garbage collector to manage Mu heap objects, but allocated the Mu heap natively as raw memory. Mu memory access operations are always performed using raw addresses, which allows interaction with native programs.

The execution of Mu IR code on Holstein is hundreds of thousands of times slower than the equivalent C program compiled with GCC. Nonetheless, Holstein can run the RPySOM interpreter and the minimum core of the PyPy interpreter, as will be described in Chapter 9.

In addition to Holstein, our research group also has high-performance Mu implementation, codenamed 'Zebu', under active development. That implementation is beyond the scope of this thesis, except inasmuch our design principle of permitting efficient implementation allows Zebu to exist.

4.5 Summary

This chapter introduced the design of Mu. Mu is minimalist. It only deals with the three major concerns, and offloads language-specific concerns to the client. Mu is defined as a specification, which allows multiple implementations, and also facilitates formal verification.

Mu is open source. Both the specification and the source code of reference implementations are available on our website [MicroVM].

In the next chapter, we will present the Mu Intermediate Representation.

[1]Holstein is the name of a cow breeds. We use this name because 'Mu', when read as 'moo', is the sound of cows. Similarly, we named our high-performance Mu implementation 'Zebu', another cow breeds.

Chapter 5

Mu Intermediate Representation

In the preceding chapter, we presented Mu's high-level design. This chapter discusses the details of the Mu Intermediate Representation (IR). The Mu IR consists of the Mu type system and the Mu instruction set, both of which are designed to address the major concerns of Mu. We will introduce the Mu IR design, as well as the reasons behind the design decisions.

This chapter is structured as follows. Section 5.1 presents the overall structure of the Mu IR; Section 5.2 describes the Mu type system; Section 5.3 introduces selected parts of the Mu instruction set; Section 5.4 summarises this chapter.

5.1 Overview

The Mu intermediate representation (IR) uses LLVM IR as a frame of reference, because LLVM is a well-established framework known for good performance.

Figure 5.1 shows an example of a Mu IR bundle. A bundle contains many top-level definitions. Lines 1–4,9,12–13 define types. Line 15 defines a function signature. Lines 6–7 define constants. Line 10 allocates memory in the static space[1] . Both line 17 and line 19 define functions, but only the function on line 19 has a body. The function on line 17 has no body, but can be lazily added by the client (see Section 6.3.2).

The Mu IR uses a form equivalent to the static single information (SSI) form [Ananian, 1999]. SSI is a variant of the static single assignment (SSA) form [Cytron et al., 1991] which LLVM uses. We will cover the details of the IR in the rest of this section.

5.2 Type System

As a low-level VM, the Mu type system is simple and low-level. Like LLVM, Mu has primitive numeric and aggregate types, but also has reference types built into the type system to support garbage collection.

As shown in Table 5.1, the types can be divided into four categories: untraced numerical types, composite types, traced reference types, and miscellaneous types.

[1]The Mu specification currently calls it 'global' memory instead of 'static' memory. This name is confusing because both the garbage-collected heap and the permanent static space are globally accessible. We plan to change the name.

```
.typedef @i64     = int<64>
.typedef @double  = double
.typedef @void    = void
.typedef @refvoid = ref<@void>

.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>

.typedef @Node    = struct<@i64 @NodeRef>
.typedef @NodeRef = ref<@Node>

.funcsig @BinaryFunc = (@i64 @i64) -> (@i64)

.funcdecl @square_sum <@BinaryFunc>

.funcdef @gcd VERSION %v1 <@BinaryFunc> {
    %entry(<@i64> %a <@i64> %b):
        BRANCH %head(%a %b)

    %head(<@i64> %a <@i64> %b):
        %z = EQ <@i64> %b @i64_0
        BRANCH2 %z %exit(%a) %body(%a %b)

    %body(<@i64> %a <@i64> %b):
        %b1 = SREM <@i64> %a %b
        BRANCH %head(%b %b1)

    %exit(<@i64> %a):
        RET %a
}

.expose @gcd_native = @gcd #DEFAULT @i64_0
```

Figure 5.1: Sample Mu IR bundle. Mu IR is similar to LLVM IR. In this example, there are top-level definitions for types (`.typedef`), constants (`.const`), static variables (`.global`), function signatures (`.funcsig`) and functions (`.funcdef` and `.funcdecl`). The function `@gcd` calculates the greatest common divisor using the Euclidean algorithm. Like LLVM, a function contains multiple basic blocks (`%entry`, `%head`, `%body` and `%exit`), each of which contains multiple instructions. The last instruction of a basic block must be a *terminator instruction* which may jump to another basic block or leave the function. In this example, the `BRANCH` instructions are unconditional jumps, and the `RET` instruction returns from the function. The instruction '`BRANCH2 %z %exit(%a) %body(%a %b)`' is a conditional jump, which jumps to `%exit` when the condition `%z` is true, or jumps to `%body` otherwise. Like LLVM, a variable can only be assigned at one place, observing a single assignment. Unlike LLVM, local variables have basic block scope instead of function scope, and must be explicitly passed to the next block at every branch.

Type	Description
`int`$\langle n\rangle$	Fixed-size integer type of n bits
`float`	IEEE754 single-precision (32-bit) floating-point type
`double`	IEEE754 double-precision (32-bit) floating-point type
`uptr`$\langle T\rangle$	Untraced pointer to a memory location of type T
`ufuncptr`$\langle sig\rangle$	Untraced pointer to a native function with signature sig
`struct`$\langle T_1\ T_2\ldots\rangle$	Structure with fields $T_1\ T_2\ldots$
`hybrid`$\langle F_1\ F_2\ldots\ V\rangle$	A hybrid with fixed-part fields $F_1\ F_2\ldots$ and variable part type V
`array`$\langle T\ n\rangle$	Fixed-size array of element type T and length n
`vector`$\langle T\ n\rangle$	Vector type of element type T and length n
`ref`$\langle T\rangle$	Object reference to a heap object of type T
`iref`$\langle T\rangle$	Internal reference to a memory location of type T
`weakref`$\langle T\rangle$	Weak object reference to a heap object of type T
`funcref`$\langle sig\rangle$	Function reference to a Mu function with signature sig
`stackref`	Opaque reference to a Mu stack
`threadref`	Opaque reference to a Mu thread
`framecursorref`	Opaque reference to a Mu frame cursor (see Section 6.3.3)
`irbuilderref`	Opaque reference to a Mu IR builder
`tagref64`	64-bit tagged reference
`void`	Void type

Table 5.1: The Complete Mu Type System

5.2.1 Untraced numerical types

This category includes the `int`, `float`, `double`, `uptr` and `ufuncptr` types. These types represent non-reference data types, and are not traced by the garbage collector.

Integers are fixed-size, like C and LLVM, because Mu is only a thin layer of abstraction over the machine. There are separate floating point types, following the IEEE-754 format. The specification requires Mu to implement integers up to 64 bit in size, while not preventing the implementation of larger sizes if a given implementation considers it beneficial. These fixed-size integers and floating point types can usually be represented efficiently in machine registers.

Like LLVM, integers do not have signedness, but concrete operations, including `UDIV` and `SDIV`, may treat integer operands as signed or unsigned. In fact, most operations (such as `ADD` and `XOR`) are sign-agnostic, and the only operations involving signedness are sign extensions and the conversion between integers and floating point numbers. As a minimal design, Mu does not automatically extend or truncate integers when calling functions, as C and Java do. Thus, not having signedness on integers simplifies the overall design.

We declined the alternative design that only provides 'word' types of different sizes, for both integers and floating point numbers. Processors usually have separate general-purpose and floating point registers, and calling conventions of many platforms (such as x64 and ARM) also require integer arguments and floating point arguments to be passed in different registers. The Mu-level static type information helps the backend code generator to make the right decision.

Mu also has the untraced pointer types. The pointer types are part of the unsafe native interface, and allow Mu IR programs to directly access raw memory and call native functions. We define pointers as word-sized integers, and these can be converted back and forth from integer types. We will discuss pointers further in Section 5.3.7.

5.2.2 Composite types

This category includes the `struct`, `hybrid`, `array` and `vector` types. They describe structured data types made up from smaller components.

As a low-level VM, the composite types are similar to C and LLVM. Structures and (fixed-size) arrays are direct counterparts of those in C.

The `hybrid` type is inspired by the Maxine JVM [Wimmer et al., 2013]. Like a `struct`, a `hybrid` has fields in its fixed prefix, but it is followed by an array of unspecified size, i.e. the 'variable part'. The size of the variable-length array part in a `hybrid` is specified at allocation time. The closest counterpart in C99 is the struct types with a flexible array element, such as `struct Foo { int f1; char f2; int v[]; }`. In fact, `hybrid` is the only type whose size is determined at allocation time rather than compile time. Because the length of a Java array is also determined at allocation time, a Java client should represent Java arrays as the `hybrid` type of Mu. It should put an `int` in the fixed part holding the size of the array, and use the variable-length part for the elements. We also expect that most language clients would implement their string types with a `hybrid` type. Object-oriented languages can use `hybrid` to implement 'type information blocks (TIBs)' as Maxine does, using the fixed part for class metadata and the variable part for the virtual function table.

The `array` type represents fixed-size arrays. A value of `array<T n>` contains exactly *n* elements, each of which holds a value of type `T`. Unlike Java, the length *n* of a Mu array is a constant, and is part of the type, therefore it is not recommended to represent Java arrays using the `array` type of Mu. Like C, fixed-size arrays can be embedded into structs, hybrids or other arrays.

The `vector` type is designed for single-instruction multiple-data (SIMD) operations. Nowadays, SIMD instruction sets (such as ARM's NEON and Intel's SSE) have become increasingly available on commodity hardware, and commodity compilers (such as GCC) already assume such availability on modern platforms (such as x64). Even high-level languages such as JavaScript have gained experimental support for SIMD instructions, and Jibaja et al. [2015] have reported significant performance improvement from the SIMD language extension. Given this trend in the development of hardware, Mu also provides an abstraction over SIMD operations to support such need from the higher level.

It is noteworthy that in Mu, the type of data is orthogonal to where the data is allocated. For example, a `struct` can be allocated in the heap using the `NEW` instruction, in the stack using the `ALLOCA` instruction, in the static memory using the `.global` top-level definition, or as a local variable. As we will introduce later, the type of a reference to a heap-allocated object of type `T` is `ref<T>`. For example, '`array<int<32> 10>`' is the type of an array of ten 32-bit integer elements, regardless of allocation, but '`ref<array<int<32> 10>>`' is a reference to a heap object that contains that array. This is different from Java where values of class types, interface types and array types are implicitly references to heap-allocated objects. However, it is similar to C++ where a class type `C` is still a value type, and the type of pointer to `C` is '`*C`'.

Unlike Java classes, Mu does not add implicit headers (such as the fields of `java.lang.Object`) to Mu types, since every programming language may have its own object model. For example, a Java object usually has a reference to its type information block (TIB), a hash code and a lock; a Python object usually has an ID, a type and a dictionary that holds its attributes using keys and values. But as a language-neutral substrate, Mu does not automatically generate any of those headers. We expect the clients to design the object layout by themselves. If the client implements, for example, a Java-like type hierarchy, the preferred place to put the Java-specific headers (such as the reference to the type information block) for classes and arrays is at the beginning of a `struct` or in the fixed part of a `hybrid`.

5.2.3 Traced reference types

This category includes `ref`, `iref`, `weakref`, `funcref`, `stackref`, `threadref`, `framecursorref` and `irbuilderref`. These types are traced by the garbage collector.

Object references (`ref`) are references to objects that have been allocated in the heap managed by the garbage collector. In Mu, an *object* is defined as the unit of memory allocation in the heap. We are deliberately agnostic about the sorts of languages and type systems implemented by clients; our use of the term *object* does not presuppose any sort of object-orientation. From the client's perspective, objects are headerless. The garbage collector may add headers to heap objects, but they are details of a particular Mu implementation, thus not visible to the client.

Internal references (`iref`) provide references to memory locations that may be internal to objects (e.g. , `array` elements or `struct` fields). Both object references and internal references are traced, and will keep their referents alive on the heap if the reference is itself reachable from GC roots. We introduced `iref` in addition to `ref` to create a RISC-style instruction set, where address computation is decoupled from memory accesses. The details will be discussed in Section 5.3.4.

Weak references (`weakref`) are object references that Mu may set to `NULL` when their referent is not otherwise (strongly) reachable.

Function references (`funcref`) refer to Mu functions. We define function references as traced, too, in the sense that the runtime can always find all instances of function reference values in the entire micro virtual machine. This property is useful to implement function redefinition, which will be described in Section 6.3.2.

The type system also includes a number of opaque reference types. They refer to Mu entities such as threads, stacks, frame cursors (see Section 6.3.3) and IR builders (see Section 6.2.2). Although not all of them have to be managed by the garbage collector, the Mu specification defines them as opaque types so that concrete Mu implementations can choose their preferred strategy for managing these internal objects.

Values of all reference types can only be created in controlled ways, such as via the `NEW` instruction which allocates heap objects, and the `new_thread` API function which creates threads. Specifically, the client cannot cast integers or pointers into object references. Such a separation is a building block for exact garbage collection, which will be discussed in Section 5.3.4.

5.2.4 Miscellaneous types

This category includes `tagref64` and `void`.

The tagged reference type `tagref64` represents a tagged union of three types: (1) `double`, (2) `int<52>` and (3) a `ref<void>` and an attached `int<6>` tag. This type carefully squeezes the bits from the NaN space of the IEEE754 double-precision number to encode integers and object references. This type is especially useful for dynamic languages. This type can represent small integers and floating point numbers directly, instead of having to allocate heap objects for numerical values which tend to be very common. A 6-bit integer can be stored together with the object reference. This tag can be use to identify the object's type without having to read from the object header.

Tagged references are described by Gudeman [1993], and also used by Spider-

Monkey [Mozilla] and V8 [Google], two well-known JavaScript implementations. The double+int+ref union is only one of the many possible designs of tagged references (such as V8's int+ref union). It is an open topic to support different tagged reference designs.

The `void` type is special, as it can only be used as the referent type of references (i.e. `ref<void>`, `iref<void>` and `weakref<void>`) and pointers (i.e. `uptr<void>`) to denote that the reference or pointer can refer to any type. Although the Mu type system does not support subtyping directly, Mu defines the *prefix rules* (see Section 5.3.4) to facilitate the implementation of language-level subtyping and polymorphism. The `void` type is a prefix of all types.

Unlike LLVM and C, Mu functions do not return `void`, but instead may return a zero-tuple of values.

5.3 Instruction Set

The Mu specification defines a comprehensive instruction set. This section introduces some important parts of it.

5.3.1 Basic Instructions

The most basic instructions are arithmetic operations, bit-wise operations, comparison operations, conversion operations and conditional move. These instructions are very simple, and closely follow LLVM analogues. For example:

- An `ADD` instruction '`%r = ADD <@i64> %x %y`' adds two numbers.
- An `SDIV` instruction '`%r = SDIV <@i32> %x %y EXC(%bb1(%r) %bb2())`' divides `%x` by `%y`, treating both operands as signed numbers, and branches to basic block `%bb1`. Particularly, $-$`0x80000000`$/-1 = -$`0x80000000`. But if `%y` is zero, it branches to the basic block `%bb2`.
- An `XOR` instruction '`%r = XOR <@i64> %x %y`' computes the bitwise exclusive or of two numbers.
- An `ULE` instruction '`%c = ULE <@i64> %x %y`' compares two variables for 'less than', treating both operands as unsigned numbers.
- A `SITOFP` instruction '`%r = SITOFP <@i64 @double> %x`' converts an integer to a floating point number, treating the integer operand as signed.
- A `SELECT` instruction '`%r = SELECT <@i1 @i64> %c %x %y`' returns `%x` when the condition `%c` is 1, or `%y` otherwise.

These instructions are applicable to vector values as well as scalars.

For the convenience of the micro virtual machine rather than the client, the types of the operands are explicitly written as type arguments so that the Mu backend compiler does not need to infer the type of any instruction from the types of its operands.

These instructions and their semantics can be tracked back to their LLVM origin, but differences exist because of the differences stemming from our design principles. We now outline some key differences.

Less undefined behaviour Mu IR has *less undefined or implementation-defined behaviour* than C and LLVM.

The C programming language was designed to be applicable to many different hardware platforms. But in 1970s and 1980s, the designs of systems had not converged. The 8-bit byte length, the two's complement representation of negative numbers, and the IEEE-754 floating-pointer number representation were not common. Different processors at that time also handled integer overflow and division-by-zero error differently. Therefore C left much of these behaviours implementation-dependent or undefined. The C programming language also put its trust in the programmer to handle erroneous conditions such as division-by-zero error. Designed for C-like languages, LLVM inherited much of C's undefined behaviour in the LLVM IR instructions.

However, Mu needs to provide a *portable* platform across different hardware, and support languages with objectives like those of Java. 'Undefined behaviour' means *anything* is allowed to happen, from nothing to the computer catching on fire, and must be prevented if at all possible. If Mu had undefined behaviour in trivial cases such as integer overflow, the client would have no choice but to check the operands before every operation, which would lead to significant performance penalties. On the x64, ARMv6, ARMv7, AArch64 and POWER platforms, the behaviour in such corner cases (including overflow, division by zero and shifting an integer by more than its bit width) are defined, and there are signs of convergence.[2] Therefore we give defined behaviours for all arithmetic and logical instructions in Mu, with division by zero as a special case,[3] using the behaviour of the Java language as a frame of reference. It is still an open topic to determine a larger set of instructions that are available or easily implementable on all platforms, such as rotation and saturating addition, but it is certain that Mu must provide abstraction over such machine-level instructions for the client to use them without resorting to the 'unsafe native interface'

[2]For example, most platforms (with RISC-V as a notable exception) have instructions that add two numbers, discard overflowed bits, and set or clear the 'carry' and 'overflow' status flags. In the ARMv6 and ARMv7 instruction `LSL` (logical shift left) with the right-hand-side operand in a register, the last 8 bits of the right-hand-side contribute to the shift amount. But in AArch64, only the last n bits of the right-hand-side is significant, where 2^n is the left-hand-side register length in bits. The behaviour of AArch64 is the same as x64, POWER and RISC-V.

[3]For the `UDIV`, `SDIV`, `UREM`, `SREM` instructions, they have defined behaviours for all inputs if the client provides an exception clause to specify where the execution shall continue if the divider is zero. For example, with the exception clause `EXC(%bb1(%r) %bb2())`, the instruction `%r = SDIV <@i32> %x %y EXC(%bb1(%r) %bb2())` will continue from the `%bb2` basic block if `%y` is zero. The client can always supply exception clauses to ensure all arithmetic operations have defined behaviours for all inputs. However, if the client is absolutely sure the divider can never be zero, it can omit the exception clause, and the instruction will have undefined behaviour if the divider is zero.

(see Section 5.3.7).

Note that Mu does not attempt to eliminate *all* undefined behaviour. Mu is designed with performance in mind. If forcing defined behaviour makes the efficient implementation of high-level languages impossible, we will leave the behaviour undefined. More often than not, the client can efficiently check or avoid abnormal conditions that lead to undefined behaviour. Take an array bounds check as an example. The client can insert array bounds checks to detect out-of-bound errors, using primitive arithmetic and relational instructions of the Mu IR. However, if the client can prove at compile time that an array element access never goes beyond the array bounds, it can safely elide the checks. It is similar for null reference access — the client can eliminate the check if it can prove that a reference is never null. Such optimisations require language-specific knowledge. For example, in Java, references are always nullable; but in Kotlin, a reference is not nullable unless annotated with the '`?`' symbol, such as '`String?`'. Different languages also have different array object layouts, so there is no general way to find the capacity of array objects. A language-neutral micro virtual machine cannot efficiently handle such abnormal conditions without knowing the language specifics. Therefore, we leave the responsibility of handling such abnormal conditions to the client. In order to reduce the burden of the client implementers, there can be *client-side libraries* that implement such optimisations which are common for a particular kind of language, but not all languages. Such client-side libraries will not be part of the Mu specification, because by design, Mu must remain minimalist and language-neutral.

Fewer annotations for optimisation Mu instructions generally have *fewer annotations* than their LLVM counterparts. For example, in the LLVM instruction '`%c = add nsw i32 %a, %b`', the `nsw` (no signed wrap) keyword lets LLVM assume signed overflow never occurs. The comprehensive set of *optimisers* is a selling point of LLVM. The `nsw` annotation allows LLVM to make use of the fact that 'signed integer overflow is an undefined behaviour in C', so that C expressions like '`x + 1 > x`', where x is a signed integer, can be always considered true regardless of the actual value of `x`. However, according to Mu's design principles, since the client has full knowledge about the semantics of the language, such optimisations *can* and *should be* offloaded to the client, thus we do not adopt this LLVM annotation in Mu. The behaviour of the Mu `ADD` instruction is defined to discard overflowed high bits (i.e. `mod` n addition where n is the bit length of the integer) which is the same as on most commodity processors.

In fact, the optimisation mentioned above, i.e. replacing '`x + 1 > x`' with 'true', can only be done by the client, because such replacement is only valid for languages where signed integer overflow has undefined behaviour, such as C and C++. In Java, integer overflow has wrap-around semantics. For example, '`0x7fffffff + 1`' is, according to the Java language specification [Gosling et al., 2014], equal to `-0x80000000`. Therefore, '`x + 1 > x`' still has a chance to be false when `x == 0x7fffffff`, and loops that use this expression as a condition still have a chance to break out. The integer type of Python 3, `int`, has arbitrary precision. If a Python client specialises Python `int` into a fixed-size integer, such as `int<64>`, the code must always detect overflow, revert back to the arbitrary-precision `int` type, and still produce the correct mathematical result. As we can see, even such a trivial case of optimisation involves language-specific semantic details. This again shows why a language-neutral substrate like Mu should off-load many optimisations to the client which understands the language semantics. As we mentioned before, client-side libraries can help client implementers by providing optimisations for languages that share similar semantics.

In order to support languages that support arbitrary precision integers, notably Python, Mu's arithmetic operations may return status bits to indicate overflow. For example, in the Mu instruction '`(%result %sovf) = ADD [#V] <@i64> %a %b`', the `#V` flag causes an additional one-bit value `%sovf` to be returned, which is 1 if signed overflow occurs. High-performance implementations of languages like Python usually speculatively compile the code to use fixed-size integers, and trigger recompilation when overflow actually occurs. In Mu, this flag can be used in conjunction with the `TRAP` instruction to support such de-optimisation, which will be discussed in Section 6.3.1

5.3.2 Control Flow

In Mu IR, as shown in Figure 5.1, every Mu function version[4] is comprised of one or more basic blocks, each of which has one or more instructions. Basic blocks take parameters. The first basic block (i.e. the entry block) takes the parameters of the function, and every branch must explicitly pass arguments to the destination. Similar to Mu IR, the Swift Intermediate Language [SIL, 2017] also has basic blocks which take parameters.

Similar to SSI [Ananian, 1999], all local variables in Mu IR have basic block scope. Basic blocks with parameters are equivalent to lambda-lifted functions that tail-call each other on branching. By contrast, the SSA form is equivalent to nested functions structured by the dominator tree of basic blocks, according to Appel [1998].

Originally, Mu used the same SSA form as LLVM uses. However, liveness analysis is slow, and has to be performed by the micro virtual machine in the backend. The current Mu IR limits the scope of local variables to their respective basic blocks, which effectively forces the client to guarantee the liveness of local variables by passing them explicitly. This change increases the burden of the client, but is in line with our design principles. Having a clearer semantics of variable scopes should also help the formal verification of Mu. For example, to show that a use of a variable is legal in the SSA form, we must prove that its definition dominates its use, which is difficult. In the Mu IR, all local variables have basic block scope, so a local variable can only be used after its definition in the same basic block.

[4]Because Mu supports function redefinition, we distinguish between 'functions' and 'function versions'. A function has zero or more versions (zero when a function is subject to lazy code loading), and every version has a control flow graph. See Section 6.3.2 for more details.

```
.funcsig @addsub.sig = (@i64 @i64) -> (@i64 @i64)

.funcdef @addsub VERSION %v1 <@addsub.sig> {
    %entry(<@i64> %x <@i64> %y):
        %a = ADD <@i64> %x %y
        %s = SUB <@i64> %x %y
        RET (%a %s)
}

.funcdef @caller VERSION %v1 <@caller.sig> {
    %entry(<@i64> %a <@i64> %b):
        (%sum %dif) = CALL <@addsub.sig> @addsub (%a %b)

        // More instructions here
}
```

Figure 5.2: Multiple Return Values. A Mu function takes zero or more parameters, and returns zero or more return values. The signature '`.funcsig @addsub.sig = (@i64 @i64) -> (@i64 @i64)`' indicates that the function takes two `@i64` as parameters, and returns a pair of `@i64` values. The `@addsub` function takes two integers as parameters, and returns both their sum and their difference.

5.3.3 Function Calls and Exception Handling

A `CALL` instruction '`%rv = CALL <@sig> @func (%arg1 %arg2)`' calls a Mu function. Mu IR programs must explicitly truncate, extend, convert or cast the arguments to match the signature. Mu also provides a `TAILCALL` instruction which directly replaces the stack frame of the caller with a frame of the callee rather than pushing a new frame. The client must explicitly generate `TAILCALL` instructions to utilise this feature. Mu implementations need not automatically convert conventional `CALL`s into `TAILCALL`s, though an implementation might.

Mu allows a function to return multiple values rather than just one. This is similar to many functional languages such as ML and Haskell which support tuples, but different from C, C++, Java and LLVM. Figure 5.2 shows a function that returns two values instead of one. In fact, all Mu instructions may produce multiple values, including `CALL` and, notably, the `CMPXCHG` instruction which returns both the old value and an indication of success or failure. This symmetry was introduced during our design of the on-stack replacement (OSR) interface, which will be discussed in detain in Section 7.3.7.

Unlike LLVM, Mu has built-in exception handling primitives that do not depend on system libraries, which are usually platform-specific, compiler-specific, and mainly designed for C++. Since Mu is also designed to support exact stack scanning for GC (see Section 5.3.4) and has built-in support for on-stack replacement (OSR) (see Section 6.3.3), Mu implementations will inevitably have a powerful stack unwinder that can also be used for exception handling.

As shown in Figure 5.3, the `THROW` instruction generates an exceptional transfer of

```
.funcdef @bar VERSION %v1 <@bar.sig> {
    %entry():
        %the_exc = NEW <@MyExceptionType>
        THROW %the_exc
}

.funcdef @foo VERSION %v1 <@foo.sig> {
    %entry(<@i64> %x):
        %result = CALL <@bar.sig> @bar (%x) EXC(%nor(%result) %exc())

    %nor(<@i64> %result):
        // handle result here

    %exc() [%the_exc]:
        // handle exception here
}
```

Figure 5.3: Mu Exception Handling. When an exception is thrown in the callee, the call site with an exception clause `EXC` branches to the exceptional destination (in this case `%exc`), where the exceptional parameter `%the_exc` receives the exception, which is just an arbitrary object reference. Annotating the call site allows the JIT compiler to generate sufficient metadata to implement 'zero-cost' exception handling as used by the Itanium ABI for C++ [Itanium].

control to the caller of the current function.[5] The exception is caught by the nearest caller's `CALL` instruction with an *exception clause*, which branches to the designated basic block where an exceptional parameter receives the exception value. Unlike LLVM, an exception in Mu is an arbitrary object reference[6]. This kind of `CALL` unconditionally catches all exceptions and the type of the exceptional parameters is always `ref<void>`. The client is responsible for implementing its own exception hierarchy which can be complex (like Java's and Python's) or simple (like Lua's and Haskell's, where an error is simply a string message). The client should generate Mu IR code to check the run-time type of the exception object, and decide whether to handle, re-throw or clean up the current context.[7]

This design allows the implementation of '*zero-cost*' exception handling which is also used by the Itanium ABI for C++ [Itanium]. Unlike the `errno` variable in C which has to be checked after each call, regardless whether any error actually occurred or not, the Mu runtime can instead only set the program counter to the exceptional destination during stack unwinding when an exception is actually thrown. Therefore, there is no checking when a function returns normally, and does not penalise function calls when exceptions are not thrown, which is usually the hot path. This 'zero-cost' exception handling scheme requires the compiler to generate metadata and code

[5]Language-level exception handling within a function, for example, a `throw` statement in a `try-catch` block in Java, should be translated to branching instructions (`BRANCH` and `BRANCH2`) in the Mu IR. In this case, Mu is not aware of any exceptions being thrown.

[6]We can loosen this rule and allow an exception to be a value of any type, provided that it can be implemented as efficiently as object references.

[7]There is no `finally` in Mu, but it can be implemented as an unconditional catch followed by the actions in the `finally` block and another `THROW` instruction.

to help the stack unwinder identify the call sites, and branch to the exceptional destination with normal parameters and the exception passed. This again assumes that the Mu implementation has a powerful stack unwinder, which also supports GC root scanning and OSR.

5.3.4 Memory Operations

Memory Allocation Support for precise (exact) garbage collection is integral to the design of the instruction set. Heap memory allocation is a primitive operation in Mu. The `NEW` and the `NEWHYBRID` instructions allocate fixed and variable-length objects in the heap, respectively, automatically managed by the garbage collector. Memory can also be dynamically allocated on stacks using the `ALLOCA` and the `ALLOCAHYBRID` instructions which are similar to the `alloca` function in C. Memory cells can be permanently allocated into the static memory using a top-level definition. Newly allocated memory cells are initialised with zeros (numerical zeros or NULL references), which matches the behaviour of common high-level languages (such as Java), has less non-determinism, and is efficient to implement with proper bulk-zeroing, as demonstrated by Yang et al. [2011].

Internal References In Mu, internal references `iref<T>` refer to memory locations inside Mu memory, including the heap, the stack and the static memory. The semantics are similar to pointers except that they are traced by the GC. Mu does not expose the byte-level object layout to the client, since the *size* and *alignment* of types are different among platforms.[8] Therefore, the client cannot use address calculation on `iref` types. Instead, Mu provides several instructions that navigate internal references into nested composite types. For example, the `GETFIELDIREF` instruction takes an `iref` to a struct as parameter, and returns an `iref` to one of its fields; and `GETELEMIREF` takes an `iref` to an array as parameter, and returns an `iref` to one of its elements. These instructions are the counterpart of the LLVM instruction `getelementptr`. We consider our semantics much clearer than `getelementptr`, because we use different instructions for different types, while `getelementptr` is a single instruction that works for all composite types, and is a major source of confusion.[9] Similar to LLVM and most RISC architectures, Mu IR has dedicated `LOAD` and `STORE` operations to access memory. Compared to the JVM which has `getfield`, `setfield`, `*aload` and `*astore` instructions, Mu separates memory addressing and memory accessing, and allows arbitrarily nested object structures to be designed at the client's will. This presence of the `iref` type is vital to this separation. Without `iref`, all memory accessing operations, including `LOAD`, `STORE`, `CMPXCHG` and `ATOMICRMW`, would have to contain something like `getelementptr` as part of the instruction. In the Mu backend

[8]The unsafe native interface (Section 5.3.7) defines the sizes and alignments of *pinned* objects in order to interface with native programs at the ABI (application binary interface) level. However, the unsafe native interface is not supposed to be used in managed programs.

[9]The `getelementptr` instruction in LLVM is so complicated that there is a webpage dedicated to `getelementptr` — 'The Often Misunderstood GEP Instruction' [LLVMGEP].

compiler, the instruction selector can match across multiple memory addressing and accessing instructions, and combine them into a single machine instruction if the architecture supports the appropriate addressing mode.

Abstract Memory Locations According to the Mu specification, the Mu memory consists of abstract memory locations which are simply defined as 'regions of data storage', but not directly mapped to addresses (unless *pinned*, see Section 5.3.7). This allows Mu implementations to represent objects and references in the most efficient way, and, in principle, allows object references to be represented as handles instead of addresses. A copying garbage collector can move objects around in the heap, but this is an implementation detail, which the client cannot observe from object references. In other words, a reference to an object always refers to the same object even if the copying garbage collector moved the object to a different place in the address space of a process.

Prefix Rules and Hierarchical Type System Every (abstract) memory location has a type which is determined at allocation time and never changes. Casting references to the wrong type and accessing the memory has undefined behaviour. But Mu has a series of memory aliasing rules, called '*prefix rules*', to help the client implement hierarchical type systems. Remember that Mu does not bake in the object model of any particular language. All types can be allocated in the heap, the stack or the static memory alike, and Mu does not implicitly insert headers[10] to data types nor provide any run-time type information. However, Mu defines the 'is a prefix of' relation between types. For example, the first field of a `struct` is a prefix of the struct, and the relation is transitive. In this way, subtypes can be implemented by having parent types as first fields, and object references `ref<T>` can be cast to or from references to prefix types. The Mu implementation should lay out objects in such a way that references to the prefix can efficiently access fields of objects that are much bigger in size. Having this requirement, the micro virtual machine may have to lay out objects in a predictable way. However, there is a compromise we have to make. Without knowing platform details, it may be impossible for the client to generate the Mu struct type with the best layout. For example, the best layout for 32-bit machines may be different from the best layout for 64-bit machines because of size and alignment issues. The best layout with respect to concurrency and false sharing may also vary among platforms. We are designing an abstraction layer over the architecture, and we must be able to support object-oriented languages reasonably efficiently. Therefore, when facing multiple concerns, we chose the most predictable behaviour. This design choice is similar to our decision of giving defined behaviours to all binary arithmetic/logical operations on all platforms with the sole exception of division by zero.

[10]Mu implementations may insert headers for their own purposes, including GC, but such headers must be invisible to the client.

Exact Garbage Collection To implement exact garbage collection, Mu must be able to identify all references into the Mu heap. The GC root set is precisely defined as all references in live local variables, stack memory, static memory, those explicitly held by the client (see Section 6.1), and other thread-local states including a thread-local object reference and its pinning set.[11] Because all values in Mu come from the Mu type system, which never confuses references and untraced values, Mu can perform garbage collection internally without client intervention.

GC Algorithm as An Implementation Detail The Mu specification does not mandate any particular GC algorithm, either. This forces the client not to depend on any particular GC algorithm Mu is using, and prevents a class of language design bugs that exposes the GC implementation details to the user, such as PHP's copy-on-write semantics [Tozawa et al., 2009]. This is a critical design decision. People tend to rely on the GC algorithm of a concrete language implementation, which is not a good phenomenon, and should be avoided in the first place. For example, before Python 2.6, programmers habitually open a file, read from it, and expect the reference counting GC to close the file immediately because the last reference to the file object goes away after evaluating the statement.

```python
text = open("example.txt").read()
# The file is expected to be closed here.
```

This does not work in other Python implementations which do not use reference counting, such as PyPy [Rigo and Pedroni, 2006] and Jython [Jython], because tracing collectors do not immediately reclaim objects immediately when they become garbage. The copy-on-write semantics [Tozawa et al., 2009] of PHP is also an attempt to optimise the performance by depending on the naive reference counting GC algorithm, resulting in inconsistent and unfixable semantics when such 'optimisation' is entangled with references (the & operator) [PHP, 2002]. Both Python and PHP are examples of bad design decisions based on naive GC algorithms that hurt long-term profit. Fortunately, we observe that people have started trying to get rid of such dependency on GC. For example, Python 2.6 introduced the '`with`' statement to properly release resources without relying on immediate reference counting.

```python
with open("example.txt") as f:
    text = f.read()
# f is closed here.
```

Hack [Hack], a programming language designed by Facebook based on PHP, is another example, which removed the widely used references (&) in PHP to steer away from its broken semantics. Java also deprecated the `finalize()` method since Java 9,[12] in favor for the `AutoCloseable` interface and the 'try-with-resources' syntax for properly closing files without depending on the garbage collector. Mu will follow this trend, and provide language designers a high-performance GC implementation so

[11]Details about thread-local states are discussed in the Mu specification. See: https://gitlab.anu.edu.au/mu/mu-spec/blob/master/threads-stacks.rst

[12]See the Java 9 API: https://docs.oracle.com/javase/9/docs/api/java/lang/Object.html#finalize--

that they will not design their next language based on a poorly-performing garbage collector.

5.3.5 Atomic Instructions and Concurrency

Mu is designed with multi-threading in mind. Mu has threads and a C11/C++11-like memory model, allowing annotation of memory operations, such as `LOAD`, `STORE`, `CMPXCHG`, `ATOMICRMW` and `FENCE`, with the desired memory ordering semantics. Mu threads may execute simultaneously. Like LLVM, Mu has no 'atomic data types', but defines a set of primitive data types (such as integers and references) eligible for atomic accesses. The supported memory orders are `NOT_ATOMIC`, `RELAXED`, `CONSUME`, `ACQUIRE`, `RELEASE`, `ACQ_REL` (acquire and release) and `SEQ_CST` (sequentially consistent). This gives the client the freedom and responsibility to implement whatever memory model is imposed (or not) by the client language.

Supporting relaxed memory models is not trivial. As a design principle, the client is trusted, and may shoot itself in the foot. Abusing the memory model may result in program errors or even undefined behaviors. However, Mu does not force all users to understand the most subtle memory orders. A novice language-client implementer can exclusively use the strong `SEQ_CST` order even though the Mu implementation supports weaker orderings. Conversely, a conservative implementer of Mu itself can always correctly implement a stronger memory model than required, for example, implementing `CONSUME` as `ACQUIRE` or implementing all memory models as `SEQ_CST`, which will trade performance for simplicity.

In addition to the standard C11-like atomic operations (such as compare-and-swap) Mu provides a futex-like [Franke and Russell, 2002] wait mechanism. The client is responsible for implementing other shared-memory machinery such as blocking locks and semaphores.

Figure 5.4 gives an example of implementing spin locks using the atomic read-modify-write operations provided by the Mu instruction set. To acquire a lock, the `@lock` function repeatedly performs atomic exchange operation to store the value 1 to the shared variable `@spinlock` until it gets 0 as the old value. When multiple threads are attempting this atomic exchange, only the first one can change its value from 0 to 1, while others observe 1 as the old value. To release a lock, the `@unlock` simply writes 0 to the variable, which will be observed by the next thread that successfully acquired the lock. The `ACQUIRE` and the `RELEASE` memory order ensures that as long as the `XCHG` operation observes the old value 0, it *synchronises with* the `STORE` operation from the thread that released the lock by writing 0, and all memory writes performed before releasing the lock *happen before* all memory reads performed after acquiring the lock. Therefore, threads in the critical section can observe memory operations performed by other threads which previously held the same lock.

Drepper [2011] used C++ code to demonstrate how to implement blocking locks, condition variables and other synchronisation mechanisms using atomic read-modify-write operations and the `futex` system call provided by the Linux kernel. Those techniques can be adapted to Mu, too, given that the futex primitive of Mu is modelled

```
.typedef @i64 = int<64>

.const @I64_0 <@i64> = 0
.const @I64_1 <@i64> = 1

.global @spinlock <@i64>

.funcsig @LockSig = () -> ()
.funcsig @UnlockSig = () -> ()

.funcdef @lock VERSION %v1 <@LockSig> {
    %entry():
        BRANCH %head()

    %head():
        %old  = ATOMICRMW ACQUIRE XCHG <@i64> %spinlock @I64_1
        %succ = EQ <@i64> %old @I64_0
        BRANCH2 %succ %exit() %head()

    %exit():
        RET ()
}

.funcdef @unlock VERSION %v1 <@UnlockSig> {
    %entry():
        STORE RELEASE <@i64> %spinlock @I64_0
        RET ()
}
```

Figure 5.4: Implementing spin lock using atomic operations in the Mu memory model. The `@lock` and the `@unlock` function uses atomic operations to acquire and release the spin lock represented as a shared variable `@spinlock`.

after Linux futex. Because those futex-based synchronisation mechanisms can be difficult and tedious to implement, they may be provided by client-level *libraries*, enabling complex implementations to be shared among multiple language clients.

The Mu memory model includes the *happens-before* relationship between memory operations, which is also the basis of the Java memory model [Manson et al., 2005; Gosling et al., 2014] and the C++11 memory model [Boehm and Adve, 2008; ISO, 2012]. The happens-before relationship is a very important concept, because it determines what value a read operation may observe with respect to other write operations in the current thread or other threads. A load operation can never observe values written by store operations which happen after it. If there are two store operations, and one happens before the other, then the second store 'hides' the first store, therefore any load operations which happen after the second store cannot observe the value written by the first store. There are multiple ways to establish the happens-before relationship. The happens-before relationship is consistent with the program order — operations which are performed earlier in one thread happens before operations performed later in the same thread. Memory operations in different threads may have happens-before relationship, too. In the spin-lock example above, when a load operation of the `ACQUIRE` order observes the value written by a store operation of the `RELEASE` order, they form the synchronises-with relationship (a subset of the happens-before relationship) across different threads. Because the precise definition of the happens-before relationship is complicated, we refer the readers to the Mu specification [Wang, b] for more details.

5.3.6 Stack Binding and the SWAPSTACK Operation

Unlike many language runtimes, Mu clearly distinguishes between threads (executors) and stacks (execution contexts).

A *thread* is a flow of control that can progress concurrently with other threads. Modern operating systems provide threads (native threads) as kernel-level scheduling units that share resources with others in the same process,[13] and can be executed concurrently on parallel hardware. Programming languages, on the other hand, may implement language-level threads as native threads, strictly user-level 'green' threads mapped to a single kernel thread, or via an $M \times N$ mapping that multiplexes user-level threads over kernel-level threads.

The execution *context* of a thread includes the call stack as well as other thread-local state, such as a thread-local allocation buffer for GC. The *call stack*, or simply the *stack*, records the activations of dynamically nested function calls. Each activation record or *frame* on a stack records the values of local variables and the saved program counter of a function activation.

In conventional languages such as Java, each thread is only ever associated with one stack. More generally, however, a thread can switch among different stacks, such

[13]Some operating system kernels, such as Linux or seL4, only provide the abstraction of 'tasks' rather than actual 'threads' and 'processes'. Tasks can implement threads, processes or other kinds of isolated containers, depending on what resources are shared among the tasks.

as when switching between coroutines, or handling UNIX signals.[14]

Swapstack is a context-switching operation that saves the execution state of the current thread's top-most activation on its active stack, switches the thread to use a different destination stack, and restores the thread's execution state to that of the destination stack. Swapstack effectively provides the abstraction of symmetric coroutines, and can be used by language implementations to build higher-level language structures such as continuations and lightweight threads, as in Dolan et al. [2013].

Dolan et al. [2013] showed that with the support from the compiler, this lightweight context switching mechanism can be implemented fully in user space with only a few instructions, so it is more efficient than native threads, which inevitably involve transitioning through the kernel. With the compiler knowing the liveness of variables at each Swapstack site, only a subset of all registers need to be saved. This is impossible for library-based approaches, including `setjmp/longjmp`, `swapcontext` or customised assembly code, which have no information from the compiler and must conservatively save all registers.

The term 'Swapstack' was first used by Dolan et al. [2013] to specifically refer to their efficient language-neutral compiler-assisted mechanism in LLVM for context-switching and message passing between lightweight threads of control ('green threads'). In this work, we use the term Swapstack[15] in a broader sense—it is an abstract operation that rebinds the current thread to a different stack, regardless of its implementation.[16]

Swapstack is an integral part of Mu. The `SWAPSTACK` instruction unbinds a thread from one context and rebinds it to another context [Dolan et al., 2013]. When rebound, the thread continues from the corresponding instruction (usually another `SWAPSTACK`), where the destination context paused when last active (bound to a thread).

Figure 5.5 demonstrates the `SWAPSTACK` instruction. This example has two functions written in straight-line code, demonstrating two coroutines swapping to and from each other. Stacks can be created using the `@uvm.new_stack` intrinsic (see section 5.3.8) or the equivalent API function. The `SWAPSTACK` instruction may pass values to the destination stack. If a stack is stopped at a `SWAPSTACK` instruction, it receives the passed values as the return value of the instruction; if the stack is a newly created stack which is stopped at the entry point of a function, the passed values will be received as the parameters of the function.

Figure 5.7 is a slightly more complex example that uses `SWAPSTACK` to imple-

[14]The `sigaltstack` POSIX function can specify an alternative stack where the signal handler runs, instead of the regular user stack.

[15]Following the convention of Dolan et al. [2013], we use Swapstack (Small Caps) to denote the abstract operation, and use `SWAPSTACK` (ALL CAPS) for the concrete instruction in the Mu instruction set.

[16]Therefore, in our terminology, the `Boost.Context` library [Kowalke, 2016], the (deprecated) `swap-context` POSIX function, as well as the LLVM primitive created by Dolan et al. [2013], are all implementations of Swapstack, although the work of Dolan et al. [2013] usually out-performs others because it only saves as many registers as necessary. A Mu implementation also provides an implementation of Swapstack.

```
.typedef @i64 = int<64>
.typedef @d   = double
.typedef @st  = stackref

.const @C1 <@i64> = 101
.const @C2 <@d>   = 102.0d
.const @C3 <@i64> = 103
.const @C4 <@d>   = 104.0d

.funcsig @main.sig = () -> ()
.funcdef @main VERSION %v1 <@main.sig> {
    %entry():
        %cur_st    = INTRINSIC @uvm.current_stack ()
        %coro_st   = INTRINSIC @uvm.new_stack <[@coro.sig]> (@coro)
        // (A)
        %v1        = [%s1] SWAPSTACK %coro_st RET_WITH <@i64> PASS_VALUES <@st> (%cur_st)
        // (C) %v1 == 101
        (%v3 %v4)  = [%s3] SWAPSTACK %coro_st RET_WITH <@i64 @d> PASS_VALUES <@d> (@C2)
        // (E) %v3 == 103, %v4 == 104.0
                     [%s5] SWAPSTACK %coro.st RET_WITH <> PASS_VALUES <> ()
        // (G)
                     RET ()
}

.funcsig @coro.sig = (@st) -> ()
.funcdef @coro VERSION %v1 <@coro.sig> {
    %entry(<@st> %main_st):
        // (B) %main_st is the main stack
        %v2        = [%s2] SWAPSTACK %main_st RET_WITH <@d> PASS_VALUES <@i64> (@C1)
        // (D) %v2 == 102.0
                     [%s4] SWAPSTACK %main_st RET_WITH <> PASS_VALUES <@i64> (@C3 @C4)
        // (F) %s4 did not receive any values
                     [%s6] SWAPSTACK %main_st KILL_OLD PASS_VALUES <> ()
        // unreachable
}
```

Figure 5.5: Example of the basic usage of the `SWAPSTACK` instruction. The identifiers in square brackets before the instruction mnemonics, such as `[%s1]`, give names to instructions. The `@main` function creates a stack with a single frame stopped at the beginning of the `@coro` function. This stack represents a coroutine. As the program executes, the control flow reaches point (A), point (B), . . . , point (G) in this order, during which the `SWAPSTACK` instructions are executed in the order of `%s1`, `%s2`, . . . , `%s6`. From point (A), the thread executes the first `SWAPSTACK` instruction `%s1` which rebinds the current thread to the stack referred to by the stack reference `%coro_st`. As specified by the `PASS_VALUES(%cur_st)` clause, it passes `%cur_st` — the reference to the current stack — to the coroutine stack. As specified by the `RET_WITH <i64>` clause, the main stack will be stopped at the instruction `%s1`, waiting for a value of type `@i64`. After executing `%s1`, the control flow continues from where the coroutine stack was stopping at, which is the beginning of the `@coro` function because it is a newly created stack. The parameter `%main_st` receives the value of `%cur_st`, and the control flow reaches point (B). The thread then executes the next `SWAPSTACK` instruction `%s2`, passing constant `@C1` to the main stack, and reaches point (C). Because the main stack was stopped at `%s1`, the constant `@C1` was received by the return value `%v1`. Instruction `%s3` then passes constant `@C2` to return value `%v2` of instruction `%s2`, and reaches point (D). Instruction `%s4` passes two values `@C3` and `@C4` to `%v3` and `%v4` of `%s3`, and reaches point (E). Instruction `%s5` swaps to the coroutine stack without passing values, and reaches point (F). The last `SWAPSTACK` instruction `%s6` swaps back to the main stack, and destroys the current stack (the coroutine stack) as specified by the `KILL_OLD` clause. The thread then returns at point (G).

```python
def main():
    c = coro(i)

    v1 = c.send(None)
    v2 = c.send(None)
    v3 = c.send(None)

    c.throw(StopIteration)

def coro(i):
    try:
        while True:
            yield i
            i += 1
    except StopIteration:
        # When a generator returns, it yields None.
        return
```

Figure 5.6: A simple Python generator example. The `coro` function is a generator function that yields number `i`, `i+1`, ..., until it receives a `StopIteration` exception. The main function creates the generator, resumes it three times to get three values, and then throws an exception to stop it.

ment a Python-style generator roughly equivalent to the code in Figure 5.6. Like the `CALL` instruction, the `SWAPSTACK` instruction may have an exception clause (see Section 5.3.3), too. The coroutine runs in a loop, sending values back to the main routines via `SWAPSTACK` until it receives an exception. There are several notable differences between the Swapstack primitive of Mu and the generators of Python. Python generators are asymmetric coroutines, where the parent uses the `send` method to resume the child, while the child uses the `yield` expression to resume its implicit parent. Mu stacks are symmetric, therefore every `SWAPSTACK` instruction must have the destination stack as its operand. In Python, the arguments of a generator are supplied when creating the generator, while the arguments to the function of a newly created Mu stack are supplied the first time a thread is bound to it via Swapstack.

In Mu, the Swapstack primitive is not only used by the `SWAPSTACK` instruction for coroutines and green threads, but is also a foundation for many aspects of the micro virtual machine design, including thread creation, trap handling, and on-stack replacement. Stacks are inactive upon creation; a newly created thread starts execution by binding to an inactive stack and using it as its execution context. We will discussed more about the use of Swapstack in trap handling in Section 6.3.1, and we dedicate Chapter 7 to the discussion of our Swapstack-based OSR API.

5.3.7 Unsafe Native Interface

Although Mu is designed for managed languages, Mu does not preclude direct interaction between Mu IR programs and native programs (often written in C), including

```
.typedef @i64 = int<64>
.typedef @st  = stackref
.typedef @StopIteration = struct<...>
.typedef @ref_StopIteration = ref<@StopIteration>

.const @C0 <@i64> = 0
.const @C1 <@i64> = 1

.funcsig @main.sig = () -> ()
.funcdef @main VERSION %v1 <@main.sig> {
    %entry():
        %cur_st    = INTRINSIC @uvm.current_stack ()
        %coro_st   = INTRINSIC @uvm.new_stack <[@coro.sig]> (@coro)

        [%s1] SWAPSTACK %coro_st RET_WITH <> PASS_VALUES <@st @i64> (%cur_st @C1)

        %v1  = [%s2] SWAPSTACK %coro_st RET_WITH <@i64> PASS_VALUES <> ()
        %v2  = [%s3] SWAPSTACK %coro_st RET_WITH <@i64> PASS_VALUES <> ()
        %v3  = [%s4] SWAPSTACK %coro_st RET_WITH <@i64> PASS_VALUES <> ()

        %exc = NEW <@StopIteration>
        [%s5] SWAPSTACK %coro_st RET_WITH <> PASS_VALUES <@ref_StopIteration> (%exc)
        RET ()
}

.funcsig @coro.sig = (@st @i64) -> ()
.funcdef @coro VERSION %v1 <@coro.sig> {
    %entry(<@st> %main_st <@i64> %i0):
        [%t1] SWAPSTACK %main_st RET_WITH <> PASS_VALUES <> ()
        BRANCH %head(%main_st %i0)

    %head(<@st> %main_st <@i64> %i):
        [%t2] SWAPSTACK %main_st
                RET_WITH <> PASS_VALUES <@i64> (%i)
                EXC (%cont(%main_st %i) %exit(%main_st))

    %cont(<@st> %main_st <@i64> %i):
        %i2 = ADD <@i64> %i @C1
        BRANCH %head(%main_st %i2)

    %exit(<@st> %main_st) [%exc]:
        // On exception, destroy the current stack and swap back to the main stack
        [%t3] SWAPSTACK %main_st KILL_OLD PASS_VALUES <> ()
}
```

Figure 5.7: Example of using `SWAPSTACK` in a loop. This example is roughly equivalent to the Python code in Figure 5.6. The first `SWAPSTACK` instruction `%s1` in the `@main` function 'initialises' the coroutine with necessary contexts — the reference to the main stack and the initial number. The first `SWAPSTACK` instruction `%t1` in the `@coro` function swaps back to the main stack. This leaves the coroutine stack in a state that whenever the thread swaps to the coroutine stack, passing no values, the coroutine will swap back, carrying the next value. Concretely, the thread then executes the `SWAPSTACK` instruction `%s2` in `@main` to swaps to the coroutine which continues from `%t1` and swaps back at `%t2`, carrying the first value. Then `%s3` makes `%t2` continue with the normal destination `%cont`, where it computes the next value, jump back to the head, and swap stack again at `%t2`. It is similar for `%s4`. The last `SWAPSTACK` in `@main`, namely `%s5`, throws an exception at the coroutine stack, which receives the exception, and continue from `%t2` to the exceptional destination `%exit`, where it destroys the current stack and swaps back.

system calls[17] and third-party libraries.

Some existing VMs, such as the JVM, intentionally forbid direct interaction with native programs in order to keep the managed language safe. However, since interacting with native programs is usually necessary, bridge code between Java and C has to be written. The JNI [Oracle] interface forces this portion of bridge code to be written in C, and also introduces non-trivial overhead at the C-Java boundary. We take the approach of C# [Hejlsberg et al., 2003], which has built-in support for pointers and allows calling C functions directly. In this way, we encourage programmers to write more code in managed languages like C#. The entire system will be safe because of less C and more managed code, and the compiler has the opportunity to generate low-overhead calling sequences with the knowledge of the native calling convention.

Mu interfaces with native programs at the level of application binary interfaces (ABI) instead of the C programming language. The C language specification does not define the memory layout (including sizes and alignments of objects) or the calling convention, and it is the ABI of each platform that defines them. The main Mu specification is extended with an API for each platform.[18] In principle, the ABI includes a Mu IR-level calling convention equivalent to the C calling convention, and the memory layout is defined to be compatible with the C programming language ABI. This is because C is currently the most widely used language for writing native programs, and other languages, such as C++, Rust as well as managed languages such as Python, usually have their foreign function interfaces (FFI) target the C convention.

Mu assumes that the process runs in a flat (not segmented) address space, which is true for most modern platforms, such as x64, AArch64, and even the 32-bit IA-32 architecture on GNU/Linux which forces some segment registers to be zero.[19] Untraced pointers '`uptr<T>`' and function pointers '`ufuncptr<sig>`' are simply the addresses of data storage regions or entry points of native functions. Memory operations, such as `LOAD` and `STORE`, can access native memory via pointers, the same way as accessing the managed memory via internal references. In addition to the `CALL` instruction, the `CCALL` instruction allows the client to call native functions via untraced function pointers with a specified calling convention, usually the C calling convention.

Object Pinning

Many high-performance garbage collectors move objects within the heap to compact memory. However, heap objects may need to be accessed by native functions, such

[17]The 'system calls' accessible in the C language are usually C functions that wrap the actual system call instructions.

[18]At the time of writing, we have only defined the ABI for x64 platforms that uses the SysV ABI for AMD64 [Matz et al., 2012]. A new Mu ABI needs to be defined when porting Mu to any platform.

[19]Not all platforms use one flat address spaces. Some devices may have multiple address spaces which are accessed using different instructions, such as special instructions for device IO. To support such platforms, the Mu IR needs to be adjusted to distinguish between different address spaces. The `addrspace(x)` annotation for pointer types in the LLVM IR provides a reasonable frame of reference for this.

as the `read` and the `write` system call, when objects contain a buffer to be read or written by the native function. In Mu, heap objects must be *pinned* before becoming accessible via pointers. The *pinning* operation ensures that while an object is pinned, it has a constant address. This prevents the garbage collector from moving the object.

The Mu specification also allows native programs to make atomic memory accesses that synchronise with Mu programs. But we leave the exact way *how* the native program can do this as implementation-defined, because different implementations of the C language (i.e. different C compilers) may implement atomic operations differently while still complying with the C standard.[20] When accessing unmanaged memory, the behaviour has to be implementation-dependent because the underlying system has so much freedom in managing the address space. Addresses may be mapped to devices, and different address regions may be mapped to the same physical memory region and create aliases that are not known statically.

Calling Back from Native Code

Mu functions may need to be called from native functions, too. Some native libraries let their users provide call-back functions which are called by the library on specific events. Mu functions must be *exposed* before becoming callable from native programs. The calling convention of a particular Mu instance may not be the same as the C calling convention, thus adapters may need to be generated to convert between different calling conventions, although it is recommended that Mu let functions follow the C calling conventions, too, to avoid this level of conversion.

While Mu IR has functions, the callable unit of the high-level language may not be functions, but methods or closures. The difference is that they not only contain the executable code, but also contextual data, such as the *object* of the method, or the *captured variables* of the closure. Therefore, when exposing a Mu function to a C-callable function pointer, Mu also allows the client to attach a 'cookie' — a 64-bit integer constant — behind the function pointer which can be looked up in the body of the Mu function, as depicted in 5.8(a). The cookie can be used to identify the context object in a client-specific way, such as looking up a table. This mechanism can be efficiently implemented as a simple machine instruction sequence that loads the cookie and jumps to the Mu function, as shown in 5.8(b). As the stub is simple and small, the Mu implementation can bulk-allocate arrays of such stubs. Therefore Mu functions can be frequently exposed and unexposed at run time with little overhead.

The choice of the 64-bit integer type for cookies is somewhat arbitrary, but it has enough bits to hold a pointer and has the same behaviour across platforms. Alternatively, the cookie could be defined as an object reference since managed languages are more likely to represent the context object as a heap object. However, it creates a challenge for the GC which has to update the reference atomically when the object

[20]For example, to implement the `seq_cst` memory order on x64, a compiler can implement load as `MFENCE, MOV (from memory)` and store as `MOV (to memory)`, or implement load as `MOV (from memory)` and store as `MOV (to memory), MFENCE`. Both implementations comply to the C11 semantics, but all components must agree in order to work together properly.

is moved, because the native program is non-cooperative, and may call the exposed function without regard to the GC. Some architectures, such as ARMv7, do not support pointer-sized literals, and a register needs to be initialised in two consecutive instructions. Atomically patching two instructions is hard if possible at all. If the register is loaded from memory, it will have no advantage over letting the client load from a global array using an integer-typed cookie. Alternatively, the cookie can be defined as a pointer-sized integer, which requires the client to emit different code for different platforms.

5.3.8 Intrinsics

With the design of Mu continuing to evolve, more and more primitive operations, such as object pinning, are added to the Mu instruction set. These primitive operations cannot be expressed with existing instructions, but inventing a new instruction for every new operation will cause the instruction set to explode.

Mu has a mechanism — *intrinsics*[21] — which allows the instruction set to be extended without inventing new instruction formats. All intrinsics are encoded in a common format: '`%rv = INTRINSIC @name <@T1 @T2 ...> (%arg1 %arg2 ...)`'. New intrinsics can be added by simply inventing new names.

This mechanism is similar to intrinsic functions in C and LLVM. However, unlike intrinsic functions provided by C implementations, Mu intrinsics are fully standardised by the Mu specification, and understood by all Mu implementations. Unlike LLVM, Mu intrinsics have a richer syntax than function calls. For example, Mu intrinsics may accept type arguments in addition to value arguments, making type-polymorphic intrinsics easy to encode, while LLVM intrinsic functions have to encode the type arguments as part of the function name , such as the `llvm.sqrt.f32` and `llvm.sqrt.f64` intrinsic functions for different floating point types, which may be ugly when the types are complex.

All functions in the Mu client API are also available as intrinsics. This allows meta-circular clients, i.e. Mu clients which are themselves Mu IR programs, to invoke the API via intrinsics without resorting to the native interface.

5.4 Summary

This chapter presented the Mu intermediate representation (IR). The Mu IR is designed using LLVM as a frame of reference. The Mu type system is low-level, but has built-in support for garbage-collected reference types. The Mu instruction set addresses the major concerns of Mu, namely execution, concurrency and garbage collection, and also allows direct interaction with native programs.

In the next chapter, we will introduce the Mu client interface, the API via which the client communicates with Mu.

[21]The current Mu specification calls them 'common instructions' because all such instructions have a common encoding. This name is misleading because these instructions are much less common than other instructions, such as `ADD` and `SUB`. We plan to change this name to 'intrinsics'.

```
.global @CookieToObjectMap <@refToHashMap>

.funcdef @foo VERSION %1 <@foo.sig> {
    %entry():
        %v       = INTRINSIC @uvm.native.get_cookie
        %ctx_obj = ??? // TODO: get the object from %v
        // ...
}

.const @MY_COOKIE1 <@i64> = 100
.const @MY_COOKIE2 <@i64> = 200

.expose @nativefoo1 = @foo #DEFAULT @MY_COOKIE1
.expose @nativefoo2 = @foo #DEFAULT @MY_COOKIE2
```

(a) Mu IR

```
foo:
    // prologue
    push    rbp
    mov     rbp, rsp

    // cookie is in rax
    // foo body continues here

nativefoo1:
    mov     rax, 100    // load the cookie value
    jmp     foo         // jump to the actual foo

nativefoo2:
    mov     rax, 200
    jmp     foo
```

(b) x64 Assembly

Figure 5.8: Cookies of exposed functions. Figure 5.8(a) shows a Mu function `@foo` which is exposed to two function pointers `@nativefoo1` and `@nativefoo2`, with the same default C calling convention, but different cookies. When the native program calls either exposed function, `@foo` will be executed, but the value of `%v` will be 100 if called via `@nativefoo1`, and 200 if called via `@nativefoo2`. This value can be used to to lookup the context object (such as the object of the method) using certain client-designed global map data structure. Figure 5.8(b) shows one possible implementation on x64. The register `rax` is reserved for the cookie as it is not used by the C calling convention. The exposed functions load the literal value 100 or 200 into `rax` before jumping to the actual `@foo`, where the cookie is available in the `rax` register. Mu can bulk-allocate arrays of such `mov-jmp` sequences and reuse them when Mu functions are exposed and unexposed at run time.

Chapter 6

Mu's Client Interface

The preceding chapter presented the Mu intermediate representation for programs executed on Mu. This chapter presents the Mu client interface (API), which allows the language client to control Mu and handle trap events at run time.

This chapter is structured around the use of the Mu API. Section 6.1 presents a high-level overview of the API design; Section 6.2 discusses the API for the run-time loading of Mu bundles; Section 6.3 discusses the API for trap handling and run-time optimisation; Section 6.4 summarises this chapter.

6.1 Overview

Mu provides a bi-directional API to communicate with its client. The client can send messages to Mu for the purposes of: (1) building and loading Mu IR code bundles, (2) accessing Mu memory, and (3) introspecting and manipulating the state of Mu threads and stacks. Mu sends messages to the client if a `TRAP` or `WATCHPOINT` instruction is executed.

The client API is expressed in the specification in the form of a header in the C language. This makes C the canonical language for the interface between Mu and the client, but language bindings for other languages can be created.

The client API is different from the 'unsafe native interface' introduced in Section 5.3.7. The purpose of the native interface is interacting with native libraries, while the purpose of this API is the communication between the micro virtual machine and the client.

How tightly the client is coupled with Mu is not specified. The client may be a meta-circular client which itself is a Mu IR program. The client may be a C program running in the same process as Mu. It can also be running in a different process, or a different machine, that controls the micro virtual machine remotely.

Like JNI, the API lets the client hold Mu values, including traced references, via opaque handles tracked by Mu. This hides the representation of Mu values, especially opaque reference types, from the client.

The API can create many *client contexts*. Each context is an entity in Mu that holds handles on behalf of the client, and the garbage collector may trace all references held by all contexts in order to perform exact GC. The context also lets the client allocate objects in the Mu memory, access the Mu memory (including the heap), and create other objects such as Mu threads and stacks. For efficient implementation, the context is intentionally *not thread-safe*. Each context should be used by at most one client thread at a time, allowing operations on the context to be implemented without excessive synchronisation. For example, each context might have its own heap allocation buffer, allowing the memory allocation fast-path to avoid taking a lock on the entire heap.

The states held by client contexts are similar to those held by Mu threads. A Mu thread holds many Mu values as local variables on the stack, whereas a client context holds many Mu values as handles. Both Mu threads and client contexts may have local GC allocation buffers. In some sense, a client context enables a client thread to perform operations that otherwise could only be performed by Mu threads.

6.2 Bundle Building and Loading

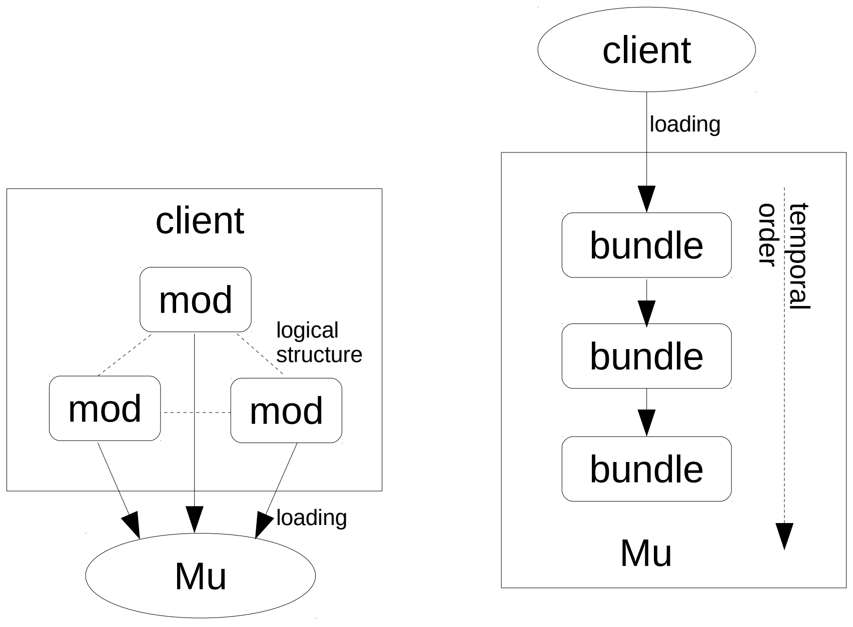
The Mu IR program which the micro virtual machine compiles and executes comes from the client.

6.2.1 Bundle as the Unit of Loading

The unit of Mu IR code loading is *bundle*. A bundle is the counterpart of a JVM `.class` file or an LLVM module. As shown in Figure 5.1, a Mu IR bundle contains many top-level definitions, which are types, function signatures, constants, static cells, functions, function versions and exposed functions. The client constructs and submits the Mu IR code bundles to Mu via the API.

Conceptually, a Mu instance has one global bundle which is initially empty. Every time the client loads a bundle, all top-level definitions are merged into the global bundle. Therefore, at any time, the global bundle contains all of the top-level definitions, such as types and functions, that the client has ever submitted to Mu. Although Mu may implement parallel bundle loading, the Mu specification requires that the loading of all bundles to be serialised, so that all bundles appear to be loaded in one particular order. Therefore, the code in each bundle may only use the top-level definitions, such as types and functions, defined in the current bundle or any previously loaded bundle, but not from bundles loaded in the future.

This model is very different from the C programming language, where all source files are 'parallel' — they are compiled independently and linked together, and each file can still refer to symbols defined elsewhere and the linker resolves the inter-dependency. C and Java programmers may find the Mu bundle loading model counter-intuitive, thinking that Mu bundles should mirror the high-level C source codes or JVM `.class` files. However, if we think from the perspective of Mu, the design is logical. As shown in Figure 6.1, when we observe from Mu's perspective, the exact organisation of language-level modules is its implementation detail which Mu is oblivious of. The only relation between bundles is the order in which they are loaded from the unknown outside world called the client. In this way, we see the process of



(a) The client's perspective (b) Mu's perspective

Figure 6.1: Bundle loading from different perspectives. If we focus on the client as in Figure 6.1(a), the modules should represent the structure of the high-level program, and Mu is just a destination of those modules. But if we focus on Mu as in Figure 6.1(b), then all details inside the client are beyond the concern of Mu. Mu only sees many bundles loaded from the client one after another, and the only relation between bundles is the temporal order of loading.

loading one bundle after another as the process in which Mu gradually gains more knowledge about the program that the client intends to execute. Naturally, to simplify the implementation of Mu, we require each bundle to only refer to knowledge, i.e. top-level definitions, which Mu has already gained (in previously loaded bundles) or is about to gain (in the current bundle), and does not require Mu to keep note of unresolved top-level definitions.

A bundle is the unit of loading. It does not need to match the logical module of the language the client is implementing. A bundle may be as small as a single function the client has just optimised. A bundle may also be as big as the amalgamation of several inter-dependent modules. There is not restriction in size, so we expect the client to build and load a bundle whenever it has the need to submit any code.

6.2.2 The IR-building API

The client builds Mu IR bundles using the API.

The IR-building API contains many functions, each creating an AST node of Mu

IR inside Mu. The client can order Mu to load the bundle when it is completely built, or abort the IR-building process at any time. The `irbuilderref` opaque reference type refers to an *IR builder* object which holds temporary AST nodes while building a bundle.

AST nodes refer to each other by symbolic IDs rather than direct pointers. The Mu IR inevitably contains cyclic references between nodes. For example, a basic block refers to a list of instructions, and the `BRANCH` instruction refers to a basic block. Functional languages may have difficulties handling cyclic references, and this general design makes the API itself (and Mu itself) implementable in both imperative and functional languages.

The purpose of the IR-building API is to communicate Mu IR code between the client and Mu as efficiently as possible. It is not intended to be used as a data structure for the client to perform code transformation, which happens to be the purpose of the LLVM IR. Adapting our API for transformation will greatly complicate the API by adding more functions for modification and query, which will increase the burden on Mu which should be kept minimalist. To compensate this, client side libraries can be developed for the client to perform Mu-IR-to-Mu-IR optimisations.

6.3 Trap Handling and Run-time Optimisation

During execution, a program may encounter events which Mu cannot handle alone. One example is lazy code loading. The client may only translate parts of the user-level program into Mu IR, and lazily load other components, such as classes or functions, only when they are first used. Another example is profiling-based optimisation. The client may detect hot (frequently executed) functions at run time and apply aggressive optimisations on these 'hot spots'.

Mu provides a trap handling mechanism to handle such events.

6.3.1 Trap Handling

Traps give clients the opportunity to introspect execution state to adapt and optimise the running program. A Mu IR thread can temporarily pause execution and transfer control to the client by executing a `TRAP` instruction: '`%ret_val = [%trap_name] TRAP <@RetTy>`'.

The `WATCHPOINT` instruction is a conditional variant of `TRAP` which is disabled in the common case but can be enabled asynchronously by the client. A `WATCHPOINT` is particularly useful for invalidating speculatively optimised code. For example, a Java implementation can compile virtual functions as non-virtual and replace all virtual call sites with a direct call if they are not overridden by any known subclasses. However, when a new class is loaded and it overrides the virtual function, all previously compiled call sites become invalid, and need to be invalidated. According to Lin et al. [2015], `WATCHPOINT` can be implemented with code patching to have near-zero cost when disabled.

`TRAP` and `WATCHPOINT` pauses the execution by unbinding the current thread from the stack using the same mechanism as `SWAPSTACK`. Conceptually, trap handling uses SWAPSTACK to switching the context (stack) of the current thread to the context of the client where a *trap handler* — a call-back function registered by the client — is executed. When the trap handler returns, it may rebinds the thread to a Mu stack and continue execution. In Section 7.3.4, we will explain how SWAPSTACK can help creating a consistent API for stack-related operations.

6.3.2 Function Redefinition

The client may perform run time optimisation within a trap handlers. For performance reasons, the client may use a quick and dirty baseline compiler to compile most functions, and only activate the expensive optimising compiler for 'hot' functions. When the optimisation is done, the client wishes to replace the existing Mu IR function with a new version.

In Mu, a *function* is merely a callable entity, while every *function version* is a block of concrete code. For C programmers, a Mu function is like a C function declaration, while a Mu function version is like a C function definition. As shown in Figure 5.1, a function version definition '`.funcdef`' has both a function name and a `VERSION` name. When loading a new Mu IR bundle, if there is a new version for a pre-existing function, it will add a new version to that function. We also say that the new version *redefines* the function. Since the loading of bundles are linearised as we described in Section 6.2.1, the order of bundle loading determines which version is newer.

When calling a Mu function with a `CALL` instruction, it executes the newest version of that function observed by the current thread. *Conceptually*, all Mu functions are indirected. A function can be thought of as a memory location that holds a pointer to the code of its current version. Defining or redefining a function is like storing a code pointer to that location, while calling a function is like loading the code pointer from that location and then calling it. Function redefinition is always atomic with respect to invocation, which means an invocation observes either the old version or the new version, but never any partially updated function. This also means that Mu thread may not always see the globally newest version of a function, because it may be concurrently redefined by another thread. To create a consistent semantics for function redefinition in multi-threaded scenarios, we use the happens-before relation (see Section 5.3.5) from the Mu concurrency memory model for the visibility of function versions. For example, if a redefinition of a function happens before an invocation of the function, and there are no other redefinitions of that function, then that invocation is guaranteed to see the newer version.

Although Mu functions are *conceptually* indirected, it is possible for Mu instances to implement function redefinition without indirection. When a function is redefined, Mu can compile the new version and patch the entry point of all old versions with a `JMP` instruction to the new version. Remember that the `funcref` type is considered a traced reference. With the garbage collector's help, it can update (forward) the existing function reference values to the address of the newest version in the same way the copying garbage collector forwards object references when objects are moved. Once forwarded, all function calls are direct.

Function redefinition only affects future invocations. When a function is redefined, future invocations of the function will execute the new version, but there may be existing activations on the stack of some threads. Mu cannot automatically transform the execution context of the old version to the context of the new version, because Mu does not understand the semantics of the high-level language. It is also unsound to abruptly terminate the execution of the old version. Therefore, we require that when a function is redefined, all existing activations continue executing from their current position of code in their current versions. This means Mu may need to retain several versions of a function at the same time. Only when all activations of the old version have returned can Mu recycle the space occupied by the old version. To let the client transform the execution context of a function to the newer version in the case when the old version needs to be invalidated, Mu provides the `WATCHPOINT` instruction and a powerful on-stack replacement (OSR) mechanism, which will be introduced later. Mu also provides an API to introspect the stack frames, and see which version of the function a stack frame is currently executing, in order to help profiling and on-stack replacement (see Section 7.3.5).

Mu makes no attempt to verify whether the old version and the new version of functions are algorithmically equivalent. The optimiser belongs to the client, and it is its responsibility to preserve the semantics of the high-level language between optimisation. Mu merely guarantees the semantics of function redefinition as described above. Thus, the client can redefine a factorial function to calculate Fibonacci number, and Mu will loyally make subsequent calls execute the Fibonacci function.

Mu functions may have no version, i.e. the function is declared but not defined. This is useful for the stubs of functions that have not been loaded due to lazy loading. Such undefined functions are still callable. When called, they behave like a `TRAP` instruction, and let the client trap handler handle this case. Naturally the client can JIT-compile the lazily loaded function to Mu IR, and then re-run that function.

6.3.3 Stack Operations

Advanced language implementations perform aggressive optimisations using the execution state of the program, such as the current program counter and the local variable values. This requires introspecting the state of stack frames. After optimisation, the runtime will replace the current execution state with the optimised state by replacing the stack frames on the stack, i.e. on-stack replacement (OSR). These mechanisms need to be supported by the low-level execution engine. In a Mu-based language implementation, Mu is the abstraction layer for compilation and execution, and therefore should provide abstraction over OSR.

As we will discuss in greater detail in Section 7.3, the API for stack operation consists of three main parts.

1. The API provides *frame cursors* which iterate through the frames of unbound stacks, allowing the client to access the frames.

2. For supporting stack introspection, there are API functions that report the current function, the current function version, and the current Mu IR instruction of a frame given by a frame cursor. The Mu IR lets the client select a set of local variables at call sites or SWAPSTACK sites; the values of those variables can also be obtained using the API.

3. For supporting on-stack replacement, the API also lets the client replace existing frames with new frames, using a technique inspired by return-oriented programming (ROP) from cyber-security.

Similar to function redefinition, Mu makes no attempt to verify whether the old frames and the new frames are functionally equivalent.

In addition to optimisation and deoptimisation, The client may use this API for debugging purpose, letting the programmer hot-patch a running program with new code, and fix bugs without restarting the program.

6.4 Summary

This chapter presented the Mu client interface, i.e. the API. The API allows the client to control the state of Mu, load Mu IR bundles at run time, and perform run-time feedback-directed optimisation.

In the next part, we will look into the topic of stack introspection and on-stack replacement, two features that distinguish Mu from other existing platforms. Since they are so important to high-performance language implementations, especially dynamic languages, we will devote a whole part to this topic.

Part II

On-stack Replacement and Its Implementation

Chapter 7

A Practical OSR API for the Client

Part I introduced the design of Mu, a thin abstraction layer over execution, concurrency and garbage collection. One important design goal of Mu is to support run-time feedback-based optimisation, which is crucial to the efficient implementation of dynamic languages. This part introduces on-stack replacement (OSR), an important VM mechanisms for supporting run-time feedback-based optimisation. This chapter discusses how a well-designed OSR API can facilitate the construction of a virtual machine, and the next chapter will discuss how the Mu stack API can be efficiently implemented on concrete hardware.

This chapter is structured as follows. Section 7.1 recapitulates the background of on-stack replacement and related concepts; Section 7.2 presents a case study of two high-performance JavaScript engines, namely SpiderMonkey and V8, which use assembly code excessively and implement OSR from scratch; Section 7.3 describes Mu's platform-independent OSR API in details; Section 7.4 demonstrates the OSR API using an experimental JavaScript Client. Section 7.5 summarises this chapter.

The work described in this part is presented in 'Hop, Skip, & Jump: Practical On-Stack Replacement for a Cross-Platform Language-Neutral VM' [Wang et al., 2018].

7.1 Background of On-stack Replacement

This section will introduce the concept of *on-stack replacement* and the related concept *stack introspection*.

In Section 5.3.6, we emphasised that Mu clearly distinguishes between threads and stacks because the difference is particularly important for on-stack replacement. A thread is a flow of control, i.e. an executor, while a stack is the execution context. A stack contains many frames, recording the contexts of nested function calls.

Normally, frames are pushed when calling a function, and popped when returning. However, high-performance language implementations need to manipulate the stack in unconventional ways as part of their optimisation and de-optimisation processes.

As we mentioned in Section 2.2.1, run-time optimisation is usually *feedback-directed*—it detects frequently executed functions and loops using run-time profiling, and uses profile information to guide the optimisation of those hot functions. For example, when a counter at a loop back-edge triggers re-compilation, the optimiser recompiles the function into a new, optimised, version of the function. The thread must then resume execution of the new code at the logically equivalent point, with a corresponding (possibly new) frame configured accordingly. Moreover, there may be other activations of the function with active frames spread across multiple stacks and active at different locations. Replacement of code for active functions in this way is known as *on-stack replacement* (OSR).

Traditionally, 'on-stack replacement' has been used to refer to the entire process of getting execution states from stack frames, mapping the old execution context to the equivalent context of the newly compiled function, removing old frames, and creating new frames. In this work, we use 'on-stack replacement' more narrowly to refer to the removal and replacement of stack frames, distinguishing it from the related but distinct task of getting execution states from frames which we refer to as '*stack introspection*'. We consider stack introspection as an orthogonal mechanism to OSR, because it can be used for purposes other than run-time re-compilation, such as recording a stack trace for exception handling, and performing a security check by examining the call stack. Although this chapter focuses on OSR, we will also introduce stack introspection mechanisms in Mu for completeness.

Stack introspection and on-stack replacement are important in order to implement languages efficiently, but they themselves are difficult to implement due to the low-level details they involve. With the program JIT-compiled into machine code, the layout of the stack is managed by the machine code, which are ultimately generated by the JIT compiler of the language implementation. This means the stack manipulator not only involves machine-specific details, but also must be helped by that JIT compiler. Implementing these operations from scratch requires the developer to be an expert at both high-level optimisation and low-level code generation and assembly programming.

In the next section, we will use SpiderMonkey and V8 as examples to see a little of the difficulties of their implementation.

7.2 Case Study of Two Real-world JavaScript Runtimes

In this section, we take a look at the OSR implementations of two real-world JavaScript engines — SpiderMonkey and V8 — and identify the challenges of implementing such a low-level mechanism from scratch.

We start with a brief overview of these two virtual machines before looking into their implementations of on-stack replacement.

7.2.1 Overview of SpiderMonkey and V8

SpiderMonkey [Mozilla] and V8 [Google] are the JavaScript Engines of the open-source Mozilla Firefox and the Google Chrome browsers, respectively. Although they are not as sophisticated as meta-circular research VMs such as JikesRVM, they represent the state-of-the-art implementation strategies for dynamic languages. Both implementations are heavily tuned and tested in real-world applications. Despite being competitors in the browser market, SpiderMonkey and V8 exhibit many striking similarities in their designs, although there is no code shared between the two projects.

Both SpiderMonkey and V8 can be classified as *monolithic VMs*. In either VM, all components are written in one C++ project from scratch, from the high-level JavaScript parser to the layout of standard JS object types, the run-time libraries, the garbage collector, the interpreter and/or the JIT compiler, the optimiser, all the way down to the machine code generator. Both engines feature generational copying garbage collectors which are capable of exact garbage collection.

Because of the dynamic nature of JavaScript, both engines use two tiers of JIT compilers. The baseline compiler is a template compiler focusing on compilation speed, while the optimising compiler aggressively optimises hot code, performing specialisation to elide dynamic type checking, and also performing conventional optimisations such as constant folding and inlining. SpiderMonkey also has a bytecode interpreter in addition to the baseline JIT. The transition between execution modes is driven by run-time feedback. The runtime switches to a higher tier when it detects frequently executed code, and falls back to a lower tier when the speculatively optimised code becomes invalid.

Stack introspection and on-stack replacement happen during execution mode transitions. In both SpiderMonkey and V8, the baseline compiler inserts profiling counters at function prologues/epilogues and loop headers/back-edges. When optimisation is triggered, the runtime introspects the existing baseline frame to recover the JS-level execution states. The optimising compiler then transforms the JS program through several levels of intermediate representations, performing aggressive optimisations on each level. Then the runtime performs on-stack replacement to remove the baseline frame and replace it with a frame for the optimised version. Similarly, when the speculatively optimised code is no longer valid, such as when a variable no longer holds values of the speculated type, the program must fall back to the baseline-compiled code which can handle all possible data types. The runtime needs to introspect optimised frames to generate the equivalent baseline frames, and uses OSR to replace the actual frames on the stack.

However, the two engines share a common problem, namely the excessive use of assembly. We now take a close look at this problem.

7.2.2 Excessive Use of Assembly for OSR

Since the VMs themselves are implemented in C++, most of the runtime libraries are written in C++, too. However, quite a few fundamental library functions are implemented manually in assembly language. One important use of assembly is to precisely control the layout of stacks during on-stack replacement.

During de-optimisation, V8 first generates the contents of new stack frames in temporary buffers in the heap. And then, an assembly routine, as shown in Fig-

```
1  #define __ masm()->
2
3  void Deoptimizer::TableEntryGenerator::Generate() {
4    GeneratePrologue();
5    const int kNumberOfRegisters = Register::kNumRegisters;
6
7    // MORE CODE HERE ...
8
9    // Replace the current frame with the output frames.
10   Label outer_push_loop, inner_push_loop,
11       outer_loop_header, inner_loop_header;
12   // Outer loop state: rax = current FrameDescription**, rdx = one past the
13   // last FrameDescription**.
14   __ movl(rdx, Operand(rax, Deoptimizer::output_count_offset()));
15   __ movp(rax, Operand(rax, Deoptimizer::output_offset()));
16   __ leap(rdx, Operand(rax, rdx, times_pointer_size, 0));
17   __ jmp(&outer_loop_header);
18   __ bind(&outer_push_loop);
19   // Inner loop state: rbx = current FrameDescription*, rcx = loop index.
20   __ movp(rbx, Operand(rax, 0));
21   __ movp(rcx, Operand(rbx, FrameDescription::frame_size_offset()));
22   __ jmp(&inner_loop_header);
23   __ bind(&inner_push_loop);
24   __ subp(rcx, Immediate(sizeof(intptr_t)));
25   __ Push(Operand(rbx, rcx, times_1, FrameDescription::frame_content_offset()));
26   __ bind(&inner_loop_header);
27   __ testp(rcx, rcx);
28   __ j(not_zero, &inner_push_loop);
29   __ addp(rax, Immediate(kPointerSize));
30   __ bind(&outer_loop_header);
31   __ cmpp(rax, rdx);
32   __ j(below, &outer_push_loop);
33
34   // MORE CODE HERE ...
35
36   __ InitializeRootRegister();
37   __ ret(0);
38 }
```

Figure 7.1: Excerpt of the `Deoptimizer::TableEntryGenerator::Generate` function for x64 in the V8 head revision (`01590d660d6c8602b616a82816c4aea2a251be63`) at the time of writing. When this routine is invoked, V8 has already generated the baseline frames in temporary buffers, and each `FrameDescription` object contains the size and the content of a frame. This snippet is a two-level loop. The outer loop iterates through each `FrameDescription`, and the inner loop reads the content of the frame word by word and pushes it onto the current stack. Line 25 uses the `PUSH` instruction which modifies the stack pointer. Since the stack pointer is constantly changing, all other operands, including the loop counters and the pointer to `FrameDescription`, must be held in registers and cannot be spilled onto the stack.

ure 7.1, uses a rather complicated two-level loop to copy the frame contents from the buffers to the actual stack. Since the stack pointer is constantly changing, the programmer must avoid any SP-relative addressing and carefully manage the register use in assembly. This two-level loop cannot be written in C++ because the C++ compiler does not expect the stack pointer to be modified outside the generated code. Similarly, SpiderMonkey also copies stack contents from a side buffer using assembly code.[1] Assembly-based on-stack replacement is also performed for the transition from baseline to optimised code during optimisation.

The use of assembly increases the engineering complexity. Writing assembly code of such complexity while properly managing registers and the control flow manually is tedious and error-prone. Moreover, the assembly routines have to be written for each supported architecture.

This problem could have been avoided if the virtual machine used the Swapstack mechanism introduced by Dolan et al. [2013]. When a thread is modifying a stack which the thread is *not* executing on, the victim stack can be treated like binary data, and modified without worrying about the stack pointer of the *current* thread. Therefore, the OSR routines can be implemented in any high-level language as long as it can access the memory. However, it appears that both SpiderMonkey and V8 have only one stack per thread. The reason why neither of them used the Swapstack mechanism is unknown, but one possible explanation is that the JIT compiler and the OSR mechanism of SpiderMonkey and V8 predate the work of Dolan et al. [2013]. Without Swapstack, the runtime has to handle stack operations with care. Since both SpiderMonkey and V8 are monolithic virtual machines, the burden is imposed on the language implementers who also develop the VM.

7.2.3 Conclusion: SpiderMonkey and V8 Depends on Assembly

As we can see, SpiderMonkey and V8 are quite similar in many aspects. Both of them use JIT compilation and aggressive optimisation. Thus they need *stack introspection on-stack replacement* as low-level tools. Unfortunately, both of them rely excessively on assembly, i.e. , too much of the program has to be implemented in assembly language. The main reason is the way these VMs are implemented.

Monolithic VM implementation Since both VMs are written from scratch, the developers have to implement both the high-level optimisation algorithm and the low-level code generators by themselves. Naturally the stack frame decoder and OSR also belong to the low level which they have to develop as part of the monolithic VM.

Operating on the same stack The necessity of assembly arises from the fact that the de-optimiser runs on the same stack as the stack it is de-optimising. Modifying an active thread's own stack is dangerous — it may interfere with the execution of

[1]See the `MacroAssembler::generateBailoutTail` function in the file `js/src/jit/-MacroAssembler.cpp` in the current Mercurial revision `65b0ac174753` at the time of writing. URL: https://hg.mozilla.org/mozilla-central/file/65b0ac174753/js/src/jit/MacroAssembler.cpp#l1429.

the current function if not done with great care. If the sizes of the stack frames grow, the replacement would not be possible without overwriting the top frame which the current function depends on.

The reliance on assembly complicates the VM development, and will multiply the engineering difficulty when more platforms are to be supported. In the next section, we will show how the Mu API can simplify the implementation of on-stack replacement.

7.3 An API for Stack Operations

In this section, we describe our platform-independent API for stack operations, capable of supporting on-stack replacement. In Section 7.4 we will demonstrate usage of the API through its application in a minimal JavaScript implementation.

7.3.1 Overview

As we learned in the previous section, two design choices complicate the implementation. (1) As monolithic VMs, the language developers have to write their own stack decoders; (2) Because the optimiser and de-optimiser run on the same stack as the JS application, the replacement of the stack frames must be handled in an assembly routine. These difficulties are countered by the following two design choices of Mu:

1. Mu abstracts out the details of execution, and provides stack introspection and OSR as part of its API;
2. Any program (including the client) which introspects or manipulates any stack must run on a different stack.

The latter is supported by the Swapstack primitive introduced in Section 5.3.6. To perform OSR, the stack must first be unbound from the thread by performing a Swapstack operation, such as executing the `TRAP` Mu IR instruction.

We list relevant Mu API functions in Figure 7.2. The client uses the *frame cursor* API to iterate through stack frames, and uses the *stack introspection* API to get the execution context which guides the optimisation or de-optimisation. After the client generates a new function, it uses the OSR API to replace stack frames. The client then lets the program continue from the new frame with another Swapstack operation, allowing it to return from the trap handler.

The usage can be summarised as 'hop, skip and jump'—hopping away from the stack, skipping several frames to create new frames, then jumping back to the stack.

Before introducing the instructions and the API, we present an abstract view of the stack that forms the foundation of the API.

Kind	API Function and Description
Frame Cursor	`FrameCursor* new_cursor(Stack* stack)` Create a new frame cursor pointing to the top frame of the given stack.
	`void next_frame(FrameCursor* cursor)` Move the frame cursor to the next frame, moving down the stack from called to caller.
	`void close_cursor(FrameCursor* cursor)` Destroy the cursor.
Introspection	`int cur_func(FrameCursor* cursor)` Return the function ID of the current frame.
	`int cur_func_ver(FrameCursor* cursor)` Return the function version ID of the current frame.
	`int cur_inst(FrameCursor* cursor)` Return the instruction ID of the current frame.
	`void dump_keepalives(FrameCursor* cursor, MuValue values[])` Dump the values of all keep-alive variables of the current instruction of the current frame.
OSR	`void pop_frames_to(FrameCursor* cursor)` Remove all frames above the current frame of the given frame cursor.
	`void push_frame(FrameCursor* cursor, void (*func)())` Create a new frame on the top of the stack pointed by the frame cursor.

Figure 7.2: Summary of Mu API functions related to stack introspection and OSR.

7.3.2 Abstract View of Stack Frames

In this thesis we adopt the convention of stacks growing up. The 'top' frame is the most recently pushed frame, and is near the top of the page in diagrams.

A stack consists of one or more frames. A frame contains the state of a function activation. A frame is *active* if it is the top frame of a stack bound to a thread. Otherwise, the frame is *inactive* because the code using it is not being executed. Specifically, if one function calls another function, the frame of the caller becomes inactive, expecting a value from the callee as a return value.

An inactive frame can receive a value of an expected type, and become active again. Specifically, when a function returns, its caller's frame receives the return value from the callee, and continues execution. Therefore, every inactive frame is *expecting a value*, and will eventually *return a value* to its caller. Symbolically, we write

$$\mathit{frm} : (E) \to (R)$$

to denote that the frame *frm* is expecting a value of type E in order to resume, and itself returns a value of type R. Because Mu functions may return multiple values (see Section 5.3.3), we can generalise this to multiple return values, writing:

$$\mathit{frm} : (E_1, E_2, \ldots) \to (R_1, R_2, \ldots)$$

We call this the *expect/return type notation*.

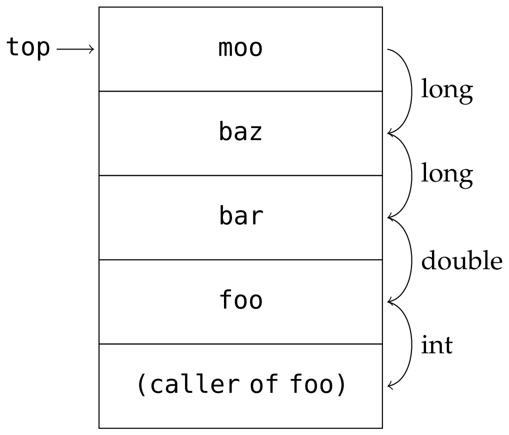
```
long moo();

long baz() {
    long x = moo();   // stop here
    return x;
}

double bar() {
    long x = baz();   // stop here
    double y = (double)(x + 1);
    return y;
}

int foo() {
    double y = bar(); // stop here
    int z = printf("%lf\n", y);
    return z;
}
```

(a) Example Code



(b) Stack Structure

$$baz : (\texttt{long}) \to (\texttt{long})$$
$$bar : (\texttt{long}) \to (\texttt{double})$$
$$foo : (\texttt{double}) \to (\texttt{int})$$

(c) Expected and Return Types

Figure 7.3: Example of nested calls. The expected types are determined by the call sites, and the return types are determined by the functions' return types. The return type of each frame must match the expected type of its caller's frame.

For example, in Figure 7.3, `foo` calls `bar`, `bar` calls `baz`, and `baz` calls `moo`. The expected type and the return type of the frames of `foo`, `bar` and `baz` appear in Figure 7.3(c) in expect/return type notation. As we can see, the expected type of a frame is determined by the call site (and the callee), and the function signature of a frame determines its return type.

A stack is *return-type consistent* if the return type(s) of every frame matches the expected type(s) of the frame below. It is obvious that if all stack frames are created by function calls, the stack is always return-type consistent.[2] However, the OSR API can create stack frames of arbitrary expected and return types. Therefore, the *client* must take care to ensure that the stack is return-type consistent at all times.

With our abstract stack view in mind, we now introduce the operations in the API.

7.3.3 Frame Cursor Abstraction

A *frame cursor* is an iterator of stack frames. A frame cursor always points to one frame at any time, and can move from top to bottom frame by frame. The API for both introspection and OSR depends on frame cursors.

7.3.4 The Swapstack Operation

A stack bound to a thread always has its top frame active and other frames inactive, because the thread executes on its top frame. As we described in Section 5.3.6, SWAPSTACK *unbinds* a thread from its stack, and *rebinds* it to another stack. SWAPSTACK deactivates the top frame of its old stack, and reactivates the top frame of the destination stack, optionally passing one or more values. After the top frame of the origin stack becomes inactive, the frame waits for another SWAPSTACK operation to bind a thread (any thread, not necessarily the original thread) to it and reactivate its top frame, optionally receiving one or more values.

It is easy to observe that an inactive frame stopping on a SWAPSTACK site is similar to an inactive frame stopping at a call site. Both of them are expecting values, and can be reactivated by receiving values. The only difference is whether the values are received by returning or SWAPSTACK. Therefore, the expect/return type notation $frm : (E) \to (R)$ is still applicable for SWAPSTACK, where E is the type of the value expected from the incoming SWAPSTACK operation.

In the Mu instruction set, the `TRAP`, `WATCHPOINT` and `SWAPSTACK` instructions perform SWAPSTACK operations, as introduced in Section 5.3. `TRAP` and `WATCHPOINT` are special SWAPSTACK operations that rebind a Mu thread to a client stack, where a client-provided *trap handler* is activated.

[2]In dynamic languages a function can return a value of any type—they do not enforce return-type consistency. However, Mu IR is statically typed. When implementing dynamic languages like Python, the Mu-level return type should be the most general type, such as `PyObject`, and all Python frames should have (`PyObject`) $\to$ (`PyObject`), which is always return-type consistent with respect to the Mu type system.

```
%call_retval = CALL <@sig> @callee (%arg1 %arg2) KEEPALIVE(%a %b %c %d)

%trap_retval = TRAP <@i32> KEEPALIVE(%e %f %g)

%wp_retval   = WATCHPOINT 1024 <@i64> %dest1() %dest2() KEEPALIVE(%h %i %j)

%ss_retval   = SWAPSTACK %new_stack
                         RET_WITH <@float>
                         PASS_VALUES <@T1 @T2> (%v1 %v2)
                         KEEPALIVE(%k %l %m %n)
```

Figure 7.4: Example of OSR point instructions. The `KEEPALIVE` clause specifies the local variables that can be introspected through the `dump_keepalive` API function. Exactly these variables can be introspected at the specific OSR point.

The Swapstack operation deactivates all frames of an unbound stack, making it ready for introspection and manipulation. All API functions related to stacks require the stack to be in the unbound (inactive) state. We now proceed to the introspection mechanisms before moving on to OSR.

7.3.5 Stack Introspection

In Section 7.1, we showed that the client needs to extract the execution state, including the program counter and the values of local variables, to guide optimisation and de-optimisation.

As shown in Figure 7.2, the `cur_func`, `cur_func_ver` and `cur_inst` API functions report the current code position of a frame.

In Mu IR, all call sites (the `CALL` instruction) and Swapstack sites (the `SWAPSTACK`, `TRAP` and `WATCHPOINT` instructions) may have a *keep-alive clause* that specifies which variables are eligible for introspection. Consider the following snippet:

```
[%trap1] TRAP <> KEEPALIVE (%v1 %v2 %v3)
```

When the `TRAP` instruction `%trap1` executes, local variables `%v1`, `%v2` and `%v3` are kept alive in the frame, and their values can be introspected using the `dump_keepalives` API function. Other local variables are not guaranteed to be live, which leaves Mu much room for machine-level optimisation. Figure 7.4 lists all 'OSR point' instructions which may have keep-alive clauses.

We let the client decide what variables are introspectable. The client, which compiled the high-level language into Mu IR, has full knowledge about what variables are relevant for the desired kind of run-time re-compilation, such as optimisation or de-optimisation. In the extreme, the client can retain all local variables, thereby preserving full information about the execution.

We now introduce the way the API allows modification of the stack.

7.3.6 Removing Frames

The `pop_frames_to` API function removes all frames above the current frame. This will expose the current frame, an inactive frame below the top frame, to the stack top. Remember that the SWAPSTACK operation passes values to the destination stack's top frame. When a thread rebinds to this stack using SWAPSTACK, it will reactivate the current frame which is stopping at a call site instead of a SWAPSTACK site. The values passed via SWAPSTACK will be received by the frame as if the values were the return values from the call site's original callee.

Popping frames will lose information about the removed frames. The client should use `dump_keepalives` to save the execution states before popping frames if needed.

The `pop_frames_to` API can only remove frames above the specified frame. If the client desires to replace a frame when the frame is re-entered, usually due to deoptimisation, the client should insert the `WATCHPOINT` instruction into the guarded function.

7.3.7 Creating New Frames Using Return-oriented Programming

The `push_frame` function pushes a frame for a given function onto the top of a stack. The frame stops at the entry point of the function.

Our approach to creating new frames is based on *return-oriented programming* (ROP).[3] We define a *ROP frame* to be a stack frame that is stopped at the entry point of a function: its return address is the entry point of the function. In contrast, the return address of a normal frame is the next instruction after a call site or SWAPSTACK site. By definition, frames created by the `push_frame` API function are ROP frames.

When a ROP frame resumes, it receives the values returned by the frame above it, or passed during a SWAPSTACK operation, as the arguments of its function which now executes from the entry point.

Consider the code snippet in Figure 7.5. If the client has pushed three ROP frames on a stack for the three functions respectively, in the order of `print`, `times_two` and then `plus_one`, then `plus_one` will be on the top of the stack and `print` at the bottom. When a subsequent SWAPSTACK reactivates the stack, passing the value 42 to the top frame, then the top frame executes as if it was a call to `plus_one(42)`. It returns 43, transferring control to an activation of `times_two` with the argument `y` receiving the value 43. This returns 86, transferring control to the activation of `print`

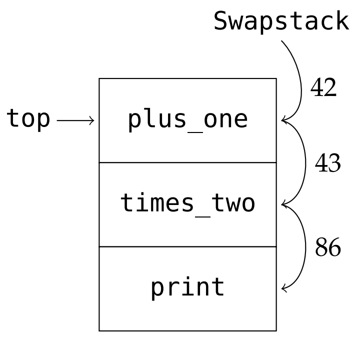
[3]ROP originates from the field of computer security to describe a particular attack technique. The attack uses malicious data to cause a buffer overflow on the stack, overwriting existing stack frames to fashion new stack frames. The return address of each frame is set to the *entry point* of the next function to execute. Therefore, after each function returns, the processor will execute the next function specified by its return address. This lets the attacker drive control flow using return instructions to transfer control to the next function, hence the name. This chain of frames is called the *ROP chain*. Prandini and Ramilli [2012] provided a more detailed overview of this attack. It is possible that future hardware will incorporate security-focused features, such as tagged memory. We believe that this may make the ROP-based OSR API harder to implement, but not impossible, because such features are intended to stop malicious use of ROP, while a micro virtual machine is a trusted underlying substrate, and will execute with adequate privilege to perform ROP in a controlled manner.

```
int plus_one(int x) {
    return x + 1;
}

int times_two(int y) {
    return y * 2;
}

void print(int z) {
    printf("%d\n", z);
    exit(0);
}
```

(a) C-like Pseudocode



(b) Stack Structure

Figure 7.5: ROP example. These three functions do not call each other. But if the run-time pushes three ROP frames for `print`, `times_two` and `plus_one`, respectively, the three functions will execute one after another from the top of the stack to the bottom (in the order of `plus_one`, `times_two` and then `print`). Each function passes its return value as the argument of the next function, making it a pipeline or, in cyber-security terms, a ROP chain.

with the formal parameter `z` bound to this value. This prints 86 to standard output and exits.

Just like frames created by function calls, ROP frames also expect values and eventually return values. We can describe the stack in the previous example in the expect/return type notation as:

$$\begin{aligned} \mathit{plus\_one} &: (\mathit{int}) \to (\mathit{int}) \\ \mathit{times\_two} &: (\mathit{int}) \to (\mathit{int}) \\ \mathit{print} &: (\mathit{int}) \to () \end{aligned}$$

Unlike frames stopped at call sites and Swapstack sites, the *parameter types* of the stopped function determine the expected types of ROP frames, as opposed to the call site. Users of the API can push frames for any functions, and as many frames as they desire, as long as the stack remains *return-type consistent* as defined in Section 7.3.2.

In Section 5.3.3, we mentioned that Mu functions may return multiple return values. In this return-oriented programming scenario, if a function takes multiple parameters, the frame above can return multiple return values, each of which are received as an argument. Otherwise we would have to force all ROP frame functions to take exactly one parameter. To make the VM design consistent, all resumption points (defined in Section 8.1.2), including call sites, Swapstack sites and function entry points, may be resumed by receiving multiple values instead of one. This is consistent with the IR design that a Mu function may return multiple values, and a Mu instruction may produce multiple results.

The pushed new frame is always a ROP frame which starts at the function entry point. However, in the typical run-time re-compilation scenario, the new frame must resume at the program counter equivalent to where the old frame stopped. Fink and

Qian [2003] solved this problem by inserting assignments and a jump instruction at the beginning of the new function, as follows:

```
  %var1 = %oldVar1
  %var2 = %oldVar2
  ...
  %varN = %oldVarN
  JUMP %cont
  ...
%cont:
  ...
```

D'Elia and Demetrescu [2016] call these instructions '*compensation code*'. The assignments set local variables of the new frame to their values in the old frame, and the jump instruction jumps to the equivalent PC. Recall that the `cur_inst` and the `dump_keepalives` API gives the client the old PC and variable values. Therefore, while the Mu API requires the new frame to start at the entry point, the client still has all the information and capability to transition the old frame state to the equivalent new state.

7.4 Demonstration of the OSR API

In this section, we demonstrate the utility of the OSR API as an aid to language implementers.

We built JS-Mu,[4] a prototype JavaScript client. It implements a small subset of JS, including operators such as the JS addition operator '+', which applies to both numbers and strings. Such dynamism gives the specialiser a chance to showcase speculative optimisation that requires OSR. We implemented JS-Mu in Scala. Examples in this section are modified to match the current version of the Mu API because Mu has evolved since JS-Mu was developed.

Like SpiderMonkey and V8, the JS-Mu execution engine consists of a baseline compiler and an optimizing compiler. There is no JS interpreter. The baseline compiler plays the role of the lowest-tier execution engine, while the optimizing compiler will optimise hot functions and loops.

7.4.1 Supported Subset of JavaScript

As a proof-of-concept project, the priority of JS-Mu is to demonstrate Mu's support for dynamic optimisation with the simplest client. We only implement a very small subset of JavaScript with the following limitations:

- The type system only includes `undefined`, `null`, `boolean`, `number`, `string` and `array`.
- We do not implement closures, thus the scope of a variable is either global or the function that defines the variable, but not nested functions.

[4]Source code: https://gitlab.anu.edu.au/mu/obsolete-js-mu

- Performing operations on incompatible types will immediately cause an error rather than implicitly converting operands as the ECMAScript specification requires.
- Only a subset of operations (operators and methods) are supported. This includes the add operator '+' that works for both numbers and strings.

7.4.2 Baseline Compiling and Trap Placement

The JS-Mu baseline compiler does not attempt to infer the concrete types of JS variables. All JS variables are represented using Mu's tagged reference type `tagref64`. All operations, such as addition, subtraction, etc., accept all types of values, and raise type errors at run time.

Like SpiderMonkey and V8, we use counters to detect hot loops at loop headers.

```
%header(...):
  ...
  %c0 = LOAD <@i64> @COUNTER
  %c1 = ADD  <@i64> %c0 @CONST_1
  STORE <@i64> @COUNTER %c1
  %hot = SGE <@i64> %c1 @THRESHOLD
  BRANCH2 %hot %trapbb(...) %body(...)

%trapbb(...):
  [%trap1] TRAP <> KEEPALIVE (%v1 %v2 ...)
```

We use Mu IR code to increment the counter, and execute the `TRAP` instruction when the counter reaches a threshold. The `KEEPALIVE` clause annotates local variables for introspection. The client maintains a simple `HashMap` to record the AST node and compiler metadata relevant to each `TRAP`.

```
class TrapInfo(val blFunc: BaselineFunction,
               val node: Node,   // AST node
               val headBB: MuBB, val trapBB: MuBB)
val trapInfoMap = new HashMap[String, TrapInfo]()
```

7.4.3 Optimisation and On-stack Replacement

The `TRAP` instruction transfers control to the trap handler.

```
def handleTrap(ctx: Context, st: MuStackRefValue, ...): TrapHandlerResult {
  val cursor = ctx.newCursor(st)
  val inst = ctx.curInst(cursor)
  val locals = ctx.dumpKeepalives(cursor)
  val trapInfo = trapInfoMap(nameOf(inst))
  ...
```

The trap handler uses `cur_inst` to identify the executed `TRAP`, and uses `dump_keepalives` to recover the current values of local variables. Using the Mu-level instruction ID and the `HashMap` described above, the optimiser finds the high level implementation (including the AST node) of the hot loop.

The main optimisation performed is specialisation, which is crucial to dynamic languages [Castanos et al., 2012]. Using the type information encoded in tagged references, the optimiser infers the types of JS variables, lowering the types to more concrete types and eliminating run-time type checking operations where possible. After specialisation, the client generates Mu IR code, and loads the code into the micro virtual machine. As we described in the end of Section 7.3.7, the optimised function takes the old local variables as parameters, and uses assignments and a jump to transfer to the equivalent code point[5] [Fink and Qian, 2003; D'Elia and Demetrescu, 2016].

After JIT compilation, the client uses `pop_frames_to` and `push_frame` to replace the stack frame.

```
...
val newFunc = compileFunction(...)
ctx.nextFrame(cursor)
ctx.popFramesTo(cursor)
ctx.pushFrame(cursor, newFunc)
ctx.closeCursor(cursor)
Rebind(st, PassValues(locals))
```

When the client returns from the trap handler, Mu uses SWAPSTACK to rebind the current thread to the old stack, passing the old values of local variables. The JS application resumes execution, executing the optimised version, continuing with the equivalent state to that at the time the optimisation was triggered.

7.4.4 Result

Figure 7.6 gives a concrete example of a simple JS function, JIT-compiled during OSR triggered at a hot loop. The OSR API provided sufficient type information that the optimised Mu IR code for the tight loop is almost equivalent to the LLVM IR code that Clang might have generated from an equivalent C program with static types.

JS-Mu is built on Holstein [Wang, a], the proof-of-concept reference implementation of Mu which is unsuitable for performance evaluation. However, the implementation is sufficient to demonstrate the completeness and correctness of the API. Note that the OSR mechanism itself is *not* performance critical, since it executes just once for each recompilation, which will be dominated (by many orders of magnitude) by the subsequent execution of the optimised code in any typical OSR setting.

7.5 Summary

This chapter discussed how a well-designed stack introspection and on-stack replacement API can facilitate the construction of a virtual machine. From two mature JavaScript VMs, namely SpiderMonkey and V8, we revealed the difficulty of implementing run-time optimisation without any abstraction for introspection and OSR.

[5]See Figure 7.6 for a concrete snippet.

```
function sum(f, t) {
    var s = 0;
    for (var i = f; i <= t; i++) {
        s = s + i;
    }
    return s;
}

var result = sum(1, 10);
print(result);
```

(a) JavaScript

```
.funcdef @optfunc1_sum VERSION @optfunc1_sum.v1 <@optfunc1_sum.sig> {
  %entry(<@tagref64> %osrParam_f <@tagref64> %osrParam_t
         <@tagref64> %osrParam_s <@tagref64> %osrParam_i):
    // Compensation code
    %raw_f          = INTRINSIC @uvm.tr64.to_fp (%osrParam_f)
    %raw_t          = INTRINSIC @uvm.tr64.to_fp (%osrParam_t)
    %raw_s          = INTRINSIC @uvm.tr64.to_fp (%osrParam_s)
    %raw_i          = INTRINSIC @uvm.tr64.to_fp (%osrParam_i)
    BRANCH %forHead(%raw_f %raw_t %raw_s %raw_i)

  %forHead(<@double> %f <@double> %t <@double> %s <@double> %i):
    %i_le_t         = FOGE  <@double> %t %i
    BRANCH2 %i_le_t %forBody(%f %t %s %i) %forEnd(%f %t %s %i)

  %forBody(<@double> %f <@double> %t <@double> %s <@double> %i):
    %s2             = FADD <@double> %s %i
    %i2             = FADD <@double> %i @CONST_1
    BRANCH %forHead(%f %t %s2 %i2)

  %forEnd(<@double> %s)
    %tagged_s       = INTRINSIC @uvm.tr64.from_fp (%s)
    RET <@tagref64> %tagged_s
}
```

(b) Mu IR

Figure 7.6: Result of JS-Mu compiling a JS program. Sub-figure (a) is a simple JS program that sums over a range. Sub-figure (b) is the optimised Mu IR code generated when optimisation is triggered at the loop header. The IR code is adjusted to match the current Mu IR design. Auto-generated variable names are simplified for readability. The `%entry` block contains compensation code, which initialises the values of local variables from parameters, and jumps to the loop header that triggered optimisation. The compensation code also removes the tags of the values to make them plain `double` type. Therefore, within the loop of `%forHead` and `%forBody`, all variables have been specialised to the `double` type, and no conversion to or from the tagged reference type (`@tagref64`) is present. This is equivalent to the LLVM IR code which Clang could have generated from an equivalent C program with static types.

We demonstrated how the Mu API can help the high-level language developers using an experimental JavaScript client.

In the next chapter, we will discuss how the Mu stack API can be efficiently implemented on concrete hardware.

Chapter 8

Implementation of the Mu Stack API

We have introduced the platform-independent OSR API. However, the API does not remove the fundamental complexity of OSR. Rather, it hides it beneath a layer of abstraction. In this chapter we turn to the question of whether such an API is realizable in a realistic setting.

This chapter is structured as follows. Section 8.1 uses the concepts of 'resumption points' and 'resumption protocols' to formulate the consistency of stack states at a lower level, and guide the correct implementation of OSR; Section 8.2 presents a proof-of-concept project that implements the Mu stack API for native C programs, and the difficulties encountered; Section 8.3 discusses other existing implementations of OSR and compares them with Mu; Section 8.4 concludes this chapter.

8.1 Resumption Points and Resumption Protocols

Our approach is to introduce the abstractions of *resumption protocols* and *resumption points*. We use the x64[1] and the AArch64[2] architectures as our concrete setting, but the ideas are not specific to those architectures.

To demonstrate how resumption protocols can guide the implementation of the OSR API in a more realistic scenario, we developed another proof-of-concept project `libyugong` that implements the SWAPSTACK operation and the OSR API for native programs (in C, C++, LLVM, etc.) which follow the platform ABI on GNU/Linux. We will discuss `libyugong` in greater detail in 8.2.

8.1.1 Frame Cursors and Introspection

For completeness, before we start discussing OSR, we briefly introduce how frame cursors and stack introspection can be implemented. Although they are integral parts of the API, they are well-developed technologies, and this thesis does not attempt to make improvements over existing approaches.

[1]Also known as AMD64, x86-64 or Intel64, an extension to the IA32 instruction set architecture.

[2]AArch64 is the 64-bit execution mode of the ARMv8 architecture. The instruction set is called A64.

The frame cursor is an abstraction over *stack unwinding*, the process of restoring register states of frames below the top frame of a stack. Key to the implementation is how to restore callee-saved registers of the caller given the program counter. C++ compilers, such as GCC, generate stack unwinding metadata in the DWARF [DWARF Standards Committee, 2010] format on GNU/Linux for exception handling. We will discuss more about stack unwinding in Section 8.2.1.

Stack introspection uses *stack maps*, a data structure that maps machine-level execution states (including stack frame contents and callee-saved register values) to high-level language states (values of local variables). As we discussed in Section 2.2.3, stack maps are required for exact garbage collection, therefore many virtual machines, such as JikesRVM [Alpern et al., 2009], already implement stack maps. LLVM also provides the '`@statepoint`' intrinsic which generates stack maps to decode frames for LLVM-level variable values.

The rest of this section assumes these techniques are readily available, and focuses on OSR on top of those techniques.

8.1.2 Resumption Point

We define a *resumption point* as the point in the function body where an inactive frame stopped. In Mu, a resumption point can be a *call site*, a *swap-stack site*, or the *entry point* of a function (ROP frame).

The resumption point is an internal execution state not visible to its neighboring frames, while the expected type and returned type are the 'interface' through which the frame communicates with the frames above and below. However, the resumption point determines the frame's resumption protocol which we now define.

8.1.3 Resumption Protocol

The concept of resumption protocol is related to calling conventions. A *calling convention* describes the rules of function calling at the machine level, including the responsibility to set up and tear down the frames, the registers and stack locations used to pass parameters and return values, and the set of registers preserved across function calls (i.e. , the callee-saved registers). The calling convention is the agreement between the caller and the callee.

However, we are more interested in the resumption of frames than the set-up of frames. We define the *resumption protocol* as the machine-level rules governing the passing of values to an inactive frame to resume its execution.

The concrete rules are determined by the resumption points. We now describe the resumption protocols of each resumption point.

Resumption at Call Sites

When a frame is stopped at a call site, the resumption protocol is the 'returning' part of the calling convention.

For example, consider a function that is suspended, having called a function that returns a 32-bit integer. On x64 on GNU/Linux, the resumption protocol is:

> 'Move the return value into register `EAX`, and then pop and jump to the return address at the top of the stack.'

The resumption protocol on AArch64 is:

> 'Move the return value into register `w0`, and then restore the program counter from the link register `x30`.'

We use '`CCC_Ret(int)`' (C Calling Convention: Returning) as the symbolic notation for the resumption protocol. We can also generalise it to '`CCC_Ret(T)`' for the protocol of returning type `T` using the C calling convention. We omit the platform name in the symbol, because the C calling convention refers to the definition in the ABI of the platform.

At a call site, the function receives the return value from the callee according to the callee's calling convention; when the function itself returns, it will pass the return value to its caller according to the function's own calling convention. Therefore, every frame has both an *expected resumption protocol* which is the resumption protocol to re-activate the frame itself, and a *returned resumption protocol* which is used to re-active its caller. We use the following notation:

$$frm : \texttt{rp}_1\,(E_1, E_2, \ldots) \rightarrow \texttt{rp}_2\,(R_1, R_2, \ldots)$$

to denote that the frame *frm* itself is resumed using the resumption protocol $rp_1\,(E_1, E_2, \ldots)$, and will re-activate its caller using the resumption protocol $rp_2\,(R_1, R_2, \ldots)$ when it returns. We call this notation the *expect/return protocol notation* because it involves not only the types but also the resumption protocols.

Consider the example in Figure 7.3. The expected and returned protocols of `baz`, `bar` and `foo` are straightforward:

$$\begin{aligned} baz &: \texttt{CCC\_Ret}\,(long) \rightarrow \texttt{CCC\_Ret}\,(long) \\ bar &: \texttt{CCC\_Ret}\,(long) \rightarrow \texttt{CCC\_Ret}\,(double) \\ foo &: \texttt{CCC\_Ret}\,(double) \rightarrow \texttt{CCC\_Ret}\,(int) \end{aligned}$$

because both `baz`, `bar` and `foo` are stopped on call sites.

We now proceed to describe the resumption protocol for Swapstack sites.

Resumption at Swapstack Sites

Recall that the Swapstack operation unbinds the thread from one stack and rebinds it to another stack. Similar to function calls, Swapstack involves preserving the context and transferring control. Currently, there are no widely-applicable standards about the implementation of Swapstack. However, when implementing Swapstack, there

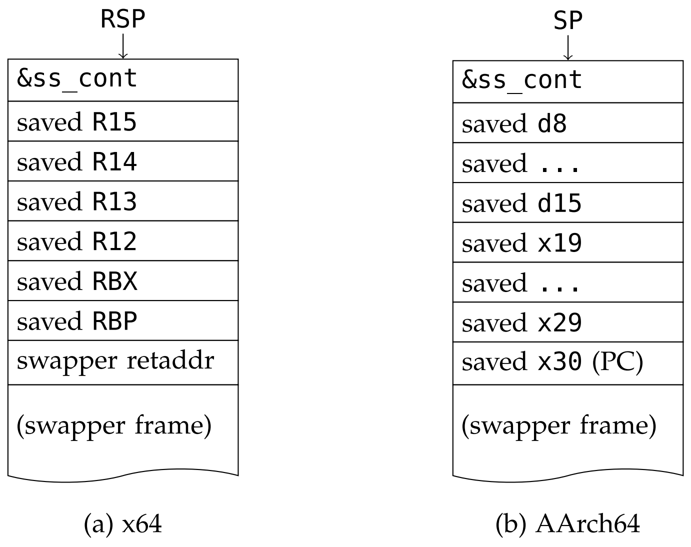


Figure 8.1: Stack-top structure of unbound stacks in `libyugong`. The callee-saved register values and the resumption point PC are all saved at the top of the stack. When rebinding a thread to an unbound stack, it continues from the address of the `ss_cont` routine which restores the callee-saved registers from the stack top, and returns to the swapper frame.

must not be any 'swappee-saved'[3] registers because it is unpredictable where the swappee stack will swap back to the swapper, or whether it will swap back from that stack at all.[4] Therefore, the SWAPSTACK operation must treat all machine registers as swapper-saved registers.

We use the symbolic notation $\texttt{SS}(T_1, T_2, \ldots)$ for the resumption protocol of a SWAPSTACK site that receives a value of type $T_1, T_2, \ldots$ when the stack is rebound.

This protocol can be implemented in many ways. In our `libyugong` library, we implemented SWAPSTACK similar to `boost-context` [Kowalke, 2016]. The SWAPSTACK implementation (See Figure 8.4) saves the execution state on the top of the unbound stack as in Figure 8.1. Therefore, resuming the swapper frame is done by restoring the callee-saved registers from the stack top structure, and jumping to the resumption point of the swapper.

Resumption at Entry Points

Remember that a ROP frame is stopped at the entry point of a function, expecting to receive values as its parameters. Therefore, the resumption point for ROP frames is the same as the 'calling' part of a calling convention.

[3]Similar to 'callee' which means the function called by a call site, we use the word 'swappee' for the destination stack in a SWAPSTACK operation. The original stack of a SWAPSTACK operation is called the 'swapper'.

[4]Intuitively, if we implement a green thread system using each stack as a light-weight task and randomly schedule the stacks to a pool of threads, then when a task yields using the SWAPSTACK operation, it is unpredictable which task will be executed next, and which task the thread is swapping from. In general, SWAPSTACK is much less predictable than call and return.

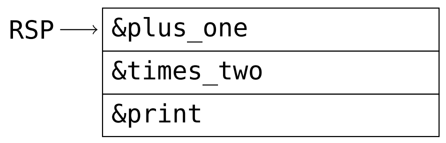


Figure 8.2: Stack structure of a naïve (and thus, incorrect) ROP frame implementation on x64 for the example in Figure 7.5. Each frame simply consists of the address of the entry point of the function. This approach will execute `plus_one`, `times_two` and `print` in order, but *will not* be able to pass return values between frames because the return values and the parameters are passed via different registers (`EAX` and `EDI`, respectively).

For example, consider a function that takes one 32-bit integer as its parameter. On x64 on GNU/Linux, the calling convention specifies that:

> 'The first 32-bit integer parameter is passed via the `EDI` register, and the return address is on the top of the stack.'

The calling convention on AArch64 is:

> 'The first 32-bit integer parameter is passed via the `w0` register, and the return address is held in the link register `x30`.'

Therefore, as long as the arguments and the return address are put in the right place, and the stack pointer is set properly, the function will start executing until it returns. We use the symbol `CCC_Entry` for the resumption protocols of ROP frames that follow the C calling convention, and `CCC_Entry(int)` denotes ROP frames that have `int` as their sole parameter.

We have introduced all three resumption points and their resumption protocols. A stack is *return-protocol consistent* if the returned protocol of every frame matches the expected protocol of the frame below.

However, it remains a question how to construct concrete ROP frames on the stack. Figure 8.2 shows a naïve ROP frame implementation on x64 for the example given in Figure 7.5. Each frame simply consists of the address of the entry point of the function, and the `RSP` register points directly at the location that holds the return address of the topmost function, `plus_one`. When the thread 'returns' by popping the return address from the stack, it will start execution from the entry point of `plus_one`. When `plus_one` returns, it will 'return' to `times_two` which will in turn start from its entry point. Eventually `print` will be executed, too.

But this naïve approach cannot pass the return value of one function to another because of the difference of register use between return values and parameters. Note that on x64, `CCC_Ret` and `CCC_Entry` expect integer values to be passed in different registers (`RAX` and `RDI`, respectively). Therefore, when the function of the next ROP frame starts, it will not find the parameter in the `EDI` register, and will not be able to receive the return value from the frame above.

Since all functions are compiled to use the `CCC_Ret` protocol for normal returning, the returned protocols of these functions are all `CCC_Ret`. Using the protocol-sensitive notation, we have:

$$\begin{aligned} \textit{plus\_one} &: \texttt{CCC\_Entry}\,(\textit{int}) \rightarrow \texttt{CCC\_Ret}\,(\textit{int}) \\ \textit{times\_two} &: \texttt{CCC\_Entry}\,(\textit{int}) \rightarrow \texttt{CCC\_Ret}\,(\textit{int}) \\ \textit{print} &: \texttt{CCC\_Entry}\,(\textit{int}) \rightarrow \texttt{CCC\_Ret}\,() \end{aligned}$$

The stack is not return-protocol consistent. The returned types match the expected types (`int` and `int`), but the resumption protocols (`CCC_Entry` and `CCC_Ret`) do not. This is the reason why this naïve approach will not work. To correctly implement ROP frames, we need adapters to convert mismatching resumption protocols.

8.1.4 Adapter Frames

An *adapter frame* sits between two frames where the *type* of the passed values match, but the *resumption protocols* do not. The adapter frame satisfies:

$$\textit{adapter} : \texttt{rp}_1\,(T_1, T_2, \ldots) \rightarrow \texttt{rp}_2\,(T_1, T_2, \ldots)$$

for some $T_1, T_2, \ldots$, that is it converts one resumption protocol to the other while preserving the value.

The following snippet implements an adapter frame on x64 of $\texttt{CCC\_Ret}(\texttt{int}) \rightarrow \texttt{CCC\_Entry}(\texttt{int})$, that is a frame that transfers the return value to the register which holds the parameter.

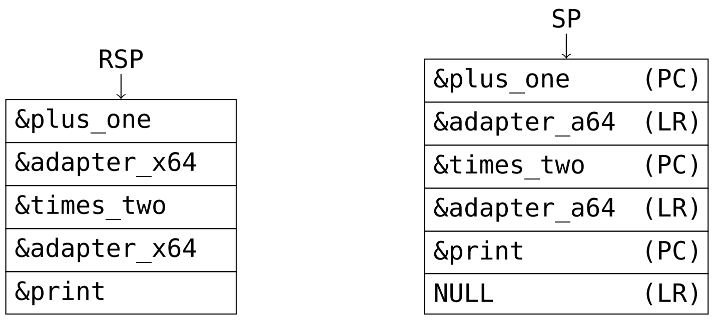
```
adapter_x64:
    MOV     EDI, EAX
    RET
```

The `MOV` instruction moves the value to the correct register, and the `RET` instruction resumes the next frame.

Figure 8.3 shows a correct implementation for the example in Figure 7.5. On x64, adapter frames are inserted between adjacent ROP frames. Each adapter frame consists of only the address of the `adapter_x64` assembly routine shown above. When `plus_one` returns, the return value is held in the `EAX` register, and the control flow jumps to the `adapter_x64`, where the `MOV` instruction moves the value from `EAX` to `EDI`. At this point, the address of the `times_two` function is at the top of the stack. Therefore, the `RET` instruction in the adapter frame resumes the `times_two` function, and the return value of `plus_one` has already been moved to the expected `EDI` register which `times_two` receives as the argument. This process will happen again when `times_two` returns, and `print` will eventually receive and print out the correct value.

AArch64 faces a different challenge. The return address of the `CCC_Entry` protocol is in the link register x30 instead of on the stack, therefore we cannot put the address of the return address on the stack and expect the ROP frame to automatically return into it. We use the adapter to restore the link register:

```
adapter_a64:
```



(a) x64 (b) AArch64

Figure 8.3: Correct ROP frame implementation with adapter frames on x64 and AArch64 for the example in Figure 7.5. On x64, addresses of functions are interleaved with addresses of the adapter frame. When one function returns, the adapter will be executed before the next function starts, giving the adapter a chance to move the return value to the argument register. On AArch64, every pair of addresses from the top are restored into the x30 (LR) and the PC registers respectively, allowing the ROP function to execute correctly.

```
    LDP     x9, x30, [sp], 16
    BR      x9
```

The above code pops the ROP function address into the PC, and pops the return address of the ROP frame into x30 (LR). It uses a scratch register x9 because we cannot directly load into the PC. After `plus_one` returns, it returns into x30 which holds the address of `adapter_a64`, which restores the next ROP frame. The last function, `print`, does not return, therefore x30 holds `NULL` when entering `print`.

Using the expect/return protocol notation, we have:

$$\begin{aligned} \mathit{plus\_one} &: \texttt{CCC\_Entry}\,(\mathit{int}) \to \texttt{CCC\_Ret}\,(\mathit{int}) \\ \mathit{adapter} &: \texttt{CCC\_Ret}\,(\mathit{int}) \to \texttt{CCC\_Entry}\,(\mathit{int}) \\ \mathit{times\_two} &: \texttt{CCC\_Entry}\,(\mathit{int}) \to \texttt{CCC\_Ret}\,(\mathit{int}) \\ \mathit{adapter} &: \texttt{CCC\_Ret}\,(\mathit{int}) \to \texttt{CCC\_Entry}\,(\mathit{int}) \\ \mathit{print} &: \texttt{CCC\_Entry}\,(\mathit{int}) \to \texttt{CCC\_Ret}\,() \end{aligned}$$

The returned protocol of each frame matches the expected protocol of the frame below, therefore the stack state is now return-protocol consistent.

Different types need different adapter frames. For example, on x64 on GNU/ Linux, both the first floating point parameter and the floating point return value are held in the `XMM0` register. Apparently no register movement is necessary. But the ABI also has a 16-byte stack alignment rule at the entry point of every function.[5] Thus a frame for a trivial adapter routine such as:

[5]SIMD vector instructions, such as `MOVABS`, usually have strict alignment requirements for the memory operand. In order for arrays allocated on the stack to be eligible for such instructions, the array must be allocated at properly aligned locations. When all stack frames are required to be properly aligned, stack allocation can be greatly simplified for the compiler.

```
ret_to_entry_double:
    RET
```

can be used to 'pad' the size of the one-word (8 bytes on x64) ROP frame to a multiple of 16 bytes in order to meet the alignment requirement.

One adapter routine needs to be written for each pair of resumption protocols for each type. JIT compilers can generate adapters on demand.

Adapter frames help us abstract out the differences between resumption protocols, which are essentially architectural details. Clients only need to reason about return-type consistency instead of return-protocol consistency, as we described in Section 7.3.2.

A Mu implementation can designate a particular resumption protocol as the 'default' protocol between high-level language frames, and insert adapter frames when the resumption protocols do not match. The default protocol should usually be the protocol for call sites, because the vast majority of stack frames are created by function calls, not OSR. The top stack frame, if stopped, usually stops on a Swapstack site, although OSR can push ROP frames at the top of a stack in rare cases. Therefore, it is advisable to let the top stack frame have the expected resumption protocol of Swapstack, while all other frames have the expected resumption protocol for normal return.

When a frame cursor iterates through a stack, it can identify the presence of adapter frames by their code addresses, and skip those frames so that they remain invisible to the client.

8.1.5 Conclusion

Summarizing, the correct implementation of the OSR API relies on maintaining consistent stack frame states. At the machine level, the consistency manifests as the matching of resumption protocols between adjacent frames. Different resumption points give frames different resumption protocols, but the difference can be hidden from the API user by using adapter frames which convert between protocols while preserving the value.

This means that the OSR API of Mu can be simple for the client to use while still being realistic to implement on concrete machines. In the next section, we will demonstrate the concept by implementing the OSR API for the C programming language.

8.2 Stack Operations in Native Programs

Because Holstein, the reference implementation, is implemented as an interpreter, complex stack manipulation is trivially implementable. However, the real-world feasibility of the Mu stack API should also be evaluated in a natively compiled environment. However, we cannot use Zebu — our high-performance Mu implementation — as the experiment platform since it still does not support the stack API at the time of writing.

Instead, we used native programs (in C, C++, LLVM, etc.) as the test bed of the stack operation implementation. Existing compilers for native languages, such as GCC, already generate metadata for stack unwinding to support C++ exception handling. Such information can be reused to to implement a Mu-like stack API.

We developed `libyugong`[6], a library that implements the Mu-like OSR API for native programs, and the Mu-like stack introspection API for LLVM-based languages. To support OSR, we reused the DWARF stack unwinding information generated by LLVM as well as commodity C/C++ compilers such as GCC and Clang, and used `libunwind` [Mosberger] to assist the stack unwinding process. To experiment with machine-independent stack introspection, we used LLVM IR [Lattner and Adve, 2004] as the machine-independent intermediate language, and used the experimental 'stack map' feature of LLVM to locate the value of LLVM-level SSA variables from the stack or registers, which serves as the counterpart of the keep-alive variables in Mu IR.

As the result, `libyugong` successfully supported the Swapstack operation, the removal of stack frames from native stacks, and the creation of ROP frames (explained in Section 7.3.7). However, due to many obstacles in LLVM and native system libraries, stack introspection for LLVM IR does not work properly. We eventually gave up the attempt.

The source code of `libyugong` is available online at: https://gitlab.anu.edu.au/kunshanwang/libyugong

We first look at the stack unwinding information available in native programs and how it can help us implement the Mu stack API.

8.2.1 Stack Unwinding Information

The Mu API for stack operations includes stack introspection and on-stack replacement, but the foundation of both technologies is the ability to unwind the stack. *Stack unwinding* is the process of iterating through stack frames and restoring the states of registers and the stack layout at the resumption point (usually call site) of each frame.

If a platform does not use callee-saved registers and always uses frame pointers (such as `EBP` on x86), stack frames can be iterated by simply following the frame pointers. However, the calling conventions of modern platforms (such as ARM as well as x64 on GNU/Linux, Mac OS X and Windows) define callee-saved registers and do not demand the use of frame pointers. Every function may structure its frame arbitrarily, and may save an arbitrary subset of callee-saved registers anywhere in its frame, provided that it cleans up its own frame and restores the callee-saved registers when returning. Therefore extra metadata is necessary for unwinding the stack outside the usual control flow.

Commodity C/C++ compilers, such as GCC and Clang, generate stack unwinding information when compiling C/C++ functions (unless explicitly disabled by compiler flags, such as '`-fno-exceptions`'). This mechanism exists to support exception

[6]Yu Gong, literally 'foolish old man', is a legendary character in the Chinese fable *The Foolish Old Man Removes the Mountains*. We use 'mountains' as a metaphor for stacks, and the library is going to achieve the seemingly impossible task of removing (part of) it.

handling for the C++ programming language. When an exception is thrown in C++, the runtime will search through the stack frames for an appropriate exception handler. These compilers generate the stack unwinding information even for C functions, so C++ exceptions can propagate through C frames even though the C language has no knowledge about C++ exceptions.

On GNU/Linux, the stack unwinding information is encoded in a DWARF4-like format and is stored in the `.eh_frame` section of ELF images. It uses an incremental format to efficiently encode 'how to unwind the current function frame' at every program counter position in a function. Mac OS X can use both DWARF and a more concise 'compact unwind information' encoding in the `__compact_unwind` section of its Mach-O images.

Since the stack unwinding information already exists in the executable images of C/C++ programs, we can make use of such information to implement the Mu-like OSR API.

8.2.2 Implementing OSR for Native Functions

The proof-of-concept project `libyugong` is implemented on x64 for GNU/Linux and Mac OS X. It includes the Swapstack operation for C/C++ and a Mu-like frame cursor which iterates through the stack frames, pops frames and creates ROP frames.

For ease of implementation, the two platforms are chosen because both use the System V ABI for AMD64. We limited the type of the value passed between stacks by the Swapstack operation, and the return types of functions, to `uintptr_t`. Other types can also be passed according to the calling convention.

Resumption Protocols

As discussed in Section 8.1.4, resumption protocols are the key to implementing a unified view of stack frames, where the expected type of one frame only needs to match the returned type of the frame above.

Consider that a commodity C compiler will compile most functions to return using the `CCC_Ret` protocol. We add adapter frames such that all frames expect the `CCC_Ret` protocol from the frame above. Thus, adapter frames only need to be inserted above ROP frames.

The top frame of a stack is special: when a stack is unbound, it is always expecting to be re-bound by a Swapstack operation, thus may be different from other frames which are expecting normal returns. We design the structure of the top of an unbound stack as in Figure 8.4(b). The top word, as pointed to by the saved stack pointer, is always the address of the `_yg_ss_cont` routine (in Figure 8.4(a)). When the Swapstack operation rebinds to a new stack, it will always pop the top word and jump to it. The value of all callee-saved registers are immediately below the stack-top word, which the `_yg_ss_cont` routine pops to actual registers. The return address returns to the resumption point of the first user frame of the swappee stack. Using this design, all unbound stack can be resumed the same way.

The SWAPSTACK Operation

The Mu API requires a Swapstack operation to be performed to leave the current stack in an unbound state before the stack can be unwound. Dolan et al. [2013] suggested that the most efficient implementation of Swapstack needs the assistance of the compiler which can optimise for register usage. Since we are building a proof-of-concept project, we take a less efficient but much simpler approach.

The `yg_stack_swap` routine in Figure 8.4(a) performs a Swapstack operation and passes a value to the swappee stack. We force this routine to be called as if it is a C function. This effectively forces the C compiler to save all caller-saved registers of the C calling convention at the caller's side. Thus `yg_stack_swap` only needs to save all callee-saved register so that all registers are saved on the swapper stack.

The `yg_stack_swap` routine lays out the current stack in the format of Figure 8.4(b). When the `RET` instruction in `yg_stack_swap` is executed, the current thread always jumps to the `_yg_ss_cont` routine which restores the callee-saved registers and resumes the top frame of the swappee. When another Swapstack operation swaps back to the swapper, the `_yg_ss_cont` routine resumes the swapper's stack in the same way.

The Initial Stack Layout

When creating a new stack, we allocate a memory buffer, and lay out the top of the stack as in Figure 8.5. This layout is compatible with Figure 8.4(b), and can be resumed by the Swapstack operation in the same way.

There is a ROP frame for the user-specified stack-bottom function below the saved registers. The implementation of the ROP frame and its adapter frame is exactly as described in Section 8.1.4. When swapping to such a stack, execution will start from the entry point of the stack-bottom function. With the adapter frame, the value passed to the swappee stack by `yg_stack_swap` (in the return value register `RAX`) will be received by its sole parameter.

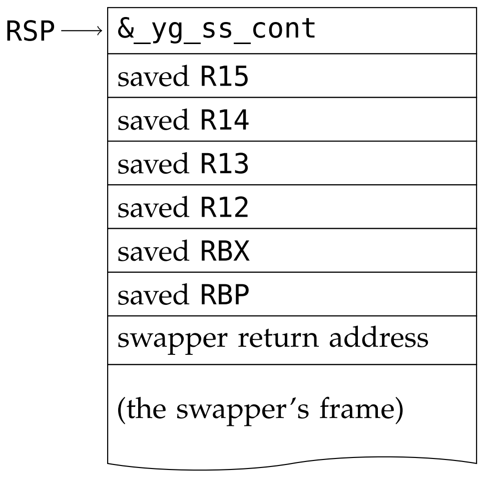
Stack Unwinding

With the stack-unwinding metadata available in the `.eh_frame` section, we could have implemented our own stack unwinder. However, we used the `libunwind` library to do it for us. `libyugong` has to initialise the register states for `libunwind` by decoding the stack-top layout we defined; after this, `libunwind` can unwind other stack frames using the information from the `.eh_frame` section, and help us restore the values of the program counter, the stack pointer, callee-saved registers in each stack frame.

`libunwind` can usually identify the function by the return address and look up the metadata accordingly. However, `libunwind` does not have enough information to unwind adapter frames, which must be unwound manually by `libyugong`. `libyugong` must also behave like the Mu API and skip adapter frames so that the user does not see them as part of the stack. As described in Section 8.1.4, `libyugong` knows

```
1  yg_stack_swap:
2    push rbp
3    push rbx
4    push r12
5    push r13
6    push r14
7    push r15
8
9    mov  rax, rdx
10
11   lea  rcx, [_yg_ss_cont+rip]
12   push rcx
13
14   mov  [rdi], rsp
15   mov  rsp, [rsi]
16   ret
17
18 _yg_ss_cont:
19   pop  r15
20   pop  r14
21   pop  r13
22   pop  r12
23   pop  rbx
24   pop  rbp
25   ret
```

(a) Implementation



(b) Stack-top Structure of Unbound Stacks

Figure 8.4: Swapstack implementation in `libyugong`. Figure (a) shows the assembly code for x64 on GNU/Linux. The `yg_stack_swap` function is called by the swapper, and must be called using the C calling convention with the signature '`uintptr_t yg_stack_swap(void** swapper, void** swappee, uintptr_t value)`'. The first two parameters (`RDI` and `RSI`) are the locations where the stack pointer of the current stack is saved and where the stack pointer of the swappee is loaded from. The third parameter is the value to be passed to the other. When called using the C calling convention, the caller-saved registers must have already been saved by the caller compiled by a compliant C compiler. Lines 2–7 then save all callee-saved registers; line 9 moves the third argument (`RDX`) to the return value register (`RAX`); lines 11–12 push the address of `_yg_ss_cont`; lines 14–15 save the current stack pointer and load the stack pointer of the swappee stack; and line 16 finally returns to the swappee. In `libyugong`, the top of all suspended stacks is laid out as in Figure (b), and the stack pointer always points to the address of the assembly snippet `_yg_ss_cont`. When the swapper executes the `RET` instruction in line 16, the thread continues at line 19, restoring the callee-saved registers saved by the swappee stack and returns to the resumption point of the top frame of the swappee. The resumption point is usually the next instruction after the call to `yg_stack_swap`. But when OSR happens and stack frames are popped and pushed, `libyugong` will fix the stack top layout to match Figure (b) so that any thread can swap to any unbound stack the same way.

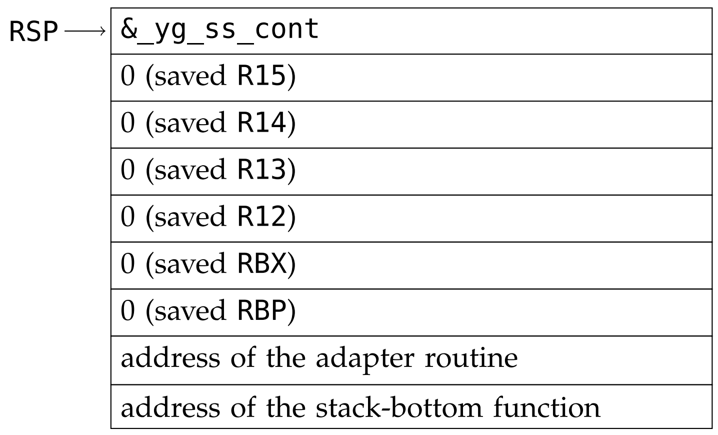


Figure 8.5: The structure of newly-created stacks. The stack is laid out in a way compatible with Figure 8.4(b), except the callee-saved registers all have zero values. Below the saved registers, there is the ROP frame for the stack-bottom function.

the addresses of the adapter routines, and reports to the client as a ROP frame for a function.

Popping and Pushing Frames

The unwinding information can restore the callee-saved registers at the resumption point of any stack frame. If the client wants to pop all frames above a give frame, `libyugong` only needs to re-construct the stack-top structure above the stack location where the return address of the frame is held. The callee-saved registers of the resumption point will be saved in the stack top structure as usual.

To push ROP frames on the stack, `libyugong` can simply remove the stack-top structure, place the ROP frame right above the stack location where the return address of the original stack-top frame is held, and push a new stack-top structure above it. ROP frames do not modify callee-saved registers, because when resuming from the entry point, the function will always save the callee-saved registers for its caller. Therefore pushing ROP frames preserves all saved callee-saved registers in the stack-top structure.

A Limitation of `libyugong`

The limitation of `libyugong` is its sensitivity to *inlining*. When the C/C++ compiler inlines one function into a call site, the frame for the callee is no longer created, and cannot be identified by `libyugong`. To overcome this limitation, the compiler must assist the stack unwinder to identify logical frames within physical frames, as done by JikesRVM [Alpern et al., 2009] and the Self VM [Hölzle et al., 1992].

8.2.3 LLVM and Stack Maps

In order to experiment with the Mu-style machine-independent stack-introspection API, we need a machine-independent intermediate language. LLVM IR is an obvious choice since Mu IR itself is designed with LLVM IR as its baseline.

The LLVM intrinsic function '`@stackmap`'[7] is the counterpart of the `KEEPALIVE` variables of Mu IR. The LLVM stack map was originally designed to help garbage collectors locate the values of object references in local variables. The LLVM backend generates a stack map at every '`@stackmap`' intrinsic function. Each stack map records the way to retrieve the values of user-selected variables at its program counter. A variable may be held in a register, may be computed by adding a register and a value, may be held in a certain offset of the stack frame, and may also be a constant because LLVM may perform constant folding before stack map generation. The stack map is stored in the '`__llvm_stackmaps`' section of the generated ELF or Mach-O image, and has an open format which can be decoded using a library provided by LLVM.

To use the LLVM stack map, the application must be written in LLVM IR, and keep-alive variables must be annotated with the '`@stackmap`' intrinsic function. The intrinsic function is usually placed after call sites so that its program counter is the next instruction after the machine-level `CALL` instruction. The LLVM client must maintain a map between stack map program counters and the high-level call sites. At each call site, the client also needs to enumerate all keep-alive variables in order to interpret the stack map correctly.

We developed the `libyugong-llvm` module to assist this process. At compile time, the module helps the LLVM client emit `call` instructions annotated with keep-alive variables. At run time, the module helps the client decode the stack map and restore the LLVM-level variable values. The run-time goal is achieved by collaboration between the `.eh_frame` unwinding information and the LLVM stack maps, with the former restoring the values of callee-saved registers and stack pointers, and the latter mapping the machine-level stack frame states to LLVM-level values.

However, the `libyugong-llvm` module did not work as reliably as expected. Unfortunately, the LLVM backend compiler may insert instructions between the machine-level `CALL` instruction and the stack map. For example, after a function returns, there will be `MOV` instructions that copy the return value into the desired register or memory location. This will cause the stack maps to be generated after those `MOV` instructions instead of immediately following the `CALL`. As a result, the keep-alive values can only be successfully retrieved when LLVM eliminates those `MOV` instructions when certain optimisations are turned on, which does not always happen.

Then we realised we had been using the `@stackmap` intrinsic the wrong way, because the stack map must be generated together with the function call. Given enough time, we could eventually work around the problem by either modifying LLVM or

[7]We abbreviated the names for simplicity. The full names of the intrinsic functions discussed in this section are `@llvm.experimental.stackmap` and `@llvm.experimental.gc.statepoint`.

trying the newer '`@statepoint`' intrinsic. However, our attempt to implement the Mu-style stack API for native programs had been hindered by many other difficulties, so we abandoned the attempt.

8.2.4 Difficulties in Implementing Stack Operations in Uncooperative Native Programs

The attempt to support the Mu-style stack operation API proved to be very difficult. There are at least two major sources of difficulty.

There is no publicly available platform-independent library for unwinding a native stack. The `libunwind` implementation from http://www.nongnu.org/libunwind/ is developed for GNU/Linux. Mac OS X has its own `libunwind` implementation, and LLVM also provides an `libunwind` implementation donated by Apple. These implementations are API-compatible, but different in implementation details. In fact, `libunwind` is designed to unwind the *current* stack instead of an unbound stack, and is not designed to modify the stack in the Mu style. Using `libunwind` this way inevitably requires deep knowledge of its implementation detail.

The low-level system runtime libraries are poorly documented. For example, `libgcc` is responsible for the handling of C++ exceptions on GNU/Linux at run time. When code is loaded at run time (usually due to the dynamic loading of shared libraries, or JIT compiling), the `__register_frame` function needs to be called to register the `.eh_frame` section to the runtime system. However, this function is, at the time of writing, undocumented [LibGCCEH]. On Mac OS X, the stack unwinding support for C++ exception handling is provided by `libunwind` itself, and the `__register_frame` function is provided by `libunwind` as an internal function. The undocumented API makes it impossible for language implementers to properly support stack unwinding for JIT-compiled or dynamically loaded native code without digging deep into the implementation details of system libraries. Moreover, the `__register_frame` function behaves differently on the two platforms we explored. GNU/Linux requires one call per `.eh_frame` section, while Mac OS X requires one call per each frame description entry (FDE) in that section. This further complicated the situation.

In essence, the native world consists of non-cooperative components that are not designed to work together to support a Mu-style runtime API. `libunwind` was not designed for Mu-style OSR. The exception handling mechanisms of system libraries were designed to support C++ exceptions which can be turned off by a compiler flag. LLVM was not designed with introspection and OSR in mind, either.

What language developers need is a single platform that is designed carefully as an abstraction over execution, providing a clearly defined API for JIT compiling, execution, stack introspection and OSR.

Mu provides such a platform. The efficient implementation of high-level languages demands much more than the mere generation of machine code. It is the collaboration between the high-level client and the low-level substrate that makes an efficient JIT-compiling language runtime.

8.3 Related Work

This section discusses other existing implementations of OSR in addition to SpiderMonkey and V8, and compares their approaches with Mu.

Self VM The Self VM [Hölzle et al., 1992; Hölzle and Ungar, 1994] implemented its own SWAPSTACK mechanism which predates the work of Dolan et al. [2013]. A thread switches to a dedicated VM stack for handling de-optimisation, and could therefore manipulate the victim stack using high-level C++ code. The Self VM uses return address patching[8]—replacing the return address of a frame so that frame replacement can happen when the patched frame returns. Unlike Mu, the Self VM is not a multi-language VM, thus it only uses SWAPSTACK and OSR as an internal mechanism.

JikesRVM JikesRVM [Alpern et al., 2009] is a JIT-compiling JVM with OSR support. JikesRVM recompiles methods and generates the contents of new frames in separate OSR threads. However, JikesRVM uses return address patching to let the victim thread execute a piece of code to replace its own frames upon returning from yieldpoints. The code is generated at run time by stitching together platform-specific assembly snippets,[9] because the victim thread is replacing stack frames on the current stack, and thus must carefully handle the stack layout.

JVM The JVM does not have a public API for its internal OSR facilities. The JVM Tool Interface (JVM TI) provides some stack-related API functions for debugging, including introspecting local variables and popping frames. However, unlike the Mu API, JVM TI does not support constructing new frames on existing stacks, therefore does not have all tools needed by OSR.

Truffle and Graal [Würthinger et al., 2013] provide a high-level partial evaluation-based language implementation framework and a compiler framework on the JVM. Truffle also has an abstraction of stack frames to support interpreting and de-optimisation. This is a higher-level abstraction than Mu. Mu only provides a *minimal* API for the Mu client to build higher-level abstraction. Truffle can be viewed as a potential client that could be implemented *upon* Mu. If properly designed, Mu should allow the Truffle API to be implemented in terms of Mu's API.

LLVM The LLVM [Lattner and Adve, 2004] compiler framework provides stack maps [LLVMStackMap] for stack introspection, but no high-level OSR API.

[8]'Return address patching' is unrelated to 'return-oriented programming' despite the similar names. ROP is a paradigm where the return values of a function are passed as parameters to the next frame; while return address patching is a technique that overwrites the return address so that some code can be executed after the victim function returns, but before the next frame is activated.

[9]See the `org.jikesrvm.osr.ia32.CodeInstaller.install` method: https://github.com/JikesRVM/JikesRVM/blob/master/rvm/src/org/jikesrvm/osr/ia32/CodeInstaller.java#L50

D'Elia and Demetrescu [2016] developed OSRKit, a library built on LLVM for 'dynamically transferring execution between different versions of a function at run time', which they define as 'OSR'. It is an improvement over its predecessor in McJIT by Lameed and Hendren [2013]. In OSRKit, the old version of a function tail-calls a stub which recompiles the function and then tail-calls the new version. Both OSRKit and Mu achieved the goal of transferring execution under the constraint that the high-level optimiser and de-optimiser must not depend on machine-level details. Built on LLVM which does not provide any abstraction over stack manipulation, OSRKit chose to depend only on function calls, and not to 'adjust the stack so that execution can continue in f' (the new version) with the current frame' [D'Elia and Demetrescu, 2016], because that 'requires manipulating the program state at machine-code level and is highly ABI- and compiler-dependent.' [D'Elia and Demetrescu, 2016]. Unlike LLVM frontends, Mu clients can rely on the platform-independent OSR API of Mu to replace stack frames while still retaining platform independence. As a more powerful substrate, the Mu API is more flexible than pure call-based approach. For example, the API can easily replace multiple frames at a time, which is useful for handling inlining [Hölzle et al., 1992; Hölzle and Ungar, 1994].

8.4 Summary

This chapter discussed the implementation of the Mu stack API. Using resumption points and resumption protocols, we showed how the API itself can be correctly implemented by Mu at the machine level. Our partially failed proof-of-concept OSR implementation for native programs indirectly reflected the importance of a well-designed platform that abstracts over execution, a principal goal of Mu.

In the next chapter, we will shift our focus to supporting real-world language implementations as Mu clients.

Part III

Supporting Real-world Language Runtimes Using Mu

Chapter 9

RPython and GHC as Mu Clients

In Part I and II, we presented the design of Mu, and had a closer look at the support of stack operations. In this chapter, we will describe our experience in porting RPython as a client of Mu, and our preliminary research on GHC. The RPython toolchain from the PyPy project has supported many language other than Python, including SOM, Racket, Erlang, etc. Therefore, supporting RPython on Mu has the strategic significance of simultaneously enabling many languages built on RPython to work on Mu. Investigating a real-world language framework also forced us to address issues we did not take into consideration before, such as ahead-of-time compilation, boot-image building, and linking with native programs. GHC is the de-facto standard Haskell implementation, also known for its performance. We choose Haskell because we want to address a *wide* range of different languages, and Haskell is an unorthodox functional language with lazy evaluation.

This chapter is structured around the two clients. Section 9.1 discusses the RPython client; Section 9.2 discusses our preliminary research on GHC as a client; Section 9.3 concludes this section.

The RPython client is collaborative work with my colleague John Jiabin Zhang [Zhang, 2015], and the work on GHC is collaborative work with Nathan Young [Young, 2016] and Andrew Hall. External collaborators in the University of Massachusetts Amherst are working with us on meta-tracing JIT compiler of the RPython framework.

9.1 Supporting RPython and PyPy on Mu

In this section, we will introduce our experience in supporting RPython as a client for Mu, and the impact on the design of Mu.

9.1.1 The RPython Framework

PyPy [Rigo and Pedroni, 2006] is a high-performance virtual machine for the Python language. Its underlying RPython framework underpins PyPy's performance.

RPython is a restricted, statically typed subset of Python. There are restrictions on the integer range, variable usage, the use of container types, etc., and it also forbids the run-time declaration of new types or methods. Nonetheless, RPython language is garbage-collected, has full support for exception handling, and has a comprehensive (but certainly smaller than CPython) low-level library that implements string operations and IO. In general, RPython plays the role of a 'high-level low-level language' for VM development as described by Frampton et al. [2009].

As a platform, RPython is relatively high-level compared to C or C++, but is still a suitable platform for implementing other high-level languages. The high-level language developer implements the language as an interpreter (such as the PyPy interpreter) written in RPython with some annotations. At run time, a meta-tracing JIT compiler records the RPython-level operations the language interpreter has taken in order to interpret a user-level loop, and JIT-compiles the trace into machine code [Bolz et al., 2009]. In this way, the meta-tracer works at the level of RPython operations, and provides JIT compiling for any language that has an interpreter in RPython, such as PyPy, RPySOM [Marr et al., 2014], Pycket [Baumann et al., 2015], etc.

RPython also serves as an ahead-of-time compiler that compiles RPython programs.

The frontend 'annotator' and 'RTyper' perform type inference on the input RPython program. Starting at the main function, they visit all functions transitively called from the main function, and infer the types of all parameters and variables. By the end of this step, the input RPython functions will have been turned into control flow graphs (CFGs) with low-level operations (the 'LL operations') on the low-level type system (the 'LL type system'). The LL type system is at a similar level to the Mu type system, with both C-like primitive types and garbage-collected object references; and the LL operations are at a similar level to the Mu instruction set.

The backend of the original RPython implementation injects the implementation of exception handling and garbage collection into the CFGs, and then translates the CFGs into C source code which is then compiled using commodity C compilers such as GCC. The injection of exception handling and GC is necessary for C — the default target — which does not support these mechanisms natively. Similar to CPython [CPython], exceptions are implemented as a C-level thread-local variable that points to the current exception object. Each RPython-level function call is followed by a check on this variable, which either handles the exception, or returns prematurely to propagate the exception to its caller, where the same check is performed. Since the C compiler has no knowledge about GC and object references, the backend also injects GC barriers around write operations of object references to support generational collectors, and also add information to identify object references held on the stack to support exact GC.

In order to support the meta-tracing JIT compiler, the RPython backend, when compiling, not only generates C code, but also serialises the LL-typed CFGs (of the high-level language interpreter) into 'JIT code' which is a compact bytecode format, but semantically equivalent to the LL-typed CFGs. At run time, a meta-interpreter (also implemented in RPython) executes the high-level language interpreter by interpreting its 'JIT code', and records traces which consist of the LL operations performed by the *language interpreter*. The 'meta-trace' is then optimised and JIT-compiled directly into machine code. Note that the run-time meta-tracing JIT-compiler toolchain is separate from the ahead-of-time RPython-to-C toolchain.

9.1.2 Adding Mu as a Backend of the RPython Framework

Given the structure of the RPython framework, we find two places where Mu can fit into the framework, and replace the existing backends.

Mu as An Ahead-of-time Compilation Backend We can cut into the ahead-of-time compilation pipeline of the RPython language, replace its backend, and target Mu IR instead of C source code. A language developer still implements a high-level language by writing an interpreter in RPython, but the new AoT compiler toolchain compiles the interpreter into Mu IR, instead. The interpreter will then be executed on the Mu micro virtual machine, making use of its exception handling mechanism and garbage collector.

Mu as A Just-in-time Compilation Backend We can replace the code generator of the JIT compiler of the meta-tracing framework, and target Mu IR instead of native machine code. After recording traces and optimising, the language implementation will call into the Mu micro virtual machine, and construct Mu IR for the optimised traces at run time using the Mu API. In this case, Mu will serve as a platform-independent execution engine, and will free the developer from having to emit code for each platform.

The resulting language implementation will be a *meta-circular client* of Mu — the language implementation uses Mu as a JIT backend through the Mu API, while the language implementation itself is also a Mu IR program that executes on Mu.

The implementation of both the AoT and the JIT backends is in progress. This thesis will focus on the details of the AoT backend, and will briefly introduce the JIT backend.

We now proceed to the key part of AoT compilation, that is, how to translate RPython programs into Mu IR.

9.1.3 Translating RPython Programs into Mu IR

The natural place to inject Mu into the RPython compilation pipeline is the point where the CFGs have been lowered into the LL type system, but before the exception handling and garbage collection implementation is injected. Exception handling is directly supported in Mu IR, as described in Section 5.3.3. Since the LL type system still contains object reference types, Mu can handle the implementation of GC properly, including the insertion of GC barriers and the generation of stack maps, as long as the client correctly maps its object references to the Mu-level counterparts.

From the high-level, the translation consists of the mapping of types and the mapping of operations.

LL Types	Mu Types
`Signed`	`int<32>`
`Unsigned`	`int<32>`
`SignedLongLong`	`int<64>`
`UnsignedLongLong`	`int<64>`
`Char`	`int<8>`
`Bool`	`int<1>`
`Float`	`double`
`SingleFloat`	`float`
`Void`	`void`
`Struct`	`struct<...>`
`GcStruct`	`struct<@GcHeader ...>`
`FixedSizeArray`	`array`
`Array`	`hybrid<int<64> T>`
`GcArray`	`hybrid<@GcHeader int<64> T>`
`Ptr<Struct>`	`uptr<struct<...>>`
`Ptr<GcStruct>`	`ref<struct<@GcHeader ...>>`
`Ptr<FuncType>`	`funcref`
`Address`	`uptr`

Table 9.1: Mapping Between the LL Type System and the Mu Type System

Mapping of LL Types The LL type system is at a similar level to the Mu type system. Table 9.1 lists the corresponding Mu types for selected LL types. Primitive types map directly to primitive Mu types, except that Mu only distinguishes signed values from unsigned values in instructions instead of types. Composite types usually have Mu counterparts, too, except that RPython-generated 'GC headers'[1] need to be explicitly added to Mu `struct`s, and the arrays whose length is determined at allocation time need to map to the `hybrid` type in Mu. A major difference between RPython and Mu is the handling of references. In the LL type system, the `Ptr` type can be used for both traced references and untraced pointers. Whether it is traced is determined by the type it points to, whereas in Mu the traced-ness is determined by the `uptr` or the `ref` types themselves. Therefore, the LL `Ptr` type maps to the Mu `uptr` type when pointing to raw types, but maps to `ref` when pointing to `GcStruct` and its derived types.

Mapping of LL Operations LL operations are also at a similar level to the Mu instruction set. The CFG structure in RPython uses the SSI form [Ananian, 1999], which is equivalent to the form of Mu IR,[2] therefore the translation is mostly straightforward. Table 9.2 lists the corresponding Mu instruction sequences for selected LL operations. Primitive operations, such as arithmetic, comparison and conversion, have direct counterparts in Mu. For memory allocation, the `malloc` LL operation works for both GC types and non-GC types, in a similar way how the LL type `Ptr` can express both traced reference and untraced pointers. We translate the allocation of GC types into the `NEW` and the `NEWHYBRID` heap allocation instructions, but the allocation of non-GC types has to be off-loaded to the native `malloc` function. The Mu instruction set is also more RISC-like, therefore some LL operations, such as `getfield`, correspond to more than one Mu instructions.

Since we intercepted the RPython work flow before exception handling injection, the CFG still represents exception handling as a multi-way branch at the end of a basic block, mapping each exception type to a different destination. We need to implement the actual exception handling by manually checking the exception type and branching to the correct destination, since Mu has no knowledge about the client-level exception type hierarchies. This translation involves long sequences of Mu instructions, but the process is straightforward. Similarly, since we intercepted before the injection of GC implementations, GC-related LL operations, such as barriers, do not appear at this stage, and we can safely omit them in our RPython-to-Mu translator.

[1]Despite the name, the so-called 'GC header' generated by RPython may contain more than GC-related fields, such as the hash code of an object. The exact fields of the 'GC header' depend on the concrete GC algorithm. For example, when using reference counting, the 'GC header' contains a reference count and a hash code; when using the 'minimark' algorithm, it contains a type ID and several flags, but no hash code. When translating for Mu, we customised the 'GC header' to only contain a hash code. Mu can generate and manage its own metadata for garbage collection, which is totally based on the definition of Mu-level types, such as the presence of `ref` fields in Mu `struct`s, and does not depend on any GC headers added by RPython.

[2]Mu was using the LLVM-style SSA form before we moved to the current SSI-equivalent form. The experience with RPython also spurred the transition.

LL Operations	Mu Instructions
`int_add`	`ADD <@i32>`
`int_sub`	`SUB <@i32>`
`int_neg`	subtract from zero using `SUB`
`int_floordiv_zer`	`SDIV <@i32> ...  EXC(...)`
`int_add_ovf`	`ADD [#V] <@i32>`
`llong_add`	`ADD <@i64>`
`int_lt`	`SLT <@i32>`
`uint_lt`	`ULT <@i32>`
`llong_lt`	`SLT <@i64>`
`ullong_lt`	`ULT <@i64>`
`cast_int_to_longlong`	`SEXT <@i32 @i64>`
`truncate_longlong_to_int`	`TRUNC <@i64 @i32>`
`cast_longlong_to_float`	`SITOFP <@i64 @double>`
`convert_longlong_bytes_to_float`	`BITCAST <@i64 @double>`
`malloc` (of GC types)	`NEW`
`malloc` (of non-GC types)	Call the C function `malloc` with `CCALL`
`malloc_varsize` (of GC types)	`NEWHYBRID`
`malloc_varsize` (of non-GC types)	Call the C function `malloc` with `CCALL`
`getfield`	`GETFIELDIREF` and then `LOAD`
`setfield`	`GETFIELDIREF` and then `STORE`
`getarrayelement`	`GETELEMIREF` and then `LOAD`
`setarrayelement`	`GETELEMIREF` and then `STORE`
`direct_call` (to RPython functions)	`CALL` (to Mu functions)
`direct_call` (to C functions)	`CCALL` (with C calling convention)
`cast_ptr_to_adr`	`INTRINSIC @uvm.native.pin`
`gc_identity_hash`	implement as Mu IR function
`raw_memcopy`	not directly supported
`gc_write_barrier`	will not appear in the Mu backend

Table 9.2: Mapping Between LL Operations and the Mu Instruction Set

RPython apparently assumed C source code as its eventual target, which introduced some difficulties in the translation.

Under this assumption, the LL operation `direct_call` can be used to call both RPython functions and C functions. This works with the C backend, because RPython functions are translated to C functions too. Therefore both RPython and C function can be called using the C function call expression in the resulting C source code. This does not work with the Mu backend, because the Mu backend translates RPython functions into Mu functions. Mu calls Mu functions using the `CALL` instruction, and calls native C functions using the `CCALL` instruction.

Moreover, RPython can use C macros as if they were variables or functions. For example, on GNU/Linux, '`stat64`' is a macro that expands to the actual function '`__xstat64`':

```
#define stat64(fname, buf) __xstat64 (_STAT_VER, fname, buf)
```

RPython declares '`stat64`' as an external function, therefore the LL-typed CFGs contain function call instructions to '`stat64`', and the C code generator of the RPython C backend simply emits the identifier '`stat64`' whenever this 'function' is used. This may generate something like '`int v3 = stat64(v1, v2);`' which is valid C code because the code is subsequently compiled by the C compiler which expands this macro. This does not work with Mu, because Mu interacts with native programs at the ABI level. If we define such macros as external C variables or functions, the micro virtual machine will try to resolve symbols, such as '`stat64`', in the `.so` or `.dylib` files of LibC, which do not always exist.

The RPython-to-Mu translator has to handle calls to RPython and C functions differently and, if necessary, wrap C macros into our own wrapper functions written in C in order to be called via the Mu native interface.

Another difficulty is with handling features that are closely related to garbage collection. In RPython, every object also contains a hash code like a Java hash code in its header. The '`gc_identity_hash`' LL operation returns the hash code stored in the header, or initialises this field with the current address of the RPython object if it is not yet initialised. We implement this operation as a Mu IR function, which initialises the hash code with the current address of the corresponding Mu object[3].

Some existing low-level RPython code, such as its string library, makes too many assumptions about how its garbage collector works. The code casts references to pointers using the `cast_ptr_to_adr` LL operation, and makes sure GC does not happen while the raw address is in use. This trick is used to copy large buffers using the native `memcpy` function, such as during string concatenation. In Mu, the address of heap objects can only be obtained by 'object pinning' (see Section 5.3.7). We had to carefully work around by inserting pinning operations. However, copying

[3]Getting the address of a Mu object requires pinning, which is overkill for hashing. An alternative approach is to initialise the hash code field with a random number. The number can be the result of any CPU counter or any random number generation instruction supported by the processor, such as the `RDRAND` instruction on x86, as long as the result is cheap to obtain and *relatively* random. We can also use traditional address-based hashing like JikesRVM. In that case, the Mu garbage collector should provide an intrinsic for getting the hash code of a heap object.

by pinning and memcpy only works for objects that do not contain references. Object references cannot be copied with memcpy without completely disabling GC, since GC may still concurrently update the references while memcpy is running. It is arguable that Mu should provide an instruction that can copy arrays of GC-traced types, and implement it in the fastest way with respect to the GC it used.

Influence on the Mu design Working on the RPython-to-Mu translator helped identifying missing features that need to be provided by Mu. Actually, the ADD instruction with the overflow flag in Mu is inspired by the need to translate the corresponding LL operation 'int_add_ovf'. This is also needed by other languages that need de-optimisation upon overflow. The unsafe native interface did not exist when this RPython backend project started, and it pushed the Mu design to address object pinning and native function calls in detail.

9.1.4 Building Boot Images

Working with RPython forced us to make a concrete design of the boot image mechanism in Mu.

A *boot image* is a image that contains both executable code and preserved data, including heap objects and the static memory space. In the case of Mu, a boot image also includes the contents of loaded Mu IR bundle, such as types and functions.

Boot image has been used by many existing language implementations, including SMALLTALK-80 [Goldberg and Robson, 1983] and JikesRVM [Alpern et al., 2009]. In the original RPython C backend, the generated executable file is also a boot image.

Boot image is necessary for supporting RPython. RPython programs contain heap objects that are pre-allocated before the program starts, such as the string literal "Hello world" in the following example:

```
def foo():
    print "Hello␣world"
```

This demonstrated that Mu needs a way to persistently save heap objects, too, and load them quickly at start-up time.

We intentionally do not define the exact format of boot images. Concrete Mu implementations can choose the most efficient format for the particular implementation of the GC and the JIT compiler for the desired platform. An ELF image, as used by Moxie [Blackburn et al., 2008], is a good candidate, because it is widely supported by native toolchains. The ideal format may be an object file (.o). The pre-compiled Mu functions and the saved Mu heap can be simply memory-mapped from the ELF image, and the dynamic linker can be used to resolve dependencies between functions and objects. Emitting a .o file rather than shared objects or executable files gives the client more freedom to link the image with the client and external libraries. Metacircular clients can be built directly into the boot image, as depicted in Figure 4.2(b). Even non-metacircular clients can benefit from boot images by pre-compiling library objects and functions, such as the implementation of the java.lang.Object in Java.

The Mu API supports building boot images. There is only one extra function: '`make_boot_image`'. It is essentially a 'one-button' mechanism to preserve the current VM state. The client will supply the following information to this API function as parameters:

- A white list of top-level definitions. This allows the client to selectively store only useful items from the bundle to the boot image.
- The state of the primordial thread. A *primordial thread* is a thread that starts execution after the boot image is loaded. Its state includes the 'main' function which the thread starts with. In the future, this could be extended to support multiple primordial threads and preserve stacks.
- The output file. It will specify where to write the boot image, although the format of the boot image is not specified.

Implementation-wise, the relocation of the heap needs to be handled carefully. The dynamic loader may load the image at any location in the address space, and the references to objects in the primordial heap may have different addresses in each run. Therefore, the boot image builder must assist the dynamic loader with symbols and relocation entries so that it can fix the addresses after loading. The garbage collector can identify all references in the heap, and is able to generate appropriate metadata. Mu may choose not to depend on the system loader, in which case it will have to implement the relocation manually.

Since real-world programs need to call C functions which are usually also loaded dynamically, we need to let the dynamic loader find the addresses of native functions at load time so that Mu functions can directly call native functions without having to look them up manually using system functions such as `dlsym`. We added another kind of constant — 'external constant' — to Mu IR. For example, '`.const @write_addr <@SomeFuncPtrType> = EXTERN "write"`'. Different from other constants whose values are determined at compile time, the values of external constants are determined at load time. This allows the dynamic loader to fix call sites with the appropriate callee address. The exact way the loader looks up the address from the symbol, such as '`write`', has to be left unspecified by the Mu specification, for it depends on so many run-time libraries that are not well-documented and behave subtly differently on different platforms. Fortunately, there is ongoing research carried out by Kell et al. [2016] on the formalisation of linker behaviours.

The experiences with RPython forces us to think more about ahead-of-time compiling. The 'external constant' is not necessary for the JIT-compiling scenario, because at run time, all functions have fixed addresses, and the JIT compiler can simply use constant addresses instead. There are still many issues about ahead-of-time compiling, which Mu has not properly addressed at this time. These issues are mainly related to linking and loading, such as weak symbols, symbol versioning, native library dependencies, and symbol resolution with namespaces. It is a future research topic to address these issues to provide better support for AoT compilation in Mu.

9.1.5 Preliminary Results

Currently, we have worked to the point that the RPython client can support the RPySOM [Marr et al., 2014] interpreter after fixing a few bugs in RPySOM. It runs on Holstein, and passes the test suite provided by the SOM language.

The RPython client can only support the PyPy interpreter with a minimum set of RPython modules. In PyPy, low-level modules, such as socket and regular expression, are implemented in RPython. Some of these modules use low-level RPython primitives which depend on the implementation details of the original RPython backend, and have not been correctly translated to Mu, yet. Nevertheless, we are able to start an interactive Python shell on Holstein, and run simple Python programs.

We are also able to run many other smaller programs in the RPython test suite. Among those tests, we consider the ability to run the SHA-1 message digest algorithm as a major milestone in the development of the Mu backend. Being able to run this non-trivial algorithm demonstrated the correctness of the translation, because any mistake in the computation would result in a completely wrong hash code.

9.1.6 Supporting the Meta-tracing JIT Compiler

We have discussed how to support the *ahead-of-time* compilation toolchain of RPython using Mu as its backend. However, the performance advantage of PyPy mainly comes from its *just-in-time* compiler that dynamically compiles traces into optimised machine code. Research on supporting PyPy's meta-tracing JIT compiler was carried out at the University of Massachusetts in parallel to our work. When targeting Mu, the JIT compiler should emit Mu IR code just like the AoT RPython-to-Mu translator does.

The main challenge for supporting JIT compilation has been to provide enough Mu-level information to the run-time JIT compiler. The official RPython backend loses information when persisting LL-typed CFGs into bytecode (i.e. the 'JIT code', which is the compact run-time representation of the CFGs). For instance, heap-allocation operations no longer know the LL type names. Instead, they are 'exploded' into an allocation of a certain size, followed by operations that fill in the GC headers. This approach specialised the CFGs too much for the concrete PyPy GC. In Mu, the heap allocation instruction '`NEW`' takes the type as its parameter instead of the size, since Mu does not expose the object layout to the client. Therefore, the persisted 'JIT code' has to be augmented with Mu-level type information so that the JIT compiler can correctly generate Mu instructions with Mu-level type information.

9.1.7 Summary and Future Work

In summary, the ahead-of-time translation from LL-typed RPython code to Mu IR was relatively straightforward because its LL type system lies roughly at the same level as Mu. We were able to run the RPySOM interpreter and the PyPy interpreter with a minimal set of modules on the Holstein reference implementation. This experience has influenced the design of Mu to add missing features, and pushed us to think more deeply about metacircularity and ahead-of-time compiling.

9.2 Supporting GHC on Mu

We have introduced our experiences with RPython. In this section, we will move on to our preliminary work on supporting GHC, the de-facto standard implementation of Haskell. This section will briefly discuss our findings so far, because we have not created any working GHC client at the time of writing.

9.2.1 Haskell and GHC

Haskell is a lazy pure functional language. Unlike imperative languages, Haskell programs encode the relation between values and expressions, instead of directly encoding the control flow. Its laziness allows the evaluation of an expression to be postponed to the time when its value is needed, if needed at all. These characteristics contrast sharply with physical machines and Mu, which are both imperative and strict. Therefore, the implementation of such a language has always been a challenge.

GHC compiles Haskell source code into native code via several intermediate languages, namely Core, STG and C-- (C minus minus).

The Core language is a compact, de-sugared representation of Haskell with only nine primitives. After parsing, all Haskell programs are de-sugared into Core, and many optimisations are performed on Core, too.

The Spineless Tagless G-machine (STG) was proposed by Peyton Jones and Salkild [1989] as an abstract machine for non-strict functional languages. Like both Haskell and Core, the STG language is still lazy and functional, but STG also has an 'operational semantics' that can be evaluated imperatively with a stack-based state-transition model, which serves as a bridge between the functional language and the imperative hardware.

C-- is a strict, imperative, C-like language without garbage collection. It is designed by Peyton Jones et al. [1997] as a compilation target similar to LLVM. GHC then compiles C-- to native code using either its 'Native Code Generator' (NCG) or using LLVM as an intermediate step.

GHC also has a runtime system (RTS) which comprises about 50,000 lines of C and C-- code. This library provides the underlying mechanisms for primitive types, raising exceptions, garbage collection, concurrency, a byte-code interpreter (GHCi), profiling, software transactional memory, etc.

9.2.2 Targeting Mu

Given the many levels of translation in GHC, the immediate question is: 'Which is the right place to start translating to Mu IR?'

We have been struggling with whether STG or C-- is the right level to start targeting Mu. C-- was initially attractive because it is already imperative, and there is an official backend that translates C-- to the LLVM IR which is similar to the Mu IR. However, C-- proved to be too low-level for Mu. GC is already translated into concrete memory operations, and other low-level mechanisms, such as the check for stack overflow and the management of the stack and the calling convention are already injected, too. C-- could be a valid source if the target is a non-GC IR or the machine, but Mu abstracts out all of the above mechanisms so that the client do not need to, and also not able to, control them directly. Thus we instead start with the STG language, one step before C--.

STG represents all values, evaluated or not, as *closures*. A closure in STG is a self-contained heap-allocated object consisting of a code-pointer and zero or more argument fields. A closure can be 'entered' by jumping to its code-pointer which evaluates the closure; after evaluation, the closure is updated so that the result value is stored inside the closure rather than having to be computed again. When targeting Mu, the closure structure naturally maps to the `hybrid` type with a function reference in its fixed part as the code pointer, and an array of references in the variable part to represent the argument fields. The code can be implemented as Mu functions that evaluate and update the `hybrid`. The `tagref64` tagged reference type is ideal as the reference type, so that primitive types, such as integers, can be represented directly without heap allocation.

STG also has an explicit argument stack. Arguments to functions are passed via the stack. Mu does not provide such an explicit stack for the client. Although the client can always emulate such a stack by explicitly allocating an array using Mu's heap allocation primitives, it is more advisable to use Mu local SSA variables for temporary values, and use Mu's standard `CALL` instruction instead of any explicit stacks, since the register allocator in Mu can presumably make better use of machine registers and generate more efficient code for function calls. Since we do not have any executable code produced from Haskell programs yet, it is an open topic to find the best way to translate STG code into the Mu IR.

9.3 Conclusion

We have been able to support RPython on Mu to the extent that the RPySOM interpreter and the core of the PyPy interpreter can execute on Holstein — the Mu reference implementation, while the support for the meta-tracing JIT compiler in RPython and the support for the GHC Haskell compiler are still work in progress. Although supporting a real-world language on Mu is not a trivial task, it has already been shown with RPython that Mu can ease the task of VM construction by removing difficult low-level implementation details, such as GC, exception handling and machine code generation from the focus of the high-level language implementers.

The experience with real-world languages also revealed missing features in the design of Mu. In the end, we refined the Mu IR, added the boot image building API, and thought more carefully about supporting ahead-of-time compilation.

We also experienced difficulties when the original language implementations made too many assumptions about their original targets, which makes these implementations difficult to port to Mu. For example, RPython is too tightly coupled with its C backend and its particular GC implementation. We have been working closely with the upstream, such as the PyPy developers, during our development.

Part IV

Conclusions

Chapter 10

Conclusion

Many languages suffer from poor performance and inscrutable semantics. A fundamental reason is that designing and implementing programming languages is difficult. Without adequate monetary and intellectual investment, many languages are developed with naïve implementation strategies, baking bad decisions in at the early stage of the language design, which has long-term consequences that hinders the development of the language. On the other hand, existing platforms, such as LLVM or JVM, are insufficient in helping the language developers. They either miss crucial abstractions, such as garbage collection, which are important but difficult to implement, or couple too tightly with specific languages, such as Java, which forms a semantic gap and also introduces redundant dependencies to the concrete language to be supported.

This thesis attempts to create a proper platform for the development of managed languages. We identify that concurrency, execution and garbage collection are the three major concerns that contribute to the complexity in language development. We thus introduce the concept of micro virtual machines which are minimalist virtual machines that provide abstractions over exactly these three major concerns. With their minimalism avoiding a semantic gap and unnecessary dependencies, and the abstraction over the most difficult concerns, we expect micro virtual machines to be the proper platform for the development of managed languages.

We designed Mu, a concrete micro virtual machine. Mu is defined as a specification which allows multiple compliant implementations. The language implementation is explicitly divided into the minimalist low-level micro virtual machine and a high-level 'client' that uses the micro virtual machine to implement concrete languages. Mu is heavily influenced by LLVM. It has an LLVM-like low-level type system that does not enforce any object model, and also has LLVM-like primitive operations that are close to the machine. However, unlike LLVM, it provides object references, heap allocation and garbage collection as primitives, freeing the client from having to implement the GC manually. To support concurrency, Mu provides threads, a C11-like memory model with atomic memory operations and explicitly annotated memory orders, as well as SWAPSTACK operation which supports massive multi-threaded execution. To support JIT compilation, Mu provides an API that allows the client to submit code for JIT compiling at run time, and also provides trap handling, function redefinition, stack introspection and on-stack replacement

(OSR) mechanisms that allow the client to handle lazy loading and feedback-directed optimisation during execution. We also developed 'Holstein', the Mu reference implementation, in Scala. Despite not being a high-performance implementation, it provided a platform for early adopters to experiment with.

Stack introspection and on-stack replacement (OSR) are two important low-level mechanisms to support feedback-directed optimisation and JIT compilation, which are extremely important for managed languages. However, they are very difficult to implement due to their connection to the machine architecture and the code generator, and, to our knowledge, no existing platforms provide them in their APIs for their clients. We analysed SpiderMonkey and V8, two state-of-the-art JavaScript engines, and revealed the difficulties in implementing stack introspection and OSR from scratch, mainly the over-reliance on hand-written assembly code. We showed how the Mu API can help with introspection and OSR during feedback-directed optimisation without exposing machine-level details to the client, using a proof-of-concept JavaScript client on Mu. We also showed how the stack API of Mu can be implemented on concrete machines using the concepts of 'return-oriented programming', 'resumption points' and 'resumption protocols'. We built `libyugong`, a library that implements the Mu-like OSR and stack introspection API for native C/C++ programs, as a proof of concept. Although we managed to get OSR working, stack introspection did not work properly due to the non-cooperative nature of existing compilers and system libraries. This indirectly reflected that a carefully-designed low-level platform like Mu is much needed.

The real-world significance of Mu cannot be demonstrated without real-world language clients. We are re-targeting RPython and GHC as clients of Mu. RPython is a subset of Python which is static, ahead-of-time compiled and garbage collected, and has supported many languages including PyPy (Python), RPySOM (SOM), Pyrlang (Erlang) and Pycket (Racket). Observing the similarity between the 'Low-Level' (LL) Type System of RPython and the Mu type system, we devised a straightforward translation process for the types as well as the corresponding operations between RPython and Mu, although operations that couple tightly with the GC needed careful handling. The experience influenced the design of Mu itself by adding in missing instructions needed by RPython, as well as forcing us to think more carefully about the ahead-of-time compilation of Mu IR code, and concretely design the boot image building mechanism. As a result, we are able to translate many RPython programs, including the RPySOM interpreter, the core of the PyPy interpreter, and many small RPython programs in the RPython test suite, into Mu IR, and run them on the 'Holstein' Mu reference implementation. We also made efforts on reusing GHC as a frontend for Mu. We had some preliminary thoughts about translation of STG closures, but we still cannot produce executable programs from Haskell at the time of writing.

10.1 Future Work

The Mu project is in active development. Much work can be done to improved Mu, and bring more languages onto the Mu platform.

10.1.1 Refining the Design of Mu

As we explore the clients and the real-world language implementations, we frequently find deficiencies in our existing design. We have already made changes to Mu several times to adapt to the new ideas, but there is still much room for improvement.

Removing the type parameter of references and pointers

One example is the removal of the type parameter from pointer and reference types, such as the '<T>' in `ref<T>`, `iref<T>` and `uptr<T>`. In conventional languages such as C, C++ and Java, the referent type is a part of the pointer or reference type, and the pointers and references to one type can be *cast* to that of another type. Mu already supports check-less type casting between reference types or pointer types with the `REFCAST` or `PTRCAST` instructions, assuming the client has already inserted proper checks for validity. However, during our work on a Java client, our fellow researcher reported that it is impossible to construct a Mu-level reference type `ref<Foo>` to class `Foo` without actually loading the class `Foo`, because the Mu-level structure type `Foo` must encode the layout of the class. If lazy loading is to be supported, references from one class to another must be encoded as `ref<void>`, and cast to the appropriate `ref<Foo>` when accessing the fields of `Foo`. Colleagues working on the PyPy client also reported that it is often required to compare the equality between two references to two different static types, as the concrete values may refer to objects of derived types. Although `REFCAST` is essentially a no-op at the machine level, the type parameter for the `ref` type appears to be redundant, as the concrete Mu instructions, such as `EQ`, `NE`, `LOAD` and `STORE`, are already annotated with the operand type for the convenience of the Mu backend compiler. Eventually, we realised that only 'storage types' (i.e. the types that determines the *representation* or *layout* of values) matter at the micro virtual machine level, and the enforcement of high-level type consistency (such as what objects a `ref` may refer to) could be off-loaded to the client, too. However, since this change implies a major redesign of the type system, we have postponed this change to the future as other research projects, such as the PyPy client, depend on a relatively stable Mu at this moment.

Factored Control Flow Graph

Another alternative design choice we have considered is to loosen the single-exit requirement of basic blocks and allow side-exits (including branches and potential excepting instructions such as `CALL` with 'exception clauses') in the middle of basic blocks. In principle, this is similar to the 'factored control flow graphs' (FCFG) introduced by Choi et al. [1999] and used in JikesRVM. FCFGs will have fewer but longer basic blocks than the single-exit counterpart, while still permitting efficient forward and backward intra-procedural analysis with minor changes to the algorithms. The compactness of FCFG allows more efficient communication between the client and Mu, and the benefit could be more significant since the Mu IR needs to pass parameters between basic blocks. The tracing JIT compilers can also benefit from this form since the trace usually has many 'guard' operations which could be simply translated to comparison operations and side exits without having to break the trace into many tiny basic blocks just for binary branching. This change would also result in a major Mu IR redesign, thus we postponed this idea as a future work.

Transactional Memory

Transactional memory provides a scalable, composable alternative to the lock-based synchronisation mechanisms on parallel hardware. Software transactional memory (STM) is able to run general transactional programs, but with significant performance overhead. Meanwhile, recent commodity processors, such as Intel's Haswell microarchitecture, also started to offer best-effort hardware transactional memory (HTM) of limited transaction sizes. As an abstraction layer over execution and concurrency, Mu should consider providing primitives for transactional memory, allowing a hybrid STM/HTM system to be implemented, and the optimal STM or HTM strategy to be chosen dynamically at run time, similar to the work by Chapman et al. [2016].

10.1.2 Client-side Development

Clients with JIT-compiling Capability

Although Mu is designed from the very beginning as a platform for JIT compilation, the mechanisms provided by Mu IR and the API need to be tested with real-world language clients. So far, the only working Mu client capable of JIT-compiling the high-level language and performing feedback-directed optimisation is the proof-of-concept JS-Mu project which only implements a subset of JavaScript, as described in Section 7.4. The meta-tracing JIT compiler in PyPy is a good candidate to evaluate such JIT-compiling capabilities in a real-world language client.

Supporting Additional Languages

The more language we test on Mu, the better we can demonstrate and improve the generality of Mu. Some languages may potentially benefit from what Mu offers. For example, Erlang and Go may use the SWAPSTACK operation to implement their massive multi-threading model. Java and C#, as conventional static managed languages, may not see immediate performance improvement when ported on Mu as their original VMs, namely the JVM and the .NET CLR, are already highly optimised. However, supporting such languages on Mu will allow us to compare against state-of-the-art VMs on a common ground. Such static languages may also play the role of 'high-level low-level languages' [Frampton et al., 2009] that assist in the development of other languages.

Client-side Libraries

Common utilities that are beneficial to multiple languages can be factored out as reusable client-side libraries. One such library could provide common optimisations, such as common sub-expression elimination and loop-invariant code motion, at the Mu IR level. Such a library will optimise the Mu IR inside the client before submitting to Mu. In particular, the official RPython backend relies on the lower level, namely the C compiler, to perform aggressive optimisations, and thus generate low-quality C code as output. Client-side low-level optimisations must be performed for such clients to even stand a chance against their original implementations, since Mu, as a minimalist VM, is not obliged to perform any optimisations which can be offloaded to the client.

10.1.3 High-performance Mu Implementation

Mu will not be useful in supporting efficient language implementations unless Mu itself is efficient. The Holstein Mu reference implementation is designed for correctness rather than performance. The Zebu high-performance Mu implementation is under active development. Currently it can compile a subset of the Mu IR, and is yielding promising results with simple benchmarks.

10.1.4 Formal Verification

One goal of the Mu project is to build a reliable formally-verified language runtime on the top of the formally-verified seL4 microkernel. We are working on formally specifying the semantics of Mu in HOL, and much work has been done with the semantics of the Mu memory model. We are also working on porting the Rust-based Zebu implementation to seL4. However, given the incredible difficulties in the verification of the seL4 microkernel, the verification of Mu may not be an easy task, either.

10.2 Final Words

We have made our first steps in our ambitious Mu micro virtual machine project. We expect that in a foreseeable future, language developers will be able to use Mu as a solid foundation for their next languages, and the languages will be properly designed, efficiently implemented, and easier to develop and maintain than the status-quo of today. People may also design their own micro virtual machines which have the same principle of minimalism as Mu, but address their own concerns of their language development. Eventually, we will see an overall improvement in the quality of programming languages.

Bibliography

Adams, K.; Evans, J.; Maher, B.; Ottoni, G.; Paroski, A.; Simmers, B.; Smith, E.; and Yamauchi, O., 2014. The HipHop virtual machine. In OOPSLA '14: *Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, OR, USA, Oct. 2014), 777–790. ACM, New York, NY, USA. doi:10.1145/2660193.2660199. (cited on page 15)

Alpern, B.; Attanasio, C. R.; Cocchi, A.; Lieber, D.; Smith, S.; Ngo, T.; Barton, J. J.; Hummel Flynn, S.; Sheperd, J. C.; and Mergen, M., 2009. Implementing Jalapeño in Java. In OOPSLA '09: *Proceedings of the 14th ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications* (Denver, CO, USA, Nov. 2009), 314–324. ACM, New York, NY, USA. doi:10.1145/320384.320418. (cited on pages 8, 26, 84, 95, 98, and 110)

Ananian, C. S., 1999. *Static Single Information Form*. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. (cited on pages 29, 38, and 107)

Appel, A. W., 1998. SSA is functional programming. *ACM SIGPLAN Notices*, 33, 4 (Apr. 1998), 17–20. doi:10.1145/278283.278285. (cited on page 38)

Baumann, S.; Bolz, C. F.; Hirschfeld, R.; Kirilichev, V.; Pape, T.; Siek, J.; and Tobin-Hochstadt, S., 2015. Pycket: A tracing JIT for a functional language. In ICFP '15: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada, Aug. 2015), 22–34. ACM, New York, NY, USA. doi:10.1145/2784731.2784740. (cited on pages 4 and 104)

Behrens, S., 2008. Concurrency and Python. http://www.drdobbs.com/open-source/concurrency-and-python/206103078. Retrieved: 9 Sep 2018. (cited on pages 1 and 14)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004. Oil and water? High performance garbage collection in Java with MMTk. In ICSE '04: *Proceedings of the 26th International Conference on Software Engineering* (Edinburgh, Scotland, UK, May 2004). IEEE. doi:10.1109/ICSE.2004.1317436. (cited on page 18)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In PLDI '08: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA, Jun. 2008), 22–32. ACM, New York, NY, USA. doi:10.1145/1375581.1375586. (cited on pages 12, 13, and 28)

Blackburn, S. M.; Salishev, S. I.; Danilov, M.; Mokhovikov, O. A.; Nashatyrev, A. A.; Novodvorsky, P. A.; Bogdanov, V. I.; Li, X. F.; and Ushakov, D., 2008. The Moxie JVM experience. Technical Report TR-CS-08-01, Australian National University, Department of Computer Science. (cited on page 110)

Boehm, H.-J., 2005. Threads cannot be implemented as a library. In PLDI '05: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA, Jun. 2005), 261–268. ACM, New York, NY, USA. doi:10.1145/1065010.1065042. (cited on pages 10 and 18)

Boehm, H.-J. and Adve, S. V., 2008. Foundations of the C++ concurrency memory model. In PLDI '08: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA, Jun. 2008), 68–78. ACM, New York, NY, USA. doi:10.1145/1375581.1375591. (cited on pages 10 and 46)

Boehm, H.-J. and Weiser, M., 1988. Garbage collection in an uncooperative environment. *Software: Practice & Experience*, 18, 9 (Sep. 1988), 807–820. doi:10.1002/spe.4380180902. (cited on page 13)

Bolz, C. F.; Cuni, A.; Fijalkowski, M.; and Rigo, A., 2009. Tracing the meta-level: PyPy's tracing JIT compiler. In ICOOOLPS '09: *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems* (Genova, Italy, Jul. 2009), 18–25. ACM, New York, NY, USA. doi:10.1145/1565824.1565827. (cited on pages 14, 18, and 104)

Bolz, C. F. and Tratt, L., 2015. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming (SCICO)*, (Feb. 2015), 408–421. doi:10.1016/j.scico.2013.02.001. (cited on page 16)

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. In OOPSLA '12: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, AZ, USA, Oct. 2012), 195–212. ACM, New York, NY, USA. doi:10.1145/2398857.2384631. (cited on pages 2, 8, 16, 18, 25, and 79)

Chapman, K.; Hosking, A. L.; and Moss, J. E. B., 2016. Hybrid STM/HTM for nested transactions on OpenJDK. In OOPSLA '16: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Amsterdam, Netherlands, Nov. 2016), 660–676. ACM, New York, NY, USA. doi:10.1145/2983990.2984029. (cited on page 122)

Choi, J.-D.; Grove, D.; Hind, M.; and Sarkar, V., 1999. Efficient and precise modeling of exceptions for the analysis of java programs. In PASTE '99: *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Toulouse, France, Sep. 1999), 21–31. ACM, New York, NY, USA. doi:10.1145/316158.316171. (cited on page 121)

CPython. Welcome to Python.org. https://www.python.org/. Retrieved: 9 Sep 2018. (cited on pages 18 and 104)

Cytron, R.; Ferrante, J.; Rosen, B. K.; Wegman, M. N.; and Zadeck, F. K., 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13, 4 (Oct. 1991), 451–490. doi:10.1145/115372.115320. (cited on page 29)

D'Elia, D. C. and Demetrescu, C., 2016. Flexible on-stack replacement in llvm. In CGO '16: *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain, Mar. 2016), 250–260. ACM, New York, NY, USA. doi:10.1145/2854038.2854061. (cited on pages 77, 79, 98, and 99)

Dolan, S.; Muralidharan, S.; and Gregg, D., 2013. Compiler support for lightweight context switching. *ACM Transactions on Architecture and Code Optimization*, 9, 4 (Jan. 2013), 36:1–36:25. doi:10.1145/2400682.2400695. (cited on pages 4, 16, 47, 69, 93, and 98)

Drepper, U., 2011. Futexes are tricky. https://www.akkadia.org/drepper/futex.pdf. Retrieved: 9 Sep 2018. (cited on page 44)

DWARF Standards Committee, 2010. Dwarf debugging information format, version 4. http://www.dwarfstd.org/doc/DWARF4.pdf. Retrieved: 9 Sep 2018. (cited on page 84)

Ebcioğlu, K.; Saraswat, V.; and Sarkar, V., 2004. X10: Programming for hierarchical parallelism and non-uniform data access (extended abstract). (cited on page 16)

Fink, S. J. and Qian, F., 2003. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In CGO '03: *Proceedings of the 2003 International Symposium on Code Generation and Optimization* (San Francisco, CA, USA, Mar. 2003), 241–252. IEEE Computer Society, Washington, DC, USA. doi:10.1109/CGO.2003.1191549. (cited on pages 76 and 79)

Frampton, D.; Blackburn, S. M.; Cheng, P.; Garner, R. J.; Grove, D.; Moss, J. E. B.; and Salishev, S. I., 2009. Demystifying magic: high-level low-level programming. In VEE '09: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Washington, DC, USA, Mar. 2009), 81–90. ACM, New York, NY, USA. doi:10.1145/1508293.1508305. (cited on pages 104 and 123)

Franke, H. and Russell, R., 2002. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In OLS '02: *Proceedings of the 2002 Ottawa Linux Symposium* (Ottawa, ON, USA, Jun. 2002), 479–495. http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf. (cited on page 44)

Garner, R.; Blackburn, S. M.; and Frampton, D., 2011. A comprehensive evaluation of object scanning techniques. In ISMM '11: *Proceedings of the 2011 International Symposium on Memory Management* (San Jose, CA, USA, Jun. 2011), 33–42. ACM, New York, NY, USA. doi:10.1145/1993478.1993484. (cited on page 18)

GEOFFRAY, N.; THOMAS, G.; LAWALL, J.; MULLER, G.; AND FOLLIOT, B., 2010. VMKit: A substrate for managed runtime environments. In VEE '10: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Pittsburgh, PA, USA, Mar. 2010), 51–62. ACM, New York, NY, USA. doi:10.1145/1735997.1736006. (cited on pages 3, 7, and 18)

GOLDBERG, A. AND ROBSON, D., 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0-201-11371-6. (cited on page 110)

GOOGLE. V8 JavaScript Engine. https://developers.google.com/v8/. Retrieved: 9 Sep 2018. (cited on pages 8, 15, 35, and 66)

GOSLING, J.; JOY, B.; STEELE JR., G. L.; BRACHA, G.; AND BUCLEY, A., 2014. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edn. ISBN 013390069X, 9780133900699. (cited on pages 10, 37, and 46)

GUDEMAN, D., 1993. Representing type information in dynamically typed languages. Technical Report TR 93-27, Department of Computer Science, Gould-Simpson Building, The University of Arizona, Tucson, AZ 85721, USA. (cited on page 34)

HACK. Hack — Programming Productivity Without Breaking Things. https://hacklang.org/. Retrieved: 9 Sep 2018. (cited on page 43)

HEJLSBERG, A.; WILTAMUTH, S.; AND GOLDE, P., 2003. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. ISBN 0321154916. (cited on page 51)

HÖLZLE, U.; CHAMBERS, C.; AND UNGAR, D., 1992. Debugging optimized code with dynamic deoptimization. In PLDI '92: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, CA, USA, Jun. 1992), 32–43. ACM, New York, NY, USA. doi:10.1145/143095.143114. (cited on pages 9, 95, 98, and 99)

HÖLZLE, U. AND UNGAR, D., 1994. A third-generation self implementation: Reconciling responsiveness with performance. In OOPSLA '94: *Proceedings of the Ninth Annual Conference on Object-oriented Programming Systems, Languages, and Applications* (Portland, OR, USA, Oct. 1994), 229–243. ACM, New York, NY, USA. doi:10.1145/191080.191116. (cited on pages 9, 98, and 99)

IERUSALIMSCHY, R.; DE FIGUEIREDO, L. H.; AND FILHO, W. C., 1996. Lua — an extensible extension language. *Software: Practice & Experience*, 26, 6 (Jun. 1996), 635–652. doi:10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P. https://www.lua.org/spe.html. (cited on page 13)

ISO, 2012. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372. (cited on pages 10 and 46)

Itanium. Itanium C++ ABI: Exception Handling (Revision: 1.22). https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html. Retrieved: 9 Sep 2018. (cited on pages 20 and 40)

Jibaja, I.; Blackburn, S. M.; Haghighat, M. R.; and McKinley, K. S., 2011. Deferred gratification: Engineering for high performance garbage collection from the get go. In MSPC '11: *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (San Jose, CA, Jun. 2011), 58–65. ACM, New York, NY, USA. doi:10.1145/1988915.1988930. (cited on pages 1, 12, and 13)

Jibaja, I.; Jensen, P.; Hu, N.; Haghighat, M. R.; McCutchan, J.; Gohman, D.; Blackburn, S. M.; and McKinley, K. S., 2015. Vector parallelism in JavaScript: Language and compiler support for SIMD. In PACT '15: *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation* (San Francisco, CA, USA, Oct. 2015), 407–418. IEEE, Washington, DC, USA. doi:10.1109/PACT.2015.33. (cited on page 33)

JRuby. Home — JRuby.org. http://jruby.org/. Retrieved: 9 Sep 2018. (cited on page 16)

Jython. The Jython Project. http://www.jython.org/. Retrieved: 9 Sep 2018. (cited on pages 16 and 43)

Kell, S.; Mulligan, D. P.; and Sewell, P., 2016. The missing link: Explaining ELF static linking, semantically. In OOPSLA '16: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Amsterdam, Netherlands, Nov. 2016), 607–623. ACM, New York, NY, USA. doi:10.1145/2983990.2983996. (cited on page 111)

Kevin Modzelewski, 2016. Pyston 0.5 release. https://blog.pyston.org/2016/05/25/pyston-0-5-released/. Retrieved: 9 Sep 2018. (cited on page 13)

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S., 2009. seL4: Formal verification of an OS kernel. In SOSP '09: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA, Oct. 2009), 207–220. ACM, New York, NY, USA. doi:10.1145/1629575.1629596. (cited on page 24)

Kowalke, O., 2016. Boost.Context. http://www.boost.org/doc/libs/1_63_0/libs/context/doc/html/index.html. Retrieved: 9 Sep 2018. (cited on pages 47 and 86)

Lameed, N. A. and Hendren, L. J., 2013. A modular approach to on-stack replacement in LLVM. In VEE '13: *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Houston, TX, USA, Mar. 2013), 143–154. ACM, New York, NY, USA. doi:10.1145/2451512.2451541. (cited on page 99)

Lattner, C. and Adve, V., 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In CGO '04: *Proceedings of the 2004 International Symposium on Code Generation and Optimization* (Palo Alto, CA, USA, Mar. 2004). IEEE Computer Society, Washington, DC, USA. doi:10.1109/CGO.2004.1281665. (cited on pages 2, 17, 18, 24, 91, and 98)

LibGCCEH. GNU Compiler Collection (GCC) Internals: Exception handling routines. https://gcc.gnu.org/onlinedocs/gccint/Exception-handling-routines.html. Retrieved: 9 Sep 2018. (cited on page 97)

Lin, Y.; Wang, K.; Blackburn, S. M.; Norrish, M.; and Hosking, A. L., 2015. Stop and go: Understanding yieldpoint behavior. In ISMM '15: *Proceedings of the 2015 International Symposium on Memory Management* (Portland, OR, USA, Jun. 2015), 70–80. ACM, New York, NY, USA. doi:10.1145/2754169.2754187. (cited on pages 13, 17, 18, and 58)

Lindholm, T.; Yellin, F.; Bracha, G.; and Buckley, A., 2014. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edn. ISBN 013390590X, 9780133905908. (cited on pages 16 and 26)

LLVMGEP. The Often Misunderstood GEP Instruction — LLVM 8 documentation. http://llvm.org/docs/GetElementPtr.html. Retrieved: 9 Sep 2018. (cited on page 41)

LLVMGoals. Goals and non-goals — Garbage Collection with LLVM — LLVM 8 documentation. http://llvm.org/docs/GarbageCollection.html#goals-and-non-goals. Retrieved: 9 Sep 2018. (cited on page 17)

LLVMStackMap. Stack maps and patch points in LLVM — LLVM 8 documentation. http://llvm.org/docs/StackMaps.html. Retrieved: 9 Sep 2018. (cited on page 98)

LuaJITGC. New Garbage Collector — The LuaJIT Wiki. http://wiki.luajit.org/New-Garbage-Collector. Retrieved: 9 Sep 2018. (cited on page 15)

Manson, J.; Pugh, W.; and Adve, S. V., 2005. The Java memory model. In POPL '05: *Proceedings of the 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (California, CA, USA, Jan. 2005), 378–391. ACM, New York, NY, USA. doi:10.1145/1040305.1040336. (cited on pages 10 and 46)

Marr, S. and Ducasse, S., 2015. Tracing vs. partial evaluation: Comparing meta-compilation approaches for self-optimizing interpreters. In OOPSLA '15: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Pittsburgh, PA, USA, Oct. 2015), 821–839. ACM, New York, NY, USA. doi:10.1145/2858965.2814275. http://stefan-marr.de/papers/oopsla-marr-ducasse-meta-tracing-vs-partial-evaluation/. (cited on page 20)

Marr, S.; Pape, T.; and Meuter, W. D., 2014. Are we there yet?: Simple language implementation techniques for the 21st century. *IEEE Software*, 31, 5 (Sept 2014), 60–67. doi:10.1109/MS.2014.98. (cited on pages 4, 104, and 112)

MATZ, M.; HUBIČKA, J.; JAEGAR, A.; AND MITCHELL, M., 2012. System V Application Binary Interface, AMD64 Architecture Processor Supplement. http://refspecs.linuxfoundation.org/elf/x86_64-abi-0.99.pdf. Retrieved: 9 Sep 2018. (cited on page 51)

MCCARTHY, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3, 4 (Apr. 1960), 184–195. doi:10.1145/367177.367199. (cited on page 12)

MICROSOFT. .NET — Powerful Open Source Cross Platform Development. https://www.microsoft.com/net. Retrieved: 9 Sep 2018. (cited on page 7)

MICROVM. The Mu Micro Virtual Machine Project. http://microvm.org/. Retrieved: 9 Sep 2018. (cited on page 28)

MONO. Mono — Cross platform, open source .NET framework. https://www.mono-project.com/. Retrieved: 9 Sep 2018. (cited on page 13)

MOSBERGER, D. The libunwind project. http://www.nongnu.org/libunwind/. Retrieved: 9 Sep 2018. (cited on page 91)

MOZILLA. SpiderMonkey — Mozilla | MDN. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. Retrieved: 9 Sep 2018. (cited on pages 8, 15, 35, and 66)

NATIONAL RESEARCH COUNCIL, 1994. *Academic Careers for Experimental Computer Scientists and Engineers*. The National Academies Press, Washington, DC. ISBN 978-0-309-04931-3. doi:10.17226/2236. (cited on page 3)

ORACLE. Java Native Interface. http://docs.oracle.com/javase/8/docs/technotes/guides/jni/. Retrieved: 9 Sep 2018. (cited on page 51)

PALL, M. The LuaJIT Project. http://luajit.org/. Retrieved: 9 Sep 2018. (cited on page 15)

PEYTON JONES, S. L.; NORDIN, T.; AND OLIVA, D., 1997. C--: A portable assembly language. In IFL '97: *Selected Papers from the 9th International Workshop on Implementation of Functional Languages* (St. Andrews, Scotland, UK, Sep. 1997), 1–19. Springer-Verlag, London, UK. http://dl.acm.org/citation.cfm?id=647976.743227. (cited on page 113)

PEYTON JONES, S. L. AND SALKILD, J., 1989. The spineless tagless g-machine. In FPCA '89: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, UK, Sep. 1989), 184–201. ACM, New York, NY, USA. doi:10.1145/99370.99385. (cited on page 113)

PHP, 2002. Doc Bug #20993 :: Element value changes without asking. https://bugs.php.net/bug.php?id=20993. Retrieved: 9 Sep 2018. (cited on pages 1 and 43)

Prandini, M. and Ramilli, M., 2012. Return-oriented programming. *IEEE Security & Privacy*, 10, 6 (Nov. 2012), 84–87. doi:10.1109/MSP.2012.152. (cited on page 75)

PyPyGC. translation.gcrootfinder — PyPy Documentation. http://doc.pypy.org/en/latest/config/translation.gcrootfinder.html. Retrieved: 9 Sep 2018. (cited on page 20)

PyPyGIL. Does PyPy have a GIL? Why? — Frequently Asked Questions – PyPy Documentation. http://doc.pypy.org/en/latest/faq.html#does-pypy-have-a-gil-why. Retrieved: 9 Sep 2018. (cited on page 14)

Pyston. The Pyston Blog. https://blog.pyston.org/. Retrieved: 9 Sep 2018. (cited on page 13)

Rigo, A. and Pedroni, S., 2006. PyPy's approach to virtual machine construction. In OOPSLA '06: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications* (Portland, OR, USA, Oct. 2006), 944–953. ACM, New York, NY, USA. doi:10.1145/1176617.1176753. (cited on pages 2, 4, 19, 43, and 103)

Ruby. Ruby Programming Language. https://www.ruby-lang.org/. Retrieved: 9 Sep 2018. (cited on page 14)

Scala. The Scala Programming Langauge. https://www.scala-lang.org/. Retrieved: 9 Sep 2018. (cited on page 16)

Shahriyar, R.; Blackburn, S. M.; and Frampton, D., 2012. Down for the count? Getting reference counting back in the ring. In ISMM '12: *Proceedings of the 2012 International Symposium on Memory Management* (Beijing, China, Jun. 2012), 73–84. ACM, New York, NY, USA. doi:10.1145/2258996.2259008. (cited on page 13)

Shahriyar, R.; Blackburn, S. M.; and McKinley, K. M., 2014. Fast conservative garbage collection. In OOPSLA '14: *Proceedings of the 2014 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, OR, USA, Oct. 2014), 121–139. ACM, New York, NY, USA. doi:10.1145/2660193.2660198. (cited on page 13)

SIL, 2017. Swift Intermediate Language (SIL) — Swift 2.2 documentation. http://apple-swift.readthedocs.io/en/latest/SIL.html. Retrieved: 9 Sep 2018. (cited on page 38)

Tozawa, A.; Tatsubori, M.; Onodera, T.; and Minamide, Y., 2009. Copy-on-write in the PHP language. In POPL '09: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA, Jan. 2009), 200–212. ACM, New York, NY, USA. doi:10.1145/1480881.1480908. (cited on pages 1, 14, and 43)

UnladenSwallow, 2011. Unladen swallow project page. https://code.google.com/p/unladen-swallow/. (cited on page 18)

Wang, K., a. Holstein: the reference implementation of Mu. https://gitlab.anu.edu.au/mu/mu-impl-ref2. Retrieved: 9 Sep 2018. (cited on page 79)

Wang, K., b. The specification of the Mu micro virtual machine. https://gitlab.anu.edu.au/mu/mu-spec. Retrieved: 9 Sep 2018. (cited on page 46)

Wang, K.; Blackburn, S. M.; Hosking, A. L.; and Norrish, M., 2018. Hop, skip, & jump: Practical on-stack replacement for a cross-platform language-neutral VM. In VEE '18: *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Williamsburg, VA, USA, Mar. 2018), 1–16. ACM, New York, NY, USA. doi:10.1145/3186411.3186412. (cited on page 65)

Wang, K.; Lin, Y.; Blackburn, S. M.; Norrish, M.; and Hosking, A. L., 2015. Draining the swamp: Micro virtual machines as solid foundation for language development. In SNAPL '15: *Proceedings of the 1st Summit on Advances in Programming Languages* (Asilomar, CA, USA, May 2015), 321–336. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. doi:10.4230/LIPIcs.SNAPL.2015.321. (cited on page 23)

Wimmer, C.; Haupt, M.; Van De Vanter, M. L.; Jordan, M.; Daynès, L.; and Simon, D., 2013. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9, 4 (Jan. 2013), 30:1–30:24. doi:10.1145/2400682.2400689. (cited on page 32)

Würthinger, T.; Wimmer, C.; Wöss, A.; Stadler, L.; Duboscq, G.; Humer, C.; Richards, G.; Simon, D.; and Wolczko, M., 2013. One VM to rule them all. In Onward! '13: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software* (Indianapolis, IN, USA, Oct. 2013), 187–204. ACM, New York, NY, USA. doi:10.1145/2509578.2509581. (cited on pages 19 and 98)

Yang, X.; Blackburn, S. M.; Frampton, D.; Sartor, J.; and McKinley, K. S., 2011. Why nothing matters: The impact of zeroing. In OOPSLA '11: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, OR, USA, Oct. 2011), 307–324. ACM, New York, NY, USA. doi:10.1145/2076021.2048092. (cited on page 41)

Young, N., 2016. *A Micro Virtual Machine Backend for the Glasgow Haskell Compiler*. Undergraduate honours thesis, Australian National University, Canberra, Australia. (cited on page 103)

Zhang, J., 2015. *MuPy: A First Client for the Mu Micro Virtual Machine*. Undergraduate honours thesis, Australian National University, Canberra, Australia. (cited on page 103)