Technical Report

UCAM-CL-TR-797
ISSN 1476-2986

Number 797

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Interpretational overhead in system software

Boris Feigin

April 2011

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2011 Boris Feigin

This technical report is based on a dissertation submitted September 2010 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Homerton College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Interpreting a program carries a runtime penalty: the interpretational overhead. Traditionally, a compiler removes interpretational overhead by sacrificing inessential details of program execution. However, a broad class of system software is based on *non-standard interpretation* of machine code or a higher-level language. For example, virtual machine monitors emulate privileged instructions; program instrumentation is used to build dynamic call graphs by intercepting function calls and returns; and dynamic software updating technology allows program code to be altered at runtime. Many of these frameworks are performance-sensitive and several *efficiency requirements*—both formal and informal—have been put forward over the last four decades. Largely independently, the concept of *interpretational overhead* received much attention in the partial evaluation ("program specialization") literature. This dissertation contributes a unifying understanding of efficiency and interpretational overhead in system software.

Starting from the observation that a virtual machine monitor is a self-interpreter for machine code, our first contribution is to reconcile the definition of *efficient virtualization* due to Popek and Goldberg with *Jones optimality*, a measure of the strength of program specializers. We also present a rational reconstruction of hardware virtualization support ("trap-and-emulate") from context-threaded interpretation, a technique for implementing fast interpreters due to Berndl et al.

As a form of augmented execution, virtualization shares many similarities with *program instrumentation*. Although several low-overhead instrumentation frameworks are available on today's hardware, there has been no formal understanding of what it means for instrumentation to be efficient. Our second contribution is a definition of efficiency for program instrumentation in the spirit of Popek and Goldberg's work. Instrumentation also incurs an *implicit overhead* because instrumentation code needs access to intermediate execution states and this is antagonistic to optimization. The third contribution is to use partial equivalence relations (PERs) to express the *dependence* of instrumentation on execution state, enabling an instrumentation/optimization trade-off. Since program instrumentation, applied at runtime, constitutes a kind of dynamic software update, we can similarly restrict allowable future updates to be consistent with existing optimizations. Finally, treating "old" and "new" code in a dynamically-updatable program as being written in different languages permits a semantic explanation of a safety rule that was originally introduced as a syntactic check.

Acknowledgements

I am grateful to my supervisor, Alan Mycroft, for stimulating discussions, encouragement and persistence. I thank members and friends of the CPRG, and in particular Anton, Ben, Bjarki, Chris, Derek, Kathy, Magnus, Sriram, Thomas, and Viktor, for conversations both on-topic but especially off.

I am indebted to the referees for PEPM 2008 and RV 2010 and the attendees of PLID 2008 for valuable feedback on some of the material appearing in this dissertation.

I thank the EPSRC for keeping me clothed and fed during my studies, Homerton College for the excellent accommodation and the Computer Lab for making it a pleasure to come to the office every morning (or afternoon).

Finally, I thank my family (Алла, Гриша, Марина, Матвей, Юля и Тоби) for their love and support.

Contents

1 Introduction **9**
- 1.1 Non-standard interpretation in system software 10
  - 1.1.1 Hardware virtualization 10
  - 1.1.2 Program instrumentation 10
  - 1.1.3 Dynamic software updating 11
- 1.2 Reducing interpretational overhead 11
- 1.3 The illusion of interpreted execution 12
- 1.4 Related work 13
- 1.5 Dissertation contributions & outline 14

2 Technical background **17**
- 2.1 Programs as data 17
  - 2.1.1 Full abstraction 18
  - 2.1.2 Dependent types and the phase distinction 19
  - 2.1.3 Program staging 20
- 2.2 Partial evaluation 20
  - 2.2.1 Jones optimality 21
- 2.3 AL 22
  - 2.3.1 Semantics 22
  - 2.3.2 Traces 23
  - 2.3.3 Update points 24
- 2.4 Computational reflection and self-modifying code 24
  - 2.4.1 External observations and optimization 26
- 2.5 Writing interpreters in C and ML 26
  - 2.5.1 Well-typed interpreters 26
  - 2.5.2 `while`/`switch` interpreters 27
  - 2.5.3 Threading 28
- 2.6 Non-standard interpretation of AL 29
  - 2.6.1 Program instrumentation 29
  - 2.6.2 Hardware virtualization 31
  - 2.6.3 Dynamic software updating 32
- 2.7 Partial equivalence relations 32

3 Jones optimality and efficient virtualization **35**
- 3.1 Virtualization versus emulation 36
- 3.2 From threaded code to trap-and-emulate 37
- 3.3 Self-interpretation in AL 39
- 3.4 Trace simulation 40

3.4.1 Trace simulation modulo privileged instructions 42
3.5 Virtualization assists 43
3.5.1 AL/STEP 43
3.5.2 AL/EXEC 44
3.6 An application of full abstraction to VMMs 45
3.7 Related work 47
3.8 Conclusions and further work 47

4 Formally efficient program instrumentation 49
4.1 IL 50
4.1.1 Semantics 51
4.1.2 Instrumenting AL programs 51
4.2 Faithful instrumentation 52
4.3 Breakpoints: AL/BRK 53
4.3.1 Semantics 54
4.3.2 A rationale for the design of AL/BRK 55
4.3.3 IL instrumentation with AL/BRK 56
4.3.4 AL/BRK vs. AL/EXEC 56
4.3.5 Efficiency is bounded overhead 56
4.4 From super-instructions to language boundaries 57
4.4.1 Lexical vs. dynamic scoping of boundaries 58
4.4.2 Multi-IL 59
4.4.3 Multi-AL/BRK 60
4.5 Optimization of dynamically instrumented code 60
4.5.1 From dynamic instrumentation to software updating 62
4.6 Related work 62
4.7 Conclusions and further work 63
4.7.1 Further work 64

5 Dynamic typing, boundaries and con-freeness 65
5.1 Preliminaries: con-freeness 66
5.2 HL 67
5.2.1 Runtime type analysis 68
5.2.2 Dynamic semantics 68
5.3 Dynamic updates 70
5.3.1 Applying an update 70
5.3.2 HL language boundaries 71
5.4 Semantic con-freeness 71
5.4.1 Definition of semantic con-freeness 72
5.4.2 Runtime enforcement 72
5.5 The con-free check as a type system 73
5.6 Interpretational overhead of DSU 75
5.7 Equivalence of updatable programs 75
5.7.1 Optimization 77
5.7.2 Deoptimization in HL_{ANF} 78
5.7.3 Other semantics for updating 79
5.8 Related work 80
5.8.1 Dynamic typing 80
5.8.2 Con-freeness and dependency 80

5.8.3 Optimization 81
5.9 Conclusions and further work 81

6 Information flow and (de)compilation **85**
6.1 Normalization and decompilation 86
6.1.1 Motivation: non-interference and full abstraction 86
6.1.2 Secure information flow for a compiler 87
6.2 Applications 88
6.2.1 Superoptimization 88
6.2.2 Randomized compilation 88
6.2.3 Adaptive compilation: a proposal 89
6.3 An operational view of debug tables 90
6.3.1 Language boundaries 91
6.4 Related work 92
6.5 Conclusions 92

7 Conclusions and further work **95**
7.1 Common-case performance 95
7.2 Indirection without a performance penalty 96
7.3 Architectural support motivated by optimization 97
7.4 Further work 97

A Notation **99**

B Applications of PERs **101**
B.1 Types 101
B.2 Non-interference 102
B.3 Static analyses 103

Chapter 1

Introduction

Most programming languages in widespread use—C, Haskell, Python among others—lack a formal mathematical semantics. The meaning of C and Haskell programs is specified in English prose by the relevant ISO document or technical report. Python, on the other hand, is defined by its standard implementation, CPython[1]. This makes CPython a *definitional interpreter*, with Python as the defin<u>ed</u> language and C as the defin<u>ing</u> language. Any interpreter for Python that does not agree with CPython on every user-visible detail of a program's behaviour is *non-standard*. A non-standard interpreter for a language defines a new language that is substantially the same as the original, modulo small variations in the meaning of some language phrases. For instance, a non-standard interpreter might cause all `print` commands to write their arguments to a file instead of displaying them on screen; another interpreter might keep track of the number of arithmetic operations performed and display the total on program termination; a custom interpreter intended to help debug programs might stop periodically to allow the user to alter the program code. It is convenient to *reason* about many low-level system applications in terms of non-standard interpreters. However, interpretation is not a viable *implementation strategy* for performance-sensitive software. Interpreters, while easy to write, are slow, because for every unit of useful work in the program, an interpreter does a lot of extra housekeeping: parsing, dispatch, encoding/decoding of values. The term "interpretational overhead" informally refers to the penalty of interpreting a program. David Gries famously said: "never put off until run time what you can do at compile time". Optimizing compilers avoid computation that does not contribute to the final result and do as much as possible of what remains statically. Since the differences between the standard and non-standard interpretations are often perceived as minor, there is an expectation that (a) the toolchain, including the optimizing compiler, can be reused, and (b) that performance of programs will not degrade significantly under the non-standard interpretation. There is an abundance of point solutions and folklore on the performance and program structuring issues posed by non-standard interpretation. Our thesis is that a unifying approach to the problem can help inform the design of future hardware and software systems.

Outline. The ideas in this dissertation apply to a broad class of software. Chapters 3 to 5 respectively deal with hardware virtualization, program instrumentation and dynamic software updating. (Notice that these systems are essentially more elaborate versions of the three non-standard interpreter examples given above.) The remainder of this chapter provides brief introductions to these areas and sets our efforts in the context of existing

[1] `http://www.python.org`

literature—technical details are postponed until the next chapter. The full dissertation outline is included at the end of this chapter.

1.1 Non-standard interpretation in system software

1.1.1 Hardware virtualization

A Universal Turing Machine (UTM) can be set up to run arbitrarily many input Turing Machines (TMs) by interleaving their executions. The basic premise of *hardware virtualization* is that—modulo their finite nature—general-purpose computer architectures are Turing-complete. A virtual machine monitor (VMM, cf. UTM) manages allocation of the hardware resources of the "host" machine between several concurrently executing "guest" virtual machines (VMs, cf. simulated TMs), each of which is a software likeness of the host. This analogy suggests that a VMM should be thought of as an *interpreter for machine code*. We further argue that a VMM is a *non-standard interpreter* because the emulated environment of a VM differs from host hardware in the number and types of available I/O devices, amount of memory, etc. For example, say executing a particular sequence of instructions displays a pixel on the screen at position (0, 15). The VMM will likely project the virtual screen onto a smaller part of the real screen, say starting at position (200, 200). Now the same sequence of instructions results in a pixel being displayed at (200, 215). Indeed, the VMM may alternatively emulate the screen by writing the pixel values to a file or network socket. In either case, the observable behaviour of the program differs depending on whether it is executed directly on the host or inside a VM.

Popek and Goldberg's [102] *efficiency requirement* states that non-privileged instructions (those that do not affect the operation of the CPU itself or other hardware) must be executed by the host with no intervention on the part of the VMM. The VMM must always be allowed to handle privileged instructions: this is necessary to enforce isolation between individual VMs, and maintain the illusion of virtual hardware which is not physically present on the host. The technology for implementing efficient VMMs was first developed by IBM in the 1960s and 70s. The last few years have seen renewed interest in virtualization: VMware Workstation[2] and Xen[3] are two popular VMM implementations, but there are many others.

1.1.2 Program instrumentation

The term "instrumentation" refers to modification of a program or its runtime environment to make hidden details of execution visible. A *dynamic binary instrumentation (DBI)* framework can be used to gather statistics about a program run: "dynamic" means that instrumentation code can be added and removed at runtime, and "binary" implies that the program is instrumented after compilation. Just like a VMM, a DBI framework can be thought of as a non-standard interpreter for machine code and, implemented naively, instrumentation can severely compromise program execution speed: the impact of instrumentation on performance is often a sticking point in the adoption of a DBI framework. Development of practical methods for low-overhead instrumentation of production systems is an active area of research (see §1.2). DTrace [22], first introduced in the Solaris kernel, and later ported to FreeBSD and Mac OS X, is a well-known industrial

[2] `http://www.vmware.com`
[3] `http://www.xen.org`

implementation. Its authors, Cantrill et al. [22], claim that "when DTrace is not in use, the system is just as if DTrace were not present at all." In other words, instrumentation may be switched on or off at certain designated points throughout the program and, when instrumentation is off, the instrumented program incurs no additional overhead. Unfortunately, an "implicit" overhead remains because optimization opportunities are lost when compilation has to preserve internal detail of program execution rather than just the input/output behaviour. For example, suppose that we instrument function-call (`CALL`) and -return instructions (`RET`) in order to build a dynamic call graph. For this to work, the compiler must forego inlining and tail-call elimination, as both of these optimizations remove `CALL`s and `RET`s. The reasoning applies to many other transformations: e.g. for instrumentation code to be able to access the arguments of a function, a frame pointer must often be maintained. Thus, in practice, it is often the case that when a program is compiled for instrumentation, all optimizations are disabled.

1.1.3 Dynamic software updating

A *dynamic update* is an update to a program's code, variable and type definitions that is applied at runtime. Microsoft's hotpatching [85] is probably the most widely-used implementation of dynamic software updating (DSU). Although the mechanics of altering the image of a running program are straightforward, at least on a von Neumann machine, what constitutes a valid update and when it may be applied is up for debate. At the same time, anecdotal evidence suggests that dynamic updates coexist poorly with program optimization. This is hardly surprising—a dynamic update makes implicit assumptions about the intermediate state of the program at the time it is applied, which in turn limit the compiler's scope for optimization. The greater the dependence of an update on the original program, the more disruptive it is.

1.2 Reducing interpretational overhead

A VMM, a DBI framework and a DSU system must perform their respective functions without imposing undue overhead. In particular, each must arrange to regain control at certain points of interest during a program run and neither can afford to resort to naive, line-by-line interpretation of the program. The alternatives are (*i*) optimizing compilation or binary translation, and (*ii*) cooperation from the hardware. Modern hardware has native architectural support for virtualization, as well as hardware breakpoints and performance counters to aid instrumentation and debugging. A virtual machine monitor can direct the CPU to intercept required instructions and pass control to the VMM (this processor feature is known as "trap-and-emulate execution"). On architectures without trap-and-emulate, a VMM has to perform binary analysis and rewriting in order to locate privileged instructions and redirect execution to the emulation code. Thus, a binary translation VMM implements a full program transformation toolchain for the platform's machine code including a disassembler which is already a tricky proposition on the x86 due to the possibility of self-modifying code. Unsurprisingly, many more x86 VMMs have emerged since both Intel and AMD implemented trap-and-emulate virtualization extensions, even though hardware assists do not necessarily result in better performance over a well-written binary translation VMM [5]. Binary instrumentation frameworks often use both binary rewriting and modest hardware support: breakpoint instructions are placed at relevant sites throughout the program and a custom breakpoint handler is installed.

Indeed, binary translation and hardware assists are means to a common end: reduction of interpretational overhead. We will show in Chapters 3 to 4 that definitions (and intuitions) for interpretational overhead that were developed independently in the partial evaluation, virtualization and instrumentation literature are closely related. In the spirit of parsimony, we also present a post-hoc rationalization for the design of hardware virtualization assists. This work is conceptually related to the efforts of Danvy [35], Ager et al. [7] and others to explain the structure of abstract machines for functional languages starting with semantics of the languages themselves. Our eventual aim (outside the scope of this dissertation) is to *derive* hardware extensions starting from optimizing program transformations for a simple base assembly language.

Dynamic software updating and dynamic instrumentation systems conflict with static program optimization because optimizing compilers are written with the standard interpretation of the language in mind. In many implementations, optimizations are either disabled outright (e.g. `gcc -O0`) or incidentally (e.g. taking the address of a function in C will most likely prevent inlining of the function). This is not so much a problem for VMMs, because (*i*) the non-standard interpretation is not any more invasive than the standard one (i.e the emulation code for `PRINT 42` is unlikely to examine the contents of registers), and (*ii*) programs in low-level languages are not optimized as aggressively, so we never expect a compiler to optimize away a privileged instruction.

1.3 The illusion of interpreted execution

Folk wisdom has it that after a good optimizing compiler is done with a program it bears little resemblance to the original code, and this is particularly true of compilers that perform whole-program optimization. However, for some applications it is necessary to maintain an illusion of naive, interpreted execution for optimized programs (much as an internally-parallel CPU might pretend to execute instructions sequentially). In general, we can distinguish between observations on program state that the language itself is capable of expressing and those from "without" (e.g. made with the help of a debugger), but close scrutiny of either kind provably hinders optimization (see Chapter 2). For example, tail-call optimization in languages with security by stack inspection [143] is awkward because the call stack needs to be occasionally examined at runtime. But the problem is most conspicuous with debugging because a debugger grants unconditional access to intermediate execution states: i.e. a debugger only supplies a *mechanism* for interacting with the running program and it is then up to the user to determine what observations and modifications to make. Crucially, the user's expectations about the values of variables, the contents of the call stack, etc. are borne out of a view of program execution as a sequential "line-by-line" activity. The compiler on the other hand is constrained only by the "end-to-end" behaviour of programs and is not obliged to preserve internal detail. The antagonism between source-level debugging and program optimization is well-known. There is a single concept behind references in the literature to "interrupt points" [66], "[on-stack-replacement] points" [47], "points of interest" [132] and "update points" [126]. These are points in execution where the state of the optimized program must be made to match what it would have been had the program not been optimized. As an example, consider that in C the use of the `volatile` qualifier is a deliberate barrier to optimization, since the compiler has to assume that the value of a `volatile`-qualified variable can be observed and modified at any time. Draft WG14/N1124 reads (p. 109):

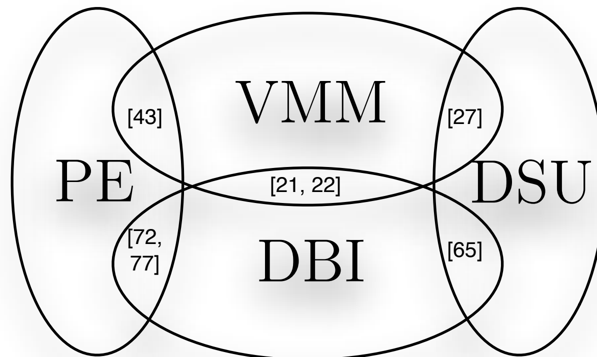
> "An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects. Therefore any expression referring to such an object shall be evaluated strictly according to the rules of the abstract machine, as described in 5.1.2.3. Furthermore, at every sequence point the value last stored in the object shall agree with that prescribed by the abstract machine, except as modified by the unknown factors mentioned previously."

As a testament to the difficulty of implementing an "interpreted" feature as part of an optimizing compiler, Eide and Regehr [39] show that many compilers miscompile `volatile`.

Thankfully, the requirements of a DBI framework or DSU system are much more limited than those of a fully-fledged debugger: the unoptimized state of the program only needs to be reconstructed rarely and then only partially. Further, unlike debugging, instrumentation code and dynamic updates can be restricted by fiat when this leads to profitable optimization. In Chapters 4 and 5 we show how such linguistic constraints enable static optimization. Dynamic deoptimization [66], a technique pioneered in the Self programming language for recovering the interpreter state from the state of an optimized program, can be used to lift restrictions in some cases.

1.4 Related work

Our contributions are set in context of prior work on the intersection of virtualization, DBI, DSU and partial evaluation (PE). However, the interplay between these topics is often overlooked—our goal is to make it explicit here and throughout the rest of the dissertation. The sampling of literature in the figure below circumscribes the space of problems this dissertation is concerned with.



Bungale and Luk [21] incorporate a DBI framework into Xen (with VT-x support), making it possible to instrument an unmodified guest operating system[4]. Chen et al. [27] describe a virtualization-based mechanism for dynamically updating a guest operating system. Hicks et al. [65] developed a general-purpose DSU framework for C programs which, as a special case, can be used to add instrumentation to a running program. Instrumentation can have a negative impact on the performance of the instrumented program: Cantrill et al.'s [22] informal statement of efficiency for DTrace (quoted in §1.1.2) closely resembles Popek and Goldberg's efficiency requirement for VMMs [102]. We previously showed [43] that Popek and Goldberg's criterion is itself a close relative of Jones optimality, a measure

[4] VMware Workstation, a popular VMM, also supports debugging of guest operating systems.

of the strength of program specializers (see Chapter 3). Program specialization is one way of removing interpretational overhead: Jones [72] describes practical considerations of writing interpreters that are *intended* for specialization rather than execution. In Kishon and Hudak's "monitoring semantics" [77], a standard interpreter is specialized with respect to a monitor specification and the resulting instrumented (i.e. non-standard) interpreter is specialized with respect to the program to obtain an instrumented program.

1.5 Dissertation contributions & outline

Chapter 2 contains a recap of various items of necessary technical background. After that, the rest of the dissertation addresses the following five interesting problems that, to the best of our knowledge, are not satisfactorily resolved in the literature.

Interpretational overhead in virtualization and partial evaluation

The virtualization and partial evaluation communities use different, independently formulated definitions of interpretational overhead. Chapter 3 relates the two approaches of limiting interpretational overhead in virtualization: specialization and hardware trap-and-emulate assists. We propose a natural, but apparently novel view of a VMM as a self-interpreter for machine code and reconcile Popek and Goldberg's efficiency criterion for VMMs with Jones optimality [71] using a new version of Jones optimality, which we call "Jones optimality for traces". (This part of the chapter is based on our PEPM 2008 paper [43].) Each virtualization mode has its own advantages and disadvantages. In principle, parts of a guest program may be naively interpreted by the VMM, others may run with trap-and-emulate support enabled and yet others may be executed natively having had privileged instructions replaced with calls into the VMM. We show how trap-and-emulate execution can be rationally explained starting from context threading, a technique for implementing efficient interpreters due to Berndl et al. [18].

Formally efficient program instrumentation

Although modern DBI frameworks like DTrace and VProbes (VMware) offer practical ways of addressing performance concerns, there has been no formal understanding of what it means for instrumentation to be efficient. Similarly, no distinction is drawn between lightweight DBI (DTrace etc.) and more invasive, heavyweight instrumentation frameworks such as Valgrind [99]. In Chapter 4, we adapt Jones optimality for traces to fill this role. The instrumentation code is written in a domain-specific language; we use partial equivalence relations to describe the "observational power" of instrumentation code. (Appendix B contains a more thorough introduction to partial equivalence relations with a survey of some of their applications.) Restricting possible future instrumentation enables more aggressive static optimization of dynamically instrumented programs.

Multi-language semantics for low-level code

A virtualized program, an instrumented program and an updated program each contain a mixture of code units which are interpreted differently. For a VMM, an example would be mixing native, trap-and-emulate and fully interpreted execution. In DBI, different parts of the program are instrumented in different ways under different sets of instrumentation. In DSU, the "old" and "new" code that is part of an updated program can be seen as

written in different languages. As part of their work on type safety for dynamically-updatable programs, Stoyle et al. proposed a novel syntactic check on the runtime state of a program at the point of update: the *con-free check* verifies that old code will never use values of updated types concretely, i.e. in a way that may cause a type error. The associated property—which those runtime states that pass the check satisfy—is called *con-freeness*. The key contribution of Chapter 5 is a reconstruction of con-freeness starting from a multi-language ("old code"/"new code") semantics for updatable programs.

Optimization of dynamically-updatable programs

Dynamic instrumentation is a special case of DSU in which the updates are derived by specialization of an instrumented interpreter with respect to the original code. Optimization of dynamically-updatable programs is considered problematic. Chapter 5 contains a proposal for optimizing dynamically-updatable programs by restricting the content of future updates. We note that tagging and untagging operations performed by dynamically-updatable programs are analogous to those in residual programs that result from specialization of interpreters written in strongly-typed languages.

Information flow in (de)compilation

Finally, Chapter 6 is a tentative presentation of decompilation as a form of attack on the program in the sense of information-flow security. We draw parallels between the normalizing effect of optimizing compilation and the notion of non-interference, and give a definition of secure information flow for program transformations. (This view naturally accommodates superoptimization and randomized compilation.) It is immediately apparent that the impossibility of perfectly optimizing compilation precludes construction of compilers with zero information flow. In an idealized setting of a compiler from an simple high-level language to a stack machine code, we formalise the idea of debug tables as boundaries between compiled and interpreted code. In conclusion, we speculate on possible applications to software protection measures such as obfuscation and watermarking. Early ideas relating to this chapter were presented at the Programming Language Interference and Dependence (PLID) Workshop in 2008 [42].

Chapter 7 concludes and also puts forward several recommendations for future hardware systems. We recap the main recurring themes of the dissertation and describe a "language-centric" approach to low-level system software: in particular, we argue for architectural support for non-standard interpretation that is directly motivated by program optimization.

Chapter 2

Technical background

The aim of this chapter is to introduce essential concepts used throughout the rest of the dissertation. The chapter presents a diverse set of topics, but the material is intended to be accessible to readers from both typed functional programming and systems backgrounds.

Outline. We begin by recalling in §2.1 that programs are data; we briefly cover program staging (quasi-quoting) and the phase distinction. A review of the main definitions of partial evaluation follows in §2.2. In §2.3 we introduce a CISC-style assembly language ("AL") for a Harvard architecture machine. We discuss computational reflection and self-modifying code (§2.4), their negative impact on program optimization and several theoretical and practical workarounds that are used to counter this. There are many ways of writing an interpreter for AL; we include examples in C and OCaml (§2.5). Next, §2.6 is a more thorough guide to non-standard interpretation in AL focusing on program instrumentation as the most obvious instance, but also covering virtualization (emulation, trap-and-emulate and binary translation) and dynamic software updating. Virtualization and DSU are dealt with in more detail in Chapters 3 and 5 respectively. The relevance of multi-language interoperability becomes apparent when we consider applying different sets of instrumentation to different parts of the program simultaneously. In Chapters 4 and 5 we will use partial equivalence relations (PERs), a basic tool used to give semantics to types and static analyses: §2.7 introduces PERs (some example applications of PERs are given in Appendix B).

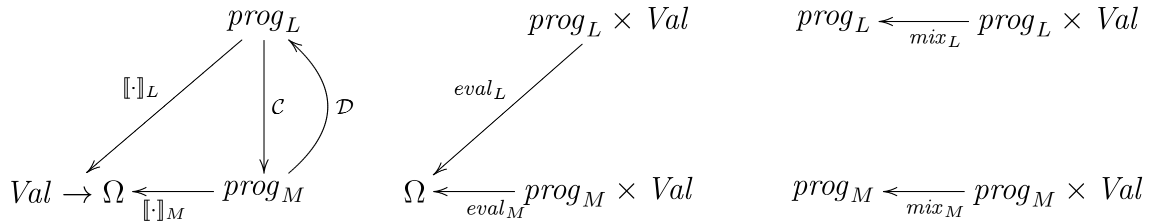
2.1 Programs as data

The word "program" is usually used to refer to a syntactic object such as a string generated by a context-free grammar. A *programming language semantics* gives meaning to programs in terms of some existing well-understood structure or formalism. Various styles of semantic definition have been proposed over the years: for example, a *Scott-Strachey denotational semantics* is a mapping from programs to continuous functions over complete partial orders (CPOs), whereas an *operational semantics* defines an evaluation relation between consecutive program execution states called "configurations". The language used to define the semantics is called the *meta-language*; the language that programs are written in is called the *object-language*. We identify a programming language with its semantics rather than its syntax: so two languages with the same syntax but differing semantics are considered distinct.

One important difference between denotational and operational approaches is that the latter treats non-interactive program *inputs* as part of the program itself by making them part of the starting configuration (the very first execution state). Let *Val* be a set of first-order[1] input values common to all languages (Lisp-style S-expressions, for example) and let $\Omega \stackrel{\text{def}}{=} Val_\perp$ be a CPO of results. A denotational semantics is usually defined as a function $[\![\cdot]\!] : prog \to (Val \to \Omega)$ from programs to denotations. An operational semantics, on the other hand, is a reduction relation between configurations: $\langle p, s\rangle \rightsquigarrow \langle p', s'\rangle$ where $p, p' \in prog$ and $s, s' \in Val$ contain execution state (which in this case is a single value). A *terminal configuration* is a configuration that cannot be reduced any further: we will take terminal configurations to be values. The definition of $[\![\cdot]\!]$ in terms of the reduction relation is a straightforward application of Kleene's s-m-n theorem (cf. currying):

$$[\![p]\!]\ v = \begin{cases} \perp & \langle p, v\rangle \text{ diverges} \\ v' & \langle p, v\rangle \rightsquigarrow \ldots \rightsquigarrow v' \ . \end{cases}$$

If we take the meta-language to be a conventional programming language (C, ML, etc.), it becomes apparent that the operational style of definition produces *interpreters* while the denotational style closely resembles *compilation*. Indeed, some of the same techniques (e.g. using monads [90, 139, 63]) can be used to write denotational semantics, interpreters and compilers. The diagrams below summarize the relationships between a compiler $\mathcal{C}$ from language L to language M; $\mathcal{D}$, a decompiler; $[\![\cdot]\!]_L$ and $[\![\cdot]\!]_M$, the semantic functions; $eval_L$ and $eval_M$, the interpreters; and mix_L and mix_M, the *program specializers*.



We assume that it is possible to define an injection from multiple values to a single value (e.g. a list or tuple constructor), so that programs can take arbitrarily many inputs. A further auxiliary function $\lfloor\cdot\rfloor : prog \to Val$ is used to map a program to a data value (an encoding of the program's abstract syntax tree), so that programs can be given as input to other programs; $\lfloor\cdot\rfloor$ must be injective. The function *mix*, the object of study in the field of partial evaluation, is discussed fully in §2.2 below, but its basic purpose is to exploit advanced knowledge about one or more of the inputs to optimize the program.

2.1.1 Full abstraction

Recall that two expressions are said to be *contextually equivalent* ($e_1 \sim e_2$) if both exhibit identical behaviour in all contexts Ctx such that $\mathsf{Ctx}[e_1]$ and $\mathsf{Ctx}[e_2]$ are closed terms:

$$e_1 \sim e_2 \iff \forall \mathsf{Ctx}[-].\ \mathsf{Ctx}[e_1] \approx \mathsf{Ctx}[e_2] \qquad \text{where } \approx \text{ is e.g. convergence .} \qquad (2.1)$$

The definition takes into account the *internal observational power* of the language: the ability of language contexts to distinguish different terms. The relation $\approx$ is usually defined using an operational semantics (e.g. let $e_1 \approx e_2$ iff e_1, e_2 either both diverge or

[1]The first-order restriction applies to program inputs only, not the values that programs manipulate.

both converge to the same value). A denotational semantics $[\![\cdot]\!]$ is *fully abstract* with respect to an operational semantics if operationally-equivalent terms are assigned the same denotation and, vice versa, terms that have the same denotation are operationally-equivalent: $[\![e_1]\!] = [\![e_2]\!] \iff e_1 \sim e_2$. Abadi [1] suggested that a safe language *translation* must also be fully abstract (i.e. preserve and reflect term equality) in the following sense: given a compiler $\mathcal{C}$ from L to M and relations $\sim_L$ and $\sim_M$ (contextual equivalence relations over L and M respectively), it should be the case that

$$e_1 \sim_L e_2 \iff \mathcal{C}(e_1) \sim_M \mathcal{C}(e_2)\ . \tag{2.2}$$

Intuitively, the definition says that translation does not permit an adversarial M-context to glean more information than any L-context might. In particular, there is a danger that a low-level target language might distinguish phrases equivalent in the source language: Abadi points out several cases where the then-current Java to bytecode compiler fails to be fully abstract; Kennedy [76] identifies similar issues in the C# to CIL compiler. In practical terms, failure of full abstraction means that the programmer cannot rely on source-level reasoning in C# and must be aware of the intricacies of the bytecode language and the compiler. Note that given some expression e and the output of the compiler $\mathcal{C}(e)$, it is not possible to tell whether $\mathcal{C}$ is fully abstract (indeed, the question itself is meaningless). Full abstraction is a relational property, not an attribute of any particular compilation run.

2.1.2 Dependent types and the phase distinction

Dependent types (or *types indexed by terms*) can be used to capture the type system of an *embedded language* in the type system of the host language. For instance, the C library function `printf` interprets its arguments according to a *format string* (e.g. `"%s%d"`) written in a small domain-specific language. Intuitively, the *value* of the format string determines the *type* of `printf`. In the dependently-typed language Cayenne [12], `printf` can be assigned a meaningful type. A similar effect can be achieved with static analysis in languages that do not support dependent types: for example, Christensen et al. [29] approximate the runtime values of string variables in Java by regular languages allowing the syntax of dynamically-generated SQL queries to be checked at compile-time.

Cardelli [23] coined the term "phase distinction" to refer to the traditional separation of that which happens at compile-time from that which happens at runtime. Cardelli notes that although type-checking is usually done at compile-time (i.e. during the static phase), strict separation fails for languages with dependent types. Because type equality in dependently-typed languages is parameterized over term equality, the type-checker will generally need to evaluate terms. This poses an immediate problem for Turing-complete languages: the implementer can either (*i*) restrict the set of terms that are allowed to index types to a decidable subset [149], (*ii*) accept the possibility of non-termination in the type-checker, or (*iii*) delay type-checking in part until runtime. Ou et al. [101] combine simple and dependent types in a single language, and check at runtime that values flowing into dependently-typed code have the correct types. Flanagan [48] insert runtime checks where a theorem prover is not able to establish type-safety statically: their calculus, λ^H is the simply-typed λ-calculus enhanced with *refinement types*, i.e. types of the form $\{x : \tau \mid e\}$ where e is a term. The key point is that it is occasionally either necessary or beneficial to delay traditionally compile-time activities until runtime. One of these activities is *synthesis of new program code*.

2.1.3 Program staging

Many useful optimizations require the ability to generate code at runtime. In extreme cases, we can separate execution into two distinct stages: during the first stage, the program manipulates fragments of syntax to produce its output, which is another program. During the second stage, this newly-generated program is executed. For example, in C, the pre-processor makes possible rudimentary program staging—although the language of pre-processor directives is meager compared to C itself. C++ templates and Lisp macros are much more interesting and powerful facilities, but unfortunately outside the scope of this chapter. Fine-grained program staging annotations, like those found in Lisp or MetaML [128], allow the programmer to (*i*) delay the evaluation of a term (i.e. quote), (*ii*) splice a delayed term into another delayed term, (*iii*) splice the result of evaluating a term into a delayed term (i.e. quasi-quote), and (*iv*) evaluate a delayed term (`run` or `eval`). The effect of the first three primitives, apart from `run`, can be achieved by ad-hoc string or S-expression manipulation in languages without support for staging; implementing `run` as an interpreter function is not practicable for performance reasons.

2.2 Partial evaluation

Staging is a useful technique which can bring substantial performance benefits, but manual staging does require forethought. A *partial evaluator*, or "program specializer", uses known program inputs to *automatically* optimize the program. Inputs whose values are known at partial evaluation time are called *static*; the other inputs are *dynamic*. The output of *mix* is called the *residual* (or *specialized*) program. Let $[\![\cdot]\!]$ be an evaluation function, p a program with two inputs[2] and *mix* a partial evaluator:

$$[\![p]\!](v_1, v_2) = [\![[\![mix]\!](p, v_1)]\!](v_2) \ . \tag{2.3}$$

Aggressive constant propagation, function inlining and loop unrolling are the hallmarks of program specialization. But it is easy to define a valid specializer that does no useful work and merely splices its inputs p and v_1 into a set program template. For example, in Emacs Lisp:

```
(defun mix (p v1)
  `(lambda (v2)
     (funcall ,p ,v1 v2)))
```

The Futamura projections. There are a few startling consequences of the equation for *mix* above. Let int_M^L be an interpreter for language M written in L, i.e.:

$$\forall p \in prog_M.\ \forall v \in Val.\ [\![int_M^L]\!]_L(p, v) = [\![p]\!]_M(v) \ . \tag{2.4}$$

Then specializing int_M^L with respect to an M-program translates the program into L:

$$\forall p \in prog_M.\ [\![mix]\!](int_M^L, p) \in prog_L \ . \tag{2.5}$$

This is known as the first Futamura projection. Many program transformations can be derived by specializing specially-crafted interpreters [72, 37]: for example, specializing an

[2] Partial evaluation can be used for single-input programs when something is known about the possible values of the input (odd, even, positive, negative, etc.).

instrumented interpreter with respect to a program "inlines" the instrumentation code (see §2.6). Specializing the specializer itself with respect to an interpreter produces a compiler:

$$\mathcal{C}^L_M = [\![mix]\!](mix,\ int^L_M)\ . \tag{2.6}$$

This is the second Futamura projection and *mix* must be a *self-applicable* partial evaluator. Finally, specializing the specializer with respect to itself produces a compiler generator:

$$cogen = [\![mix]\!](mix,\ mix)\ . \tag{2.7}$$

This is the third Futamura projection, which is included here for completeness: we will refer only to the first two projections in the rest of the dissertation.

2.2.1 Jones optimality

Jones optimality is a measure of the strength of program specializers due to Neil Jones [71]. Let *sint* range over self-interpreters (a self-interpreter is an interpreter written in the same language that it interprets), then a *mix* is Jones-optimal ("strong enough" in Jones' original nomenclature) iff

$$\exists sint.\ \forall p.\quad [\![mix]\!](sint,\ p) =_\alpha p \tag{2.8}$$

where $=_\alpha$ is α-equivalence or a similar decidable syntactic equality relation. Intuitively, this means that *mix* is capable of removing a *layer of interpretational overhead*. A side-condition is usually added to the effect that *mix* may not "cheat" by comparing the value of *sint* against a known self-interpreter and returning p on success:

$$mix_{\text{cheat}}(sint,\ p) \stackrel{\text{def}}{=} \textit{if}\ sint = \texttt{MAGIC_SINT}\ \textit{then}\ p\ \textit{else}\ [\![mix_{\text{triv}}]\!](sint,\ p)\ . \tag{2.9}$$

The cheating specializer is Jones-optimal for a single self-interpreter whose text is hard-coded (`MAGIC_SINT`) into the definition of the specializer; mix_{cheat} resorts to trivial specialization for all other values of *sint*. To paraphrase Orwell, "some self-interpreters are more equal than others". An expectation of fairness (cf. continuity) applies to many program transformations and static analyses: a minor change in the input should produce a minor change in the output. But whether a given change is minor or major is arguable because there are no accepted definitions of distance for program text.

We generalise the definition of Jones optimality in the manner of Glück [56]. Let R be a binary relation on programs capturing some equivalence or ordering, and define $Jopt_R$ as follows:

$$\begin{array}{lcl} Jopt_R(mix,\ sint) & \stackrel{\text{def}}{=} & \forall p.\ [\![mix]\!](sint,\ p)\ R\ p \\ Jopt_R(mix) & \stackrel{\text{def}}{=} & \exists sint.\ \ Jopt_R(mix,\ sint)\ . \end{array} \tag{2.10}$$

We say that *mix* is "R-strong with respect to *sint*" if $Jopt_R(mix,\ sint)$ holds, and that *mix* is simply "R-strong" if $Jopt_R(mix)$ holds. Equation 2.8 is equivalent to $Jopt_{=_\alpha}(mix)$.

The original definition of Jones optimality disallows potential optimizations that *mix* might be able to perform, since, e.g. `1+2` $\neq_\alpha$ `3`. For this reason, a subsequent version of Jones optimality [73, Definition 6.4] uses a weaker relation defined over the running times of the computations:

$$p' \leq_{\text{time}} p \stackrel{\text{def}}{=} \forall v.\ \text{time}_{p'}(v) \leq time_p(v) \tag{2.11}$$

$$
\begin{array}{rcll}
\textit{insn} & ::= & \texttt{MOV}\ r_{\text{dst}},\ a_{\text{src}} \mid \texttt{LOAD}\ r_{\text{dst}},\ \texttt{(}\ a_{\text{src}}\ \texttt{)} \mid \texttt{STORE}\ \texttt{(}\ a_{\text{dst}}\ \texttt{)},\ a_{\text{src}} & a ::= v \mid r \\
& \mid & \texttt{NOP} \mid \texttt{HLT} \mid \texttt{ALU}(\textit{op})\ r_{\text{dst}},\ a_{\text{src1}},\ a_{\text{src2}} \mid \texttt{CALL}\ a_{\text{loc}} \mid \texttt{RET} \mid \texttt{JMP}\ a_{\text{loc}} \mid \texttt{JE}\ a_{\text{loc}},\ a_1,\ a_2 & \\
& \mid & \texttt{JZ}\ a_{\text{loc}},\ a \mid \texttt{OUT}\ \texttt{<}\, a_{\text{port}}\, \texttt{>},\ a_{\text{src}} \mid \texttt{IN}\ r_{\text{dst}},\ \texttt{<}\, a_{\text{port}}\, \texttt{>} \mid \texttt{PUSH}\ a \mid \texttt{POP}\ r_{\text{dst}} \mid \texttt{UPDATE} &
\end{array}
$$

Figure 2.1: Syntax of AL.

where $time_p(v)$ is the execution time of running p on input v. Under a very reasonable assumption that α-equivalent programs consume equal execution time, this definition subsumes the original. The crucial change is that an intensional, static notion of equivalence used in Equation 2.8 is replaced with an extensional, dynamic one.

2.3 AL

In Chapters 3 and 4 we will consider virtualization and instrumentation of programs written in an assembly language called AL (Figure 2.1). The underlying CISC machine has a Harvard architecture: instructions and data are stored separately. The code store cannot be read or written to and the data store cannot be executed. Similar restrictions versus a von Neumann machine are often imposed on x86 code to allow reliable disassembly: for example, in Google's Native Client sandbox [150]. We will come back to the problem of self-modifying code in the next section. The AL machine is finite, but we do not stipulate its size, i.e. the amount of code and data memory, word size and number of registers. For given values of these parameters, the grammar of AL can be expanded completely to produce the set of all valid instructions. As in Intel x86 syntax, the destination operand comes before the source operand(s). There are two special-purpose registers: `L` (for "link") contains the return address on function call, and `S` holds the stack pointer. The instruction set is mostly standard; various arithmetic and relational operators are ranged over by *op*; `OUT` prints the contents of a register or an immediate value; `IN` reads an externally-supplied value; `HLT` terminates the program. The `IN` and `OUT` instructions are the only means of interacting with a running program: in particular, the internal machine state is *discarded on termination.* In order to preserve the state, the program must terminate by executing the special `UPDATE` instruction described in §2.3.3.

The assembler function $\langle\!|\cdot|\!\rangle$ maps AL surface syntax to AL values. A single instruction maps to four words: an opcode and up to three operands. For example, $\langle\!|\texttt{ADD r0, r1, 42}|\!\rangle = \begin{array}{|c|c|c|c|}\hline \textit{opcode}(\texttt{ADD}, \texttt{rri}) & 0 & 1 & 42 \\ \hline \end{array}$. The second argument to *opcode* selects the opcode variant according to the types of operands: here the first two are registers (`r`), and the last one is an immediate value (`i`). Every label is replaced with the address of the labelled instruction. The function $\langle\!|\cdot|\!\rangle$ is bijective modulo label names; the inverse $\langle\!|\cdot|\!\rangle^{-1}$ is the disassembler.

2.3.1 Semantics

The machine state is a tuple containing the program counter, the register file, and the data and code memories. We let $Loc \subseteq Val$ be the set of allowable memory locations, and $State \stackrel{\text{def}}{=} (Loc,\ Reg \to Val,\ Loc \rightharpoonup Val,\ Loc \rightharpoonup Val)$. The data memory and registers are zeroed at program start. Reading or writing to an out-of-range data memory address aborts execution; so does branching to a misaligned or out-of-range code memory address

If the instruction is:	then the machine configuration becomes:
MOV r_{dst}, a_{src}	$(pc + 4,\ R[r_{\text{dst}} \mapsto V(a_{\text{src}})],\ D,\ C)$
LOAD r_{dst}, (a_{src})	$(pc + 4,\ R[r_{\text{dst}} \mapsto D(V(a_{\text{src}}))],\ D,\ C)$
STORE (a_{dst}), a_{src}	$(pc + 4,\ R,\ D[V(a_{\text{dst}}) \mapsto V(a_{\text{src}})],\ C)$
ALU(op) r_{dst}, a_{src1}, a_{src2}	$(pc + 4,\ R[r_{\text{dst}} \mapsto op(V(a_{\text{src1}}),\ V(a_{\text{src2}}))],\ D,\ C)$
CALL a_{loc}	$(V(a_{\text{loc}}),\ R[\mathtt{L} \mapsto pc + 4],\ D,\ C)$
RET	$(R(\mathtt{L}),\ R,\ D,\ C)$
JMP a_{loc}	$(V(a_{\text{loc}}),\ R,\ D,\ C)$
JE a_{loc}, a_1, a_2	$(pc',\ R,\ D,\ C)$ where $pc' = (V(a_2) = V(a_2)) \Longrightarrow V(a_{\text{loc}}) \mid pc + 4$
JZ a_{loc}, a	$(pc',\ R,\ D,\ C)$ where $pc' = (V(a) = 0) \Longrightarrow V(a_{\text{loc}}) \mid pc + 4$
OUT $<a_{\text{port}}>$, a_{src}	Print $V(a_{\text{src}})$ to port $V(a_{\text{port}})$; $(pc + 4,\ R,\ D,\ C)$
IN r_{dst}, $<a_{\text{port}}>$	Read value v from port $V(a_{\text{port}})$; $(pc + 4,\ R[r_{\text{dst}} \mapsto v],\ D,\ C)$
PUSH a_{src}	$(pc + 4,\ R[\mathtt{S} \mapsto R(\mathtt{S}) + 1],\ D[R(\mathtt{S}) \mapsto V(a_{\text{src}})],\ C)$
POP r_{dst}	$(pc + 4,\ R[r_{\text{dst}} \mapsto D(s'),\ \mathtt{S} \mapsto s'],\ D,\ C)$ where $s' = R(\mathtt{S}) - 1$
UPDATE	$upd\ (pc,\ R,\ D,\ C)$
HLT	ok

A convenient shorthand: $V(v) \equiv v$ and $V(r) \equiv R(r)$.

Figure 2.2: Semantics of AL with range checks omitted.

or attempting to execute an invalid instruction. A program that "falls off the end" terminates normally: i.e. there is an implicit HLT instruction following the last instruction in the program.

The small-step semantics of AL is shown in Figure 2.2: every machine instructions takes exactly one cycle (reduction) to complete. The current machine state $(pc,\ R,\ D,\ C)$ is in scope in the right-hand column of the table. The reduction relation $\rightarrow$ ("reduces to") is defined between *machine configurations*: a non-terminal configuration is a machine state; the contents of a terminal configuration (the *answer*) depend on how the machine was stopped: the values *ok* and *err* are returned on normal (HLT) and abnormal termination respectively. The UPDATE instruction terminates the machine exposing the intermediate execution state to an external observer.

$$Answer \stackrel{\text{def}}{=} \{ok,\ err\} \cup \{upd(s) \mid s \in State\}\ .$$

2.3.2 Traces

A *state trace* is a possibly-infinite sequence of program states: $(s_1 s_2 \ldots)$ such that $\forall i.\ s_i \rightarrow s_{i+1}$; or $(s_1 s_2 \ldots s_n a)$ such that $\forall i < n.\ s_i \rightarrow s_{i+1}$ and $s_n \rightarrow a$ where $a \in Answer$. Similarly, an *operation trace* is the sequence of instructions executed together with their data-flow inputs and outputs. Since our aim is to identify unnecessary operations (i.e. overhead), we will use operation traces in preference to the more traditional state traces, and so take "trace" to mean "operation trace" unless otherwise stated; we let $\mathcal{T}[\![p]\!]$ denote the trace of p. Outputs are enclosed in square brackets and the letters following the values (r or i) identify the opcode variant used. We further define an *erasure map* $E(\cdot)$ for traces which discards the operand encoding information, and replaces register names, port numbers and memory addresses (including target addresses of jumps and calls) with a wild card

symbol. A program to calculate the factorial of 5 and excerpts of its original and erased traces are shown below.

Program	Trace	Erased trace
fac5: MOV r1, 5	MOV [5]/r1, 5/i	MOV [5], 5
MOV r2, 1	MOV [1]/r2, 1/i	MOV [1], 1
loop: JZ done, r1	JZ 24/i, 5/r1	JZ *, 5
MUL r2, r1, r2	MUL [5]/r2, 5/r1, 1/r2	MUL [5], 5, 1
SUB r1, r1, 1	SUB [4]/r1, 5/r1, 1/i	SUB [4], 5, 1
JMP loop	JMP 8/i	JMP *
done: OUT <1>, r2	JZ 24/i, 4/i	JZ *, 4
	...	...
	OUT <1/i>, 120/r2	OUT <*>, 120

Nethercote and Mycroft [98] argued that a dynamic data dependence graph built from a program trace represents the "essence" of a computation. The erased trace is an attempt to capture the same intuition with a more lightweight formalism.

2.3.3 Update points

Every occurrence of UPDATE is an *update point* (we borrow the phrase from dynamic software updating). Intuitively, the instruction suspends execution with the possibility of later restart: UPDATE can be thought of as a transfer of control to the operating system, a debugger breakpoint or a checkpoint [147]. The same basic idea is also familiar from denotational semantics of I/O using *resumptions*.

Consider a mini-language of expressions with syntax $exp ::= \texttt{read} \mid exp\ op\ exp \mid \underline{n} \in \mathbb{N}$. The two possible outcomes of running a program in this language are termination with either (*i*) a value or (*ii*) a *resumption*, i.e. a value-consuming continuation. The pre-CPO (CPO with no bottom element) of results can be defined as $\Omega \cong (\mathbb{N} + (\mathbb{N} \to \Omega))$ and the semantics is straightforward:

$$\begin{aligned} [\![\cdot]\!] &: (\mathbb{N} \to \Omega) \to \Omega \\ [\![\underline{n}]\!]\ k &= k\ n \\ [\![e_1\ op\ e_2]\!]\ k &= [\![e_1]\!]\ (\lambda v. [\![e_2]\!]\ (\lambda v'.\ k\ (v\ op\ v'))) \\ [\![\texttt{read}]\!]\ k &= \lambda v.\ k\ v\ . \end{aligned}$$

Taking the initial continuation to be the identity function, we have that $[\![\texttt{read+42}]\!]\ (\lambda v.v)$ is equivalent to $\lambda v.\ (v + 42)$. Operationally speaking, the expression read+42 terminates with a continuation pending further input: the continuation must be applied to an input value to restart the program. Notice that the means by which the value is obtained prior to being supplied to the resumption is completely orthogonal to the language semantics. By the same token, the virtue of UPDATE is that it provides a *mechanism* for dynamically updating a program, but does not specify policy, i.e. what an allowable update contains, and when and how one can be applied. The choice of Harvard architecture together with UPDATE rather than a von Neumann machine also has important implications for optimization of AL programs, and these are discussed below.

2.4 Computational reflection and self-modifying code

The state of an interpreted program is held in a number of "registers" in the interpreter: e.g. the code register, the data register and the program counter. *Computational reflec-*

tion [123, 50, 145] is a language feature allowing programs to examine and modify the contents of these registers. The values in the registers are made available to the program by *reification*, i.e. encoding into a form that the program can manipulate directly; for example, `call/cc` reifies the program continuation. New values are installed into the registers by *reflection*; `eval`, a staging primitive discussed in §2.1.3, is a form of reflection. Computational reflection is well-known to be antagonistic to program optimization: for instance, most literature on static analysis of Java uses a subset of the language that does not include its already limited computational reflection facilities.

A von Neumann architecture computer does not enforce segregation between code and data and thus readily supports reification of program text and `eval` at the machine code level. This poses two serious problems for program analysis and transformation. First, reification of program code is closely related to Gödelization, which, as a language feature, is provably harmful to optimization; and second, ad-hoc runtime code generation makes finding all reachable instructions statically undecidable. A context $\mathsf{Ctx}[-]$ is a *Gödelizing context* iff for all expressions e such that $\mathsf{Ctx}[e]$ is a closed term, $\mathsf{Ctx}[e]$ evaluates to $\lfloor e \rfloor$, a value that uniquely identifies e (see §2.1). There are no such contexts in the λ-calculus as a straightforward consequence of Church-Rosser theorem [57]. However, in Lisp, $\mathsf{Ctx}[-]$ can be defined simply as `quote` $[-]$. Indeed, some variants of Lisp support `F`-expressions (`FEXPR`s): `FEXPR`s are similar to macros in that they receive their arguments unevaluated (the calling convention is known as "call-by-text"), so the Gödelization is performed implicitly at the call site. Wand [144] showed that in a λ-calculus with `FEXPR`s and `eval`, contextual equivalence coincides with α-equivalence. This does not necessarily mean that no interesting optimizations are possible in this language—optimization relies on determining whether two terms are equivalent *in a particular context*, whereas contextual equivalence is defined over all contexts—but it does indicate that more sophisticated static analysis is likely to be required. Mitchell [88] introduced "abstraction-preserving reductions" as a means of comparing the expressiveness of programming languages and proved that there are no abstraction-preserving reductions to Lisp with `FEXPR`s: for languages with high-level notions of data abstraction, there exist no fully abstract translations to Lisp with `FEXPR`s. This holds for any language with a contextual equivalence that distinguishes terms up to syntactic equality, and, in particular, the machine code language of a stored-program computer. Binary program analysis tools for x86 executables often rule out the possibility of self-modifying code by fiat: e.g. Yee et al. [150] give a list of conditions that input code for their sandbox must meet, in effect recovering a Harvard architecture. Recent x86 processors support a per-page "no-execute" (NX) flag, which allows an operating system to mark parts of memory as non-executable. There are also practical workarounds that detect self-modification at runtime and invalidate optimizations that may have ceased to be sound (the implementation of the x86 architecture by the Transmeta Crusoe processor [78] is an interesting example of the use of this technique).

We chose to make both Gödelization and `eval` *meta-linguistic operations* in AL: `UPDATE` does not reify the machine state but makes it available to an *outside* observer. Similarly, `eval` can only be implemented in AL by self-interpretation because, although an AL program can trivially generate AL code at runtime, there is no way for it to "jump" into the new code. Instead, a program must rely on an external entity—the user or the operating system—to execute generated code on its behalf. To draw an analogy with program staging: program stages in AL are separated by termination.

2.4.1 External observations and optimization

Tolmach and Appel [132] note that "debuggers must expose the internal behavior of the original program, which may be altered by optimization". This is another manifestation of the general principle that *allowing fine-grained observations hinders optimization*, but in this case the observations are made "from without" rather than within (by a language context). For example, tail-call optimization in languages with security by stack inspection [143] is challenging because the call stack must be examined at certain points during execution. One particularly interesting strategy for balancing optimization versus observability is to *recover the unoptimized program state*—the state of the program as it would have been, had the program not been optimized—from the state of the optimized program on demand at certain program points. This is known as *dynamic deoptimization* and was originally proposed and implemented by Hölzle et al. [66] for the SELF programming language, an influential precursor of Java. Although dynamic deoptimization is used in practice (e.g. Sun's HotSpot virtual machine), there is no general, formal account of this technique. Veldhuizen and Lumsdaine [134] introduced the property of "guaranteed optimization" for compilers by defining "deoptimizing" rewrites and proving that a given compiler can undo any sequence of these. The authors' aim was to show that an additional layer of indirection (due to data abstraction in this case) can be completely removed. Dually, we may ask for "guaranteed dynamic deoptimization"—an assurance that the effects of a sequence of optimizations can be undone at runtime.

2.5 Writing interpreters in C and ML

In the functional programming community, the focus has traditionally been on encoding the static semantics of the object-language in the type system of the meta-language. The systems community, on the other hand, is largely concerned with the performance of interpreters. The emphasis in the literature is on minimizing the branch mis-prediction rate—the penalties of a mis-predicted branch on a deeply pipelined CPU are severe [41]. The common goal is to match the static and dynamic semantics of the object-language closely to the meta-language in order to maximally benefit from the meta-language implementation.

2.5.1 Well-typed interpreters

Consider writing an interpreter for a typed language. Object-language values are represented in the meta-language as values of a *universal datatype*. In untyped (also called "uni-typed") languages, every expression is assigned a universal datatype (e.g. Dynamic):

$$\begin{aligned} \texttt{9} &: \mathsf{Dynamic} \\ \texttt{"abc"} &: \mathsf{Dynamic} \\ \texttt{[1, 2, 3]} &: \mathsf{Dynamic}\,. \end{aligned}$$

Every values carries a tag describing its contents: number, string, list, etc. Language primitives, such as addition, string concatenation and consing, that operate on values of a particular type must check the tags of their arguments at runtime. A failed tag check indicates a type error and usually leads to abnormal termination. Under reasonable assumptions, an interpreter written in an untyped language aborts due to a type error iff

the input program is ill-typed (intuitively, this follows from the "progress" result for the type system and operational semantics of the interpreted language).

The runtime tag checks are made explicit (as *coercions*) when the language that the interpreter is written in is typed. Abadi et al. [2] add type Dynamic and the pair dynamic (introduction) and typecase/else (elimination) to the simply-typed λ-calculus (STLC). STLC itself is strongly-normalizing, but the authors remark that STLC+Dynamic can express the call-by-value **Y**-combinator. This is unsurprising semantically: Gunter [61, Chapter 8] describes a similar model for the untyped λ-calculus. We can easily write an interpreter for STLC in STLC+Dynamic, representing STLC values in the program as values of type Dynamic in the interpreter. However, although a well-typed STLC term cannot "go wrong", a well-typed interpreter written in STLC+Dynamic must still consider this possibility in the else-branch of each typecase because the STLC+Dynamic type system is not sophisticated enough to track the STLC type of Dynamic values. Shields et al. [119] show how, in a language with staged computation, type inference can also be staged: quoting a term defers both its evaluation and type inference. When the term is to be evaluated its type is inferred and, if it does not match the expected type, a special "exception" term is evaluated instead. As the authors point out, the net effect of this is similar to a typecase with two branches, one which executes if the deferred term has the expected type, and the other containing the exception term.

In languages with more expressive type systems (e.g. dependent types—see §2.1.2), the meta-language type of an object-language term can be used to encode the object-language type of that term. In Haskell, generalized algebraic data types (GADTs) can often be used to model the type structure of the object-language in this way. For example, `Expr` τ is the meta-language type of expressions whose object-language type is τ. Ramsey [104] and Benton [16] use type-indexed *embedding-projection pairs* to mediate flow of values between an untyped embedded language and a typed interpreting host language. An embedding-projection pair between two CPOs D and E is a pair of continuous functions $\iota : D \to E$ and $\pi : E \to D$ such that $\forall d \in D.\ (\pi \circ \iota)(d) = d$ and $\forall e \in E.\ (\iota \circ \pi)(e) \sqsubseteq e$. A universal type is defined in the host language, e.g.

```
datatype U = Pair of U * U | Fun of U -> U | Int of int .
```

Expressions of the embedded language are interpreted in this universal type. For each type `T` of the host language, two functions are defined: an embedding (`T -> U`), which allows values of the host language to be embedded in the embedded language, and a projection (`U -> T`), which allows values of the embedded language to be projected to the host. Matthews and Findler [84] extend these ideas to a symmetric setting via a combined union language and provide a further refinement by using *contracts*. Contracts [46] can be thought of a kind of coercion. Contracts provide a structured way of defining function pre- and post-conditions together with an algorithm for assigning blame when the checks fail. Findler and Blume [45] show a connection between contracts and Scott's projections[3].

2.5.2 `while/switch` interpreters

The simplest way of implementing an interpreter is to dispatch directly on the abstract syntax tree (AST). For an assembly language this takes the form of a `switch` statement

[3]This takes us full circle: domain isomorphisms (embedding/projection), universal types and coercions, contracts, and back again.

<table>
<tr><th></th><th>C</th><th>OCaml</th></tr>
<tr><td>Simple dispatch</td><td><pre>#define ADDrrr 0
extern word_t pc, R[], D[], C[];
while(1) {
    switch(C[pc++]) {
    case ADDrrr:
        dst   = C[pc++];
        src1   = C[pc++];
        src2   = C[pc++];
        R[dst] = R[src1] + R[src2];
        break;
    }
}</pre></td><td><pre>type word   = int
and loc   = word
and opcode = ADDrrr
and code   = loc -> icell
and icell = O of opcode | W of word
let eval (pc, r, d, c) =
  match (c pc, ..., c (pc+3)) with
  | (O ADDrrr, W dst, W src1, W src2) ->
      let r’ i = if i == dst
                 then r(src1) + r(src2)
                 else r(i)
      in eval (pc+4, r’, d, c)</pre></td></tr>
<tr><td>Direct-threaded code</td><td><pre>doADDrrr:
    dst   = C[pc++];
    src1   = C[pc++];
    src2   = C[pc++];
    R[dst] = R[src1] + R[src2];
    goto *C[pc++];</pre></td><td><pre>type icell = K of kont | W of word
 and kont = state -> answer
let doADDrrr (pc, r, d, c) =
  match (c (pc+1), ..., c (pc+4)) with
  | (W dst, W src1, W src2, K k) ->
      let r’ i = ...
      in k (pc+4, r’, d, c)</pre></td></tr>
</table>

Figure 2.3: Instruction dispatch.

with a `case` for each opcode. The functional language implementation mimics the operational semantics: instead of the outer loop, tail recursion is used, and instead of a `switch`, we rely on pattern matching. The outlines of both kinds of implementations are shown in Figure 2.3. Interpreters written in this way tend to perform poorly [41]. The compiler converts the `switch` statement to an indirect jump, and because the target of the indirect jump depends on the next instruction to be executed, branch-prediction hardware is likely to mis-predict the branch.

2.5.3 Threading

In *direct-threaded* code, each opcode is first replaced by the address of the routine in the interpreter that implements the opcode. The result of this process is a direct threading table (DTT). The interpreter dispatch loop is replaced with indirect branches through the DTT. Because *every* `doXXX` routine contains an indirect branch instruction, prediction has a better chance of succeeding. The threaded dispatch code (`goto *C[pc++]`) is not legal ANSI C, but must rely on either inline assembly or compiler extensions. There is no immediate functional analogue to direct threading. However, despite the name “direct” threading, this implementation technique is close to continuation-passing style: every instruction receives the continuation to call once it is complete. Context threading [18], a technique based on subroutine threading, is arguably the culmination of the line of work on threaded interpretation—see Chapter 4. Note that although threading delivers a tangible speed-up, a threaded interpreter still has to perform most of the same administrative operations as a `while`/`switch` interpreter.

2.6 Non-standard interpretation of AL

We will assume from now on that a non-standard interpretation *augments* the semantics of a language. The standard behaviour of programs is subsumed by the custom behaviour. This implies that, in some sense, a non-standard interpreter does at least as much work as a standard one. Program instrumentation is the best-known example of non-standard interpretation.

2.6.1 Program instrumentation

Recall that "instrumentation" refers to the modification—by addition of instrumentation code—of a program or its runtime environment to make some hidden details of the program's execution visible: e.g. a count of the number of times a particular system call is made. In contrast to a fully-fledged debugger, instrumentation code is passive in the sense that it only observes the program state and does not modify it. Instrumentation code also does not interfere with the program: an instrumented program produces the same final answer as the original in addition to any statistics gathered by the instrumentation code. Since the meaning of a program is usually determined by its input/output behaviour, anything else is an implementation detail, and so an *instrumented semantics* can be thought of as a model of a particular implementation. For example, assume a variable store $\Sigma \stackrel{\text{def}}{=} \mathit{Var} \to \mathit{Val}$ and a semantic function $[\![\cdot]\!] : \mathit{comm} \to \Sigma \to \Sigma_\perp$. In an instrumented semantics that keeps track of the number of times the variable `x` is assigned to, the answer domain is augmented to include the counter. Thus the semantic function is $\mathcal{I}[\![\cdot]\!] \in \mathit{comm} \to \Sigma \to (\Sigma \times \mathbb{N})_\perp$. The definitions of assignment and sequencing are changed accordingly:

$$
\begin{aligned}
\mathcal{I}[\![x := v]\!]\sigma &= \mathit{let}\ \sigma' = \sigma[x \mapsto v]\ \mathit{in}\ (\mathit{if}\ x = \texttt{"x"}\ \mathit{then}\ (\sigma',\, 1)\ \mathit{else}\ (\sigma',\, 0)) \\
\mathcal{I}[\![c_1;\ c_2]\!]\sigma &= \mathit{let}\ (\sigma',\, n_1) = [\![c_1]\!]\sigma\ \mathit{in}\ (\mathit{let}\ (\sigma'',\, n_2) = [\![c_2]\!]\sigma'\ \mathit{in}\ (\sigma'',\, n_1 + n_2))\ .
\end{aligned}
$$

Note that, because we chose to identify a language with its semantic function, an instrumented semantics for a *base language* L defines a new, *instrumented language* $\mathcal{I}(L)$ with the same syntax as L. Irrespective of how the instrumented semantics is defined, it is sensible to prove that it agrees with the standard semantics in the obvious sense. Let T be the domain constructor for the instrumented answer domain and define function $F \in T(D) \to D$ to discard the contribution of instrumentation code. The relation $=_F$ between the original and instrumented answer domains is defined as $\{(d,\, d') \mid F\ d' = d\} \cup \{(\perp,\, \perp)\}$. We expect that $\forall c.\ \forall \sigma.\ [\![c]\!]\sigma =_F \mathcal{I}[\![c]\!]\sigma$.

A modular semantics for the base language makes it easier to define an instrumented semantics since the modifications necessary are relatively minor and do not usually warrant a separate set of definitions. Monads were originally proposed by Moggi [89] as a way of structuring semantic definitions and later applied by Wadler [139] and Steele, Jr. [125] for writing modular interpreters and by Harrison and Kamin [63] for compilers. The correspondence between monads and various programming language constructs (continuations, non-determinism, imperative features) was exploited by Steele, Jr. [125] and Liang et al. [80] for building language interpreters in a modular fashion (for the counting example above, we need a single value's worth of state and partiality).

Implementation

There are several ways of obtaining fine-grained observations about the execution of a program. The simplest is to insert custom code to make the values explicitly observable ("debugging with `printf`"). This approach has several downsides: (*i*) the process is tedious and error-prone, and (*ii*) many observations cannot be made within the language itself (e.g. the contents of the call stack in Haskell). An alternative is to write a separate *instrumented interpreter* to keep track of the necessary statistics. For example, the implementation of `CALL` might inspect the target address and update a counter accordingly:

```
switch(C[pc++]) {
case CALLr:
    target = C[pc++];
    if(R[target] == 80) counter++;
    ...
}
```

The interpreter can be implemented using any of the techniques described in the previous section. Specializing an instrumented interpreter—i.e. an interpreter for $\mathcal{I}(L)$—with respect to a program translates the program from the instrumented language into the base language L. Jones [72, Section 2.3.3] points out that "specialising an instrumented self-interpreter to a source program has the effect of inserting instrumentation code into the body of the source program". Kishon and Hudak [77] detail one instance of this approach. First, a simple functional language is extended with special label commands; a base interpreter for the language is written using open recursion, to allow later extension with execution monitors that implement different behaviours for the label commands. A partial evaluation step combines the base interpreter and an execution monitor into a single instrumented interpreter; the instrumented interpreter is specialized with respect to user programs. The efficiency of this approach depends entirely on the strength of the program specializer used which, for a given specializer, can vary widely between input interpreters. Even a Jones-optimal specializer (already a difficult proposition [82]) does not guarantee good performance of the residual programs—the Jones optimality criterion (Equation 2.8) requires the existence of a single self-interpreter for which the specializer is able to remove an entire layer of interpretational overhead, but says nothing about the variability of results produced by the specializer from one interpreter to the next.

Other authors have used ad-hoc source-to-source transformation to expose internal execution details in languages for which no good specializers exist. Allwood et al. [8] add an extra argument to Haskell functions describing where the function was called from; this is used to construct a stack back-trace for the purposes of debugging a runtime exception (e.g. as in `head []`). Wallach and Felten [143] implement security by stack inspection in Java by transforming programs into "security-passing style"; in this representation, every function accepts an extra parameter containing enough security context to allow privilege checks to be implemented. Tolmach and Appel [132] built a debugger for Standard ML that adds automatically-generated instrumentation code to the program before it is compiled. The biggest advantage of a source-to-source transformation is that the standard toolchain (including an optimizing compiler, if one is available) can be used to compile the output program. However, the decision on whether to include or exclude instrumentation code must be taken at compile-time and cannot be reversed during the program run. Tracing code is usually enclosed in a pre-processor conditional ("`#ifdef DEBUG`"); a language conditional ("`if(debug)`") can be used instead, but will degrade the perfor-

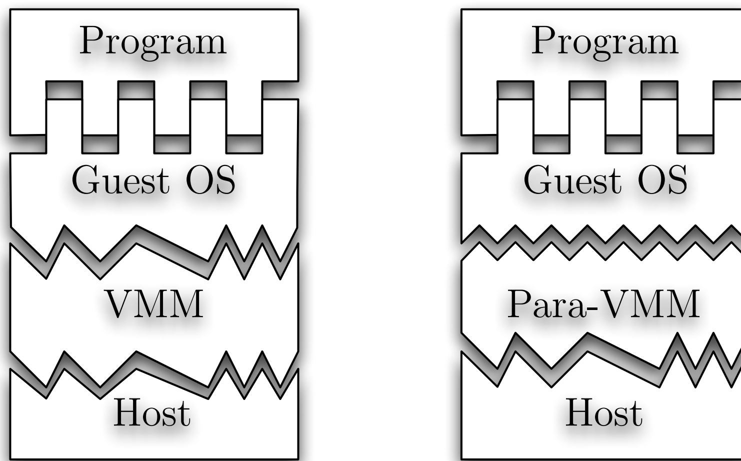


Figure 2.4: Classical virtualization vs. paravirtualization.

mance of the program even when `debug` is false[4]. In a development setting, a significant performance penalty in exchange for detailed instrumentation may be acceptable: Valgrind [99] instrumentation, described by its authors as "heavyweight", can slow down a program by orders of magnitude. By analogy, we could call DTrace [22] "lightweight" instrumentation. There is often—certainly for virtual machine monitors and lightweight instrumentation—an expectation that the program, run under the non-standard interpretation, will not perform much worse than under the original interpretation.

2.6.2 Hardware virtualization

A virtual machine monitor (VMM) oversees the concurrent execution of several *guest* virtual machines, each of which is a software likeness of the *host*, the physical machine that the VMM itself runs on. A *guest program* is a program that executes on a guest virtual machine. A virtual machine monitor is also referred to as a *hypervisor*. The virtual hardware of each guest may differ substantially from host in the number and types of devices, amount of RAM, disk space and so on. The CPU architecture of the guests is the same as that of the host, though the number of apparent virtual CPUs may be greater or smaller than the number of physical CPUs. If the total number of virtual CPUs across all guests exceeds the number of physical CPUs, guest execution is interleaved.

A simple interpreter would not make a good VMM: the overhead of interpreting each instruction (i.e. emulation) cripples performance. Popek and Goldberg [102, p. 417] suggest it should be the case that "all innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the [VMM]." The host hardware may provide assistance to the VMM by trapping those instructions that must be handled by the VMM and passing control to it. For example: the VMM may emulate the `OUT` instruction that writes directly to video memory by displaying a pixel in a window instead. This "trap-and-emulate" technology was first used in the 60s and 70s.

Early x86 VMMs relied exclusively on binary translation (replacing privileged instructions in the program code with calls into the VMM) because some virtualization-critical

[4]This is known as "the disabled probe effect".

instructions did not raise an exception even in the lowest privilege mode [109]. After two decades of neglect by x86 processor vendors, recent chips from both Intel and AMD implement trap-and-emulate in two different and incompatible ways ("VT-x" and "SMV" respectively). Finally, in *paravirtualization*, the guest operating system kernel is modified to explicitly call the VMM when it needs to perform a privileged operation. The immediate downside of paravirtualization is that it does not work with unmodified machine code—the source code must be ported to the VMM and re-compiled. The advantage is potentially better performance, since there is no need to involve the expensive trap-and-emulate mechanism.

2.6.3 Dynamic software updating

A dynamic software updating (DSU) system allows program code and data type definitions to be modified at runtime when execution reaches one of special *update points*. In other words, a DSU system is a non-standard interpreter that allows its "program text" register to be overwritten every so often when an `update` command is executed. Dynamic program instrumentation can be thought of as a dynamic software update, one where the original code is replaced with an instrumented version (or vice versa, to disable instrumentation).

On the one hand, a particular kind of static analysis for updatable programs—namely, type checking—is now well-understood thanks to the work of Hicks et al. [65], Stoyle et al. [126], Neamtiu et al. [94] and others; on the other hand, Bierman et al. [19] remark that "dynamic rebinding or update primitives invalidate general use of standard optimisations". In current systems (e.g. [27, 127, 11, 85]), updates are applied at the granularity of functions. To preserve function boundaries, some implementations disable problematic optimizations by introducing a layer of indirection into the program: i.e. functions are called through function pointers rather than directly. The key point of Popek and Goldberg's efficiency requirement for VMMs is that the common case (execution of unprivileged instructions) must not be penalised by virtualization. A program compiled for DSU incurs spurious interpretational overhead between dynamic updates (the common case) and hence runs slower than a program compiled in the usual manner.

2.7 Partial equivalence relations

Given any equivalence relation P over X, the notation $[x]_P$ (the equivalence class of x with respect to P) stands for $\{x' \mid x \; P \; x'\}$. It will be convenient to define the following two equivalence relations over a set X: $All_X = X \times X$ and $Id_X = \{(x, x) \mid x \in X\}$.

A *partial equivalence relation (PER)* is a binary relation that is symmetric and transitive. Let P be a PER over X, then the *domain* of P is the subset of X where P is reflexive: $|P| = \{x \in X \mid x \; P \; x\}$. We will write $x : P$ to mean that $x \in |P|$. Every equivalence relation is a PER and every PER is an equivalence relation on its domain. If P and Q are PERs over X and Y respectively, then $(P + Q)$, $(P \times Q)$ and $(P \Rightarrow Q)$ are PERs over $X + Y$, $X \times Y$ and $X \to Y$ respectively:

$$
\begin{array}{rlll}
\mathit{inl}\ x & (P+Q) & \mathit{inl}\ x' & \text{iff} \quad x \; P \; x' \\
\mathit{inr}\ y & (P+Q) & \mathit{inr}\ y' & \text{iff} \quad y \; Q \; y' \\
 & & & \\
(x,\, y) & (P \times Q) & (x',\, y') & \text{iff} \quad x \; P \; x' \text{ and } y \; Q \; y' \\
f & (P \Rightarrow Q) & g & \text{iff} \quad x \; P \; x' \Longrightarrow (f\ x) \; Q \; (g\ x') \,.
\end{array}
$$

Although All_X and Id_Y are equivalence relations, $All_X \Rightarrow Id_Y$ is not: the domain of this PER is the set of constant functions from X to Y. Whenever we refer to a PER over a CPO or pre-CPO, we mean a PER over its underlying set—ignoring the structure. If P is a PER over a pre-CPO X, then $P_\perp = P \cup \{(\perp,\ \perp)\}$ is a PER over the CPO $X_\perp$.

Chapter 3

Jones optimality and efficient virtualization

Many academic and commercial hardware virtualization offerings have emerged in the last few years. But the three basic requirements that a virtual machine monitor must satisfy were set out formally by Popek and Goldberg [102] in a seminal paper in 1974. First, the VMM must not impose undue overhead; second, a guest virtual machine must not be allowed to take direct control of the host's hardware resources; and third, a program run inside a virtual machine must, with caveats, produce the same result—or, exhibit the same behaviour—as when run directly on the host. These three criteria are called: "efficiency", "resource control" and "equivalence". The caveat in the equivalence criterion is there because the program may take longer to run in a virtual machine, and because the virtual hardware may differ from the physical: for example, the amount of available memory is likely to be smaller in the virtual machine, and the frame buffer of the virtual machine may be mapped onto a single window on the screen of the physical machine. In this chapter we take a language-centric approach to virtualization: our key observation is that a virtual machine monitor is, in essence, a self-interpreter for machine code. We hope that the discussion will encourage wider exchange of ideas between the virtualization and partial evaluation communities.

Outline. First, in §3.1 we introduce the main VMM implementation strategies and argue that partial evaluation naturally accounts for several kinds of virtual machine monitors. We reconstruct trap-and-emulate execution starting from *context-threaded interpretation* [18] in §3.2. A self-interpreter for a RISC subset of AL is presented in §3.3. The main payload of this chapter is contained in §3.4 where we define a version of Jones optimality over program traces which is intermediate between $Jopt_{=_\alpha}$ and $Jopt_{\leq_{\mathrm{time}}}$, and show that Popek and Goldberg's efficiency criterion can be expressed as a special case. A naive self-interpreter for AL fails to be an efficient VMM, so in §3.5 we extend AL with *virtualization assists* (AL/`EXEC`) and show that an interpreter for AL written in AL/`EXEC` is efficient in the sense above. Finally, we argue that full abstraction (described in §2.1.1) is a desirable property for VMMs (§3.6) and that it neatly captures the folklore notion of VMM *transparency*, i.e. whether or not a program can determine if it is running in a virtual machine. References to related work are given in §3.7. We conclude in §3.8 with directions for further work.

Preliminaries: privileged instructions in AL. Recall that an AL program communicates with the outside world using *ports*: the instruction IN reads a value from a port and OUT writes a value to a port. For example, IN r1,<r0> reads a value from the device attached to the port identified by register r0 and stores it in r1. Let $\xi \in \mathit{Port} \subseteq \mathit{Val}$ range over port numbers; in process calculus notation, the small-step reductions for IN and OUT are as follows. The labels on the transitions indicate that a value v is being read from ($\xi?v$) or written to ($\xi!v$) port ξ.

IN r_{dst}, $<a_{\text{port}}>$	$(pc,\, R,\, D,\, C) \xrightarrow{\xi?v} (pc+4,\, R[r_{\text{dst}} \mapsto v],\, D,\, C)$ where $\xi = V(a_{\text{port}})$
OUT $<a_{\text{port}}>$, a_{src}	$(pc,\, R,\, D,\, C) \xrightarrow{\xi!v} (pc+4,\, R,\, D,\, C)$ where $v = V(a_{\text{src}})$ and $\xi = V(a_{\text{port}})$

The IN, OUT, HLT and UPDATE instructions are privileged. For practical reasons, our virtualization setup is basic and does not deal with memory virtualization, interleaved virtual machine execution and other aspects that a VMM for real, networked x86 hardware must address. However, the essential component of virtualization as a non-standard interpretation for selected machine instructions is preserved.

3.1 Virtualization versus emulation

Let $[\![\cdot]\!]_L$ and $[\![\cdot]\!]_M$ be the evaluation functions for machine code languages L and M as implemented by the hardware. Let p be an M-machine program and v and v' range over the inputs and outputs of p respectively. An *L/M-emulator* is an L-program emu_M^L used to run M-machine code on an L-machine: the following equalities, which correspond to direct and emulated execution, hold by definition:

$$v' = [\![p]\!]_M(v) = [\![\mathit{emu}_M^L]\!]_L(p,\, v)\ . \tag{3.1}$$

DIGITAL FX!32 [28], Apple Rosetta[1] and QEMU[2] are examples of hardware emulators: all three can execute code for an architecture different from the host. Due to the difficulty of analysing low-level machine code (discussed in §2.4), an emulator must be highly conservative in its assumptions about the behaviour of emulated programs. In general, the emulator is forced to alternate between binary translation and interpreted execution, since, on a machine with data execution capability, the set of all reachable instructions cannot be determined statically. Indeed, since the instruction sets of L and M may differ substantially, there is no reason to suppose that the performance of the program under emulation will be comparable to native execution. But for some emulators, like Bochs[3], the host (emulating) and the guest (emulated) machines coincide. What makes a VMM different from an L/L-emulator is Popek and Goldberg's efficiency criterion which limits the overheads that a program may incur to those for privileged instructions only—i.e. those instructions that must be intercepted for the VMM to meet the resource control requirement. The criterion is an expression of the principle "optimize for the common case" familiar from optimizing compilers: the majority of instructions executed during a program run are—for reasonable programs—not privileged. One way of meeting the

[1] http://www.apple.com/rosetta/
[2] http://wiki.qemu.org/Main_Page
[3] http://bochs.sourceforge.net/

requirement relies on hardware support ("assists") for *trap-and-emulate* execution where privileged instructions (e.g. reading from a device) are intercepted and their handling is deferred to the VMM. Binary translation gives a means of implementing efficient VMMs on hardware architectures that do not provide virtualization assists; the translation replaces instances of privileged instructions with invocations of the VMM. *Paravirtualization* is an alternative: however, as the name suggests, paravirtualization does not qualify as virtualization in the classical sense, because a paravirtualization VMM presents a high-level *hyper-call* interface to its guest operating systems. A paravirtualization VMM is not an emulator, but rather an "an operating system for operating systems". Paravirtualization has a similar net effect to binary translation, albeit binary translation is a fully automatic process that works on compiled machine code whereas paravirtualization necessitates manual modification of the guest operating system's source code (followed by recompilation) to use the hyper-call interface. For example, the first versions of the Xen [14] hypervisor used paravirtualization exclusively: the kernel of each guest OS had to be modified to use Xen's hyper-call API ("domU"). In programming language terms, a paravirtualization VMM can be thought of as an interpreter for the language of unprivileged instructions & hyper-calls. We mention paravirtualization here for completeness: we will be primarily concerned with trap-and-emulate VMMs.

Consider an emulator fragment shown below: the code is written in a C-like language only for conciseness and should be thought of as the equivalent code in AL. Intuitively, the virtualized program on the right can be obtained by specializing the emulator (on the left) with respect to the original program.

```
switch(C[pc++]) {
case OUTir:
    i_port = C[pc++];
    r_src  = C[pc++];
    doOUTir(i_port, r_src);
    break;
}
```

Original program	**Virtualized program**
MOV r1, 3	MOV r1, 3
ADD r2, r1, 5	ADD r2, r1, 5
OUT <0>, r2	PUSH 0
	PUSH 2
	CALL doOUTir

The central argument of this chapter is that hardware virtualization can usefully be thought of in terms of interpreter specialization: i.e. $[\![\mathit{mix}]\!](\mathit{vmm}, \mathit{guest})$. Although the partial evaluation function *mix* has no immediate analogue in the virtualization literature, the choice of *mix* for the first Futamura projection gives us both naive emulators (when using a trivial specializer) as well as binary translation for stronger *mix*es. One important aspect of virtualization by binary translation is that unprivileged instructions "pass through" the translator unaltered which suggests a link to Jones optimality (recall from §2.2.1 that a Jones optimal specializer produces a residual program that is *syntactically equal* to the input program when a particular self-interpreter is specialized). Notice also that, in the figure, the call to `doOUTir` has not been inlined: this is important because it means that the virtualized program can be run under different VMMs each providing its own implementation of `doOUTir` and other `do`-procedures. This slightly unusual "semi-specialized" program structure can be adequately explained with the help of *context-threaded interpretation*.

3.2 From threaded code to trap-and-emulate

A context-threaded (CT) interpreter [18] replaces every non-branch instruction in its input program with a call to the procedure in the interpreter that implements the opcode. Once

the resulting *context threading table (CTT)* is constructed, the interpreter jumps to the first instruction in the table. Notice that (*i*) a CT interpreter is a staged program, and (*ii*) it cannot be implemented on a machine with no data execution (`eval`) capability. Instruction operands are retained in a separate table in data memory:

Original	**CTT**	(Operands)	**Partially inlined**	(Operands)
`MOV  r1, 3`	`CALL doMOVri`	`1, 3`	`MOV  r1, 3`	N/A
`ADD  r2, r1, 5`	`CALL doADDrri`	`2, 1, 5`	`ADD  r2, r1, 5`	N/A
`OUT <0>, r2`	`CALL doOUTir`	`0, 2`	`CALL doOUTir`	`0, 2`

Having a `CALL` instruction for every opcode in the program allows the interpreter to exploit return address prediction hardware—this is the original rationale for context threading. But the CTT is also an interpretation-agnostic form of the program; it can be readily executed—without parsing- or instruction-dispatch overhead—under a non-standard interpretation defined by a collection of `do`-procedures. For a set of `do`-procedures defining a self-interpreter, inlining and constant propagation can reasonably be expected to recover the original program (cf. a Jones-optimal specializer). Note that the `OUT` instruction is privileged and its implementation depends on the specific VMM used to run the program. Therefore, calls to `doOUTir`, `doOUTrr` etc. must not be inlined, but all the other `do` calls *must* (to satisfy Popek and Goldberg's requirement). Thus, the program in the right table column above executes in a *mixed compiled/interpreted mode* [151]—the bulk of the program runs natively, but the `OUT` instruction is interpreted.

Intriguingly, as a program transformation, virtualization can be thought of as selective *un-inlining* of `do`-procedures, a "de-optimization" in the words of Veldhuizen and Lumsdaine [134]. For a realistic example, consider the way paravirtualization is implemented in the Linux kernel. The hypervisor fills in a `pv_cpu_ops` structure with pointers to functions (`clts`, `write_cr0`, etc.) that emulate privileged instructions:

```
struct pv_cpu_ops {
        /* hooks for various privileged instructions */
        unsigned long (*get_debugreg)(int regno);
        void (*set_debugreg)(int regno, unsigned long value);

        void (*clts)(void);

        unsigned long (*read_cr0)(void);
        void (*write_cr0)(unsigned long);
        ...
}
```

Instead of executing a privileged instruction directly (e.g. to disable IRQs), the kernel calls the corresponding function through the structure. The code of the kernel can be thought of as a CTT where all instructions apart from the privileged ones have been perfectly inlined. The default, native `pv_cpu_ops` implementation acts directly on the hardware—i.e. it defines a self-interpreter! Notice that a paravirtualized kernel is forbidden from executing privileged instructions by convention only; there is no enforcement mechanism to prevent this.

Written in Haskell, `pv_cpu_ops` would be a type class and each hypervisor, an instance; dictionary passing replaces a global `pv_cpu_ops` variable. Dictionary passing allows for the possibility of multiple simultaneously active hypervisors. As far as we are aware, there are no current systems that support this, and the use of a global variable is another

example of a common-case optimization known as "globalization" [116]. As an aside, it is interesting to observe that this view of non-standard interpretation through context threading is closely related to Carette et al.'s [24] "final tagless" style. Carette et al. use functions rather than data constructors to represent terms of object-language programs: i.e. "`add (const 1) (const 2)`" vs. "`ADD (CONST 1) (CONST 2)`". This allows object-language code to benefit from static safety guarantees provided by the type system of the meta-language. The move from one representation to the other corresponds to a shift from an abstract syntax tree and a `switch`-based interpreter to context threading. (Note that in context threading, program state is kept in a collection of global variables, rather than being threaded through the CTT as an explicit argument to every `do` function call.) In solving different sets of problems, both the typed functional programming and the systems communities separately developed techniques based on partial evaluation.

Finally, note that, by analogy with lexical vs. dynamic scoping for variables, trap-and-emulate execution resolves the interpretation *dynamically* depending on the value of a CPU mode flag[4]. In the default CPU mode, the `OUT` instruction executes in the native hardware interpretation; once a VMM puts the CPU into virtual machine mode, subsequent occurrences of `OUT` are dispatched to the VMM-installed handler. We will return to this idea in the next chapter on instrumentation (see §4.4), where the different modes will correspond to different sets of instrumentation.

3.3 Self-interpretation in AL

Figure 3.1 shows an excerpt of a self-interpreter for a RISC subset of AL: with the exception of `MOV` and `JE` which can accept an immediate value as the first operand, operands of all other instructions must be registers. Instructions that perform useful work are shown on a grey background in the listing. For readability, we have omitted range checks on address operands. The self-interpreter uses an array of memory cells to hold the contents of the interpreted program's registers. For convenience, and due to the relative poverty of the interpreted instruction set, we choose to place this array at location zero in the data memory of the interpreter. Registers `r0` and `r1` hold useful constants; `r30` and `r31` hold the offsets of the program's code and data memories respectively; the program counter is kept in `r2`; registers `r3` to `r6` hold the opcode and the operands of the currently-executing instruction; registers `r10`, `r11` etc. contain instruction opcodes. Instructions are fetched, decoded and dispatched sequentially. Note that the `IN` and `OUT` instructions are issued by the interpreter in the same order and with the same operands as would be the case had the program been executed directly. Interpreting a program produces the same I/O behaviour—the same sequence of port/value tuples $\xi?v$ and $\xi!v$—as running the program directly. However, the handling of `UPDATE` results in a discrepancy between direct and interpreted execution. Recall that `UPDATE` exposes the entire machine state which, for the program being interpreter, includes the interpreter as well. On a von Neumann machine, it may be possible for the interpreter to *overwrite* itself with the code of the interpreted program on `UPDATE`.

Our self-interpreter is clearly not viable as a VMM. To improve its efficiency, we can either (*i*) leave the self-interpreter as is and build a good specializer for AL, or (*ii*) settle for a trivial specializer and achieve good performance by introducing trap-and-

[4]In the presence of *recursive virtualization*, the mode is not a boolean flag, but a number indicating the VMM nesting level. See §3.4.1.

```
MOV  r2, pc
MOV r10, opcode(NOP)
... ; Load remaining opcode values
dispatch: LOAD r3, (r2)
          ADD  r2, r1, r2
          ... ; Load operands into r4, r5, r6
          JE   doUPDATE, r3, r10
          JE   doHLT,    r3, r11
          JE   doNOP,    r3, r12
          ... ; Dispatch remaining opcodes
doUPDATE: UPDATE
doHLT:    HLT
doNOP:    NOP
          JMP dispatch
doMOVri:  MOV r7, r5
          STORE (r4), r7
          JMP   dispatch
doMOVrr:  LOAD  r5, (r5)
          MOV r7, r5
          STORE (r4), r7
          JMP   dispatch
doLOADrr: LOAD  r5, (r5)
          ADD   r5, r31, r5
          LOAD r5, (r5)
          STORE (r4), r5
          JMP   dispatch
```

```
doSTORErr:LOAD  r4, (r4)
          ADD   r4, r31, r4
          LOAD  r5, (r5)
          STORE (r4), r5
          JMP   dispatch
doJMPi:   ADD   r2, r30, r4
          JMP   dispatch
doJEirr:  LOAD  r5, (r5)
          LOAD  r5, (r6)
          JE  doJMPi, r5, r6
          JMP   dispatch
doADDrrr: LOAD  r5, (r5)
          LOAD  r6, (r6)
          ADD r7, r5, r6
          STORE (r4), r7
          JMP   dispatch
doINrr:   LOAD  r5, (r5)
          IN r7, <r5>
          STORE (r4), r7
          JMP   dispatch
doOUTrr:  LOAD  r4, (r4)
          LOAD  r5, (r5)
          OUT  <r4>, r5
          JMP   dispatch
```

Figure 3.1: Interpretation of selected AL instructions.

emulate assists to AL. The latter path is the one taken by the virtualization community and is explored in §3.5. But first we define an efficiency criterion for trap-and-emulate virtualization that is directly inspired by Jones optimality.

3.4 Trace simulation

Let $\mathcal{T}[\![p]\!](v)$ be the trace of program p on input v (this includes interactive input) and define *trace-equivalence* of programs ($=_{\text{trace}}$) as follows:

$$p =_{\text{trace}} p' \iff \forall v.\ \mathcal{T}[\![p]\!](v) = \mathcal{T}[\![p']\!](v)\ . \tag{3.2}$$

Intuitively, an interpreter always executes at least those instructions that a program would on its own (cf. the highlighted instructions in Figure 3.1). Let us say that a trace tr is *simulated by* tr', written $tr \preceq tr'$, iff tr is a subsequence of tr', i.e. there exists a strictly increasing function f on sequence indices such that $\forall i.\ tr(i) = tr'(f(i))$. Note that a subsequence need not be a substring: the characters of a substring occur consecutively in the string, but the elements of a subsequence merely occur in the same relative order. Define *trace-simulation* of programs ($\leq_{\text{trace}}$) accordingly by lifting $\preceq$ to programs:

$$p \leq_{\text{trace}} p' \iff \forall v.\ \mathcal{T}[\![p]\!](v) \preceq \mathcal{T}[\![p']\!](v)\ . \tag{3.3}$$

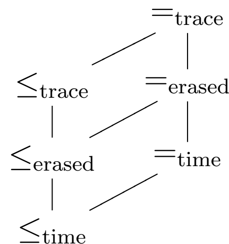
While Jones optimality in its original definition ($Jopt_{=_\alpha}$) is akin to the principle of "spatial locality" in that instructions that were close together in the original program must remain close together in the residual one. Jones optimality for traces ($Jopt_{=_{\text{trace}}}$), on the other hand, corresponds to "temporal locality": instructions that were close together *in time* in executions of the original program remain so in the residual program.

We observe that, for reasonable interpreters, the trace of the interpreter contains operations performed by the program as well as those that constitute interpretational overhead. An extreme case is a *meta-circular self-interpreter*[5] like that in the previous section, which implements each phrase of the language in terms of itself (an `ADD` with an `ADD`, a `MOV` with a `MOV`, etc.)

Instruction	**Trace**	
	Direct execution	**Self-interpreter**

`MUL r3, r1, r2`	`MUL [42]/r3, 42/r1, 1/r2`	`LOAD [42]/r5, 1/r5`
		`LOAD [1]/r6, 2/r6`
		`MUL [42]/r7, 42/r5, 1/r6`
		`STORE 2/r4, 42/r7`
		`JMP dispatch`

However, it is not true to say that the trace of the interpreter simulates the original program trace: for example, the register names for the `MUL` instruction differ between the two traces. Therefore, we will use erased traces (as defined in §2.3.2): the corresponding *erased trace-equivalence* and *erased trace-simulation* relations ($=_{\text{erased}}$ and $\leq_{\text{erased}}$) purposely conflate immediate and register operands. In particular, programs equivalent up to renaming have the same erased trace. Recall that $time(p, v)$ is the execution time of program p on input v and $=_{\text{time}}$ and $\leq_{\text{time}}$ are defined in the obvious way. Assume that trace-equivalent programs consume equal execution time, the diagram below summarises the relationships between trace-equivalence, trace-simulation and program execution time, from strongest (top) to weakest:



Erased trace simulation is deliberately a loose relation in the sense that it does not rely on an exact matching (as would a bisimulation) between the execution states under the standard and non-standard interpretations. Thus, erased trace simulation can be used uniformly with all reasonable self-interpreters, irrespective of their internal structure:

$$\forall p. \quad p \leq_{\text{erased}} [\![mix]\!](sint,\, p)\ . \tag{3.4}$$

In particular, we match up the `IN` and `OUT` instructions in the two traces, that is the interactive inputs/outputs of the program. Note that if *sint* accepts multiple input programs p_1 to p_n and alternates their execution, then $p' = [\![mix]\!](sint,\, (p_1,\, \ldots,\, p_n))$ simulates each:

$$\forall i \in \{1,\, \ldots,\, n\}.\ \forall v.\ \mathcal{T}[\![p_i]\!](v) \preceq_{\text{erased}} \mathcal{T}[\![p']\!](\ldots,\, v,\, \ldots)\ . \tag{3.5}$$

[5]The term "meta-circular interpreter" is due to Reynolds [107].

These equations are statements of correctness rather than efficiency. The essential point is that *when there is no interpretational overhead,* $[\![mix]\!](sint, p)$ *executes at most those operations that p would on its own.* Thus, $Jopt_{=_{\text{erased}}}$ is an intermediate criterion that sits between the established definitions of Jones optimality $Jopt_{=_\alpha}$ and $Jopt_{\leq_{\text{time}}}$.

3.4.1 Trace simulation modulo privileged instructions

Jones optimality in its original definition is only applicable to self-interpreters; but VMMs are "mostly self-interpreters". More precisely, a VMM behaves like a self-interpreter for unprivileged instructions (the common case), but implements a custom behaviour for privileged operations. A VMM is considered efficient provided "all innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the [VMM]" [102]. That is, no overhead should be incurred when executing non-privileged instructions such as register movements and ALU operations. Privileged instructions are exempt from this requirement as their implementation may need to emulate devices which are not physically present in the machine: it is possible that a privileged instruction may be emulated entirely by non-privileged instructions. Therefore, a privileged operation in the native execution trace of a program corresponds to a sequence of possibly-privileged operations in the trace of the virtualized program. Note that although execution of non-privileged instructions is performed by the CPU directly, they are handled differently in virtual machine mode. For example, both code and data memory addresses are offset with the base address of the virtual machine's memory with respect to the host's memory, e.g.:

Instruction	**Trace**	
	Native execution	**Trap-and-emulate VM**
`LOAD r1, (42)`	`LOAD [3]/r1, (42/i)`	`LOAD [3]/r1, ( 1046/i )`
`ADD  r2, r1, 5`	`ADD  [8]/r2, 3/r1, 5/i`	`ADD  [8]/r2, 3/r1, 5/i`
`OUT <0>, r2`	`OUT  0/i, 8/r2`	...; VMM handler prologue
		`OUT 27/i, 8/r2`
		...; Cleanup

A trace always corresponds to actual hardware operations executed ("microcode"): in other words, a VM trace is not a trace of the virtual machine but a trace of the host running the virtual machine. In the trap-and-emulate VM trace above, the VM's data memory is located at offset 1024 in the host's data memory ($1024 + 42 = 1046$), and we have assumed that the virtual registers are mapped onto the hardware registers. Note the difference between the VM trace above and the trace of a self-interpreter on p. 41.

Several VMMs may be *nested* forming a tower of interpreters, and so we annotate every operation in the trace of a program with a natural number n indicating the *virtualization nesting level* at which the operation was issued: $n = 0$ corresponds to direct execution on hardware. The current nesting level is incremented on entry into a virtual machine and decremented when control is transferred into the VMM to handle a privileged instruction.

Define a *trace projection* π_n which discards emulation of privileged instructions from the trace of a virtualized program. Assume the VMM is executing at nesting level n, then $\pi_n(tr)$ is the subtrace of tr consisting of only those operations executed at levels greater than n, i.e. those that must be attributed to the guest program. A second complementary projection $(\cdot)^{\text{unpriv}}$ removes all privileged operations from a trace. Applying π_0 to the VM trace and $(\cdot)^{\text{unpriv}}$ to the native execution trace we recover our intuitions from the previous section:

Projections for traces on p. 42	
$(\cdot)^{\text{unpriv}}$ **Native**	π_0 **VM**
`LOAD [3]/r1, (42/i)`	`LOAD [3]/r1, 1046/i`
`ADD  [8]/r2, 3/r1, 5/i`	`ADD  [8]/r2, 3/r1, 5/i`

Let tr and tr' be shorthand for the erased traces of VM and native execution respectively. We say that p' *virtualizes* p *correctly* iff $\forall v.\ (tr)^{\text{unpriv}} \preceq \pi_0(tr')$. This definition applies to a VMM that runs directly on hardware (virtualization nesting level zero). Conversely, p' virtualizes p *efficiently* iff $\forall v.\ \pi_0(tr') \preceq (tr)^{\text{unpriv}}$. Efficiency requires that, modulo emulation of privileged instructions, every operation in the trace of the virtualized program can be accounted for. Finally, define trace-equivalence between original and virtualized programs ($p =_{\text{virt}} p'$) as follows:

$$p =_{\text{virt}} p' \iff \forall v.\ (tr)^{\text{unpriv}} \preceq \pi_0(tr') \preceq (tr)^{\text{unpriv}}\ . \tag{3.6}$$

To pinpoint the emulation code in the trace of the virtualized program, the definitions above rely on the virtualization nesting level annotation being present. Equivalently, we require an indication in the trace where execution passes into and out of the virtual machine monitor (in the next section we extend AL with a virtualization-assist instruction for this purpose). However, this means that the definitions only work for trap-and-emulate VMMs: for binary translation VMMs, the projection π_0 must be constructed manually.

3.5 Virtualization assists

In this section we consider two alternative designs for virtualization assists in AL.

3.5.1 AL/`STEP`

Volume 3B of the *Intel 64 and IA-32 Architectures Software Developer's Manual*[6] states that (p. 18-12): "The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. [...] the exception is generated after the instruction is executed." To speed up the self-interpreter on non-privileged instructions without relinquishing control over execution of privileged ones, we add a `STEP` instruction to AL with allows single-stepping through guest program code. The sole operand of `STEP` is a register containing the address in data memory of a *Virtual Machine Control Block (VMCB)*. The layout of the VMCB is shown on the right in Figure 3.2. Recall that our self-interpreter of Figure 3.1 uses the first thirty-two locations in data memory to store the registers of the interpreted program. The VMCB duplicates this structure and plays a role similar to a process control block in an operating system kernel: a VMCB contains the state of a virtual machine including the program counter, register values, the code and data memories and their sizes. `STEP` executes a single non-privileged instruction at the location identified by the program counter field of the VMCB, taking care to offset all memory accesses (including branch targets) performed by the instruction. The instruction is executed as if the state of the VMCB were the real machine state. For example, given "`JMP 8`" the effect of `STEP` is to set the program counter field of the VMCB to 8; given "`LOAD r1, (24)`", and assuming the data memory of the virtual machine starts at offset 32, the value is loaded from

[6] `http://www.intel.com/products/processor/manuals/index.htm`

```
           MOV  r31, 0           ; offset of VMCB
           MOV  r10, opcode(HLT)
           MOV  r11, opcode(UPDATE)
           MOV  r12, opcode(IN)
           MOV  r13, opcode(OUT)
dispatch:  ... ; Load PC field of VMCB into r2
           ... ; Check PC is in range
           LOAD  r3, (r2)

           JE doHLT,    r3, r10
           JE doUPDATE, r3, r11
           JE doIN,     r3, r12
           JE doOUT,    r3, r13

           STEP r31
           ... ; Check status field of VMCB
           JMP  dispatch
doHLT:     ... ; Emulation of HLT
doUPDATE:  ... ; Emulation of UPDATE
doIN:      ... ; Emulation of IN
doOUT:     ... ; Emulation of OUT
```

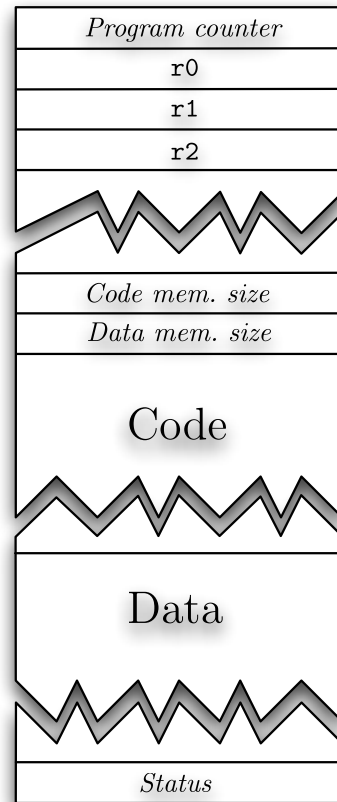


Figure 3.2: An interpreter in AL/STEP and the VMCB.

physical data memory address 56 into the r1 field of the VMCB. The STEP instruction populates the status field of the VMCB with an error value if execution of the virtual machine aborts (e.g. the instruction makes an out-of-range memory access). Note that the special interpretational encoding of the program state in the meta-circular interpreter is now uniformly captured (and more likely to be implemented efficiently) by STEP. An interpreter in AL/STEP is shown in Figure 3.2: it relinquishes some control over execution but has a much simpler implementation. However, this interpreter fails to be efficient due to the overhead of instruction dispatch. To eliminate this overhead, we now introduce an alternative instruction, closer to the spirit of trap-and-emulate virtualization.

3.5.2 AL/EXEC

AL/EXEC is an extension of AL with the EXEC instruction which has a single register operand indicating the location of a VMCB. Unlike STEP, EXEC executes any number of instructions rather than just a single one: when a privileged instruction is encountered, execution continues at the instruction following EXEC which may then determine the opcode and operands of the offending instruction via the program counter field of the VMCB. EXEC increments the virtualization nesting level on entry and decrements it on exit. Thus, in the trace, instructions that are executed by EXEC are marked with the correct level. Note that the sections of interpreter code which are responsible for emulating the privileged instructions are executed at a level one lower that those of the program being interpreted. Recall (Equation 3.6) that the π_0 projection strips out the emulation.

```
          MOV  r31, 0          ; offset of VMCB
dispatch: EXEC r31
          ... ; Check status field of VMCB
          ... ; Load opcode and operands from VMCB
          JE doHLT,    r3, r10
          JE doUPDATE, r3, r11
          JE doIN,     r3, r12
          JE doOUT,    r3, r13
doHLT:    ... ; Emulation of HLT
doUPDATE: ... ; Emulation of UPDATE
doIN:     ... ; Emulation of IN
doOUT:    ... ; Emulation of OUT
```

Figure 3.3: An interpreter in AL/`EXEC`.

An interpreter in AL/`EXEC` is shown in Figure 3.3. To draw a functional programming analogy, in AL/`STEP` the VMM is supplied with the hardware implementation of unprivileged instructions. In AL/`EXEC` the instruction dispatch loop is inverted and the hardware, rather than the interpreter, is responsible for dispatch. Let `st` = *State*, and `unpriv` and `priv` be states where the instruction pointed to by the program counter is unprivileged or privileged, respectively:

AL/STEP	**AL**/EXEC
`vmm : st -> (unpriv -> st) -> st`	`vmm : priv -> st`
`hw  : unpriv -> st`	`hw  : st -> (priv -> st) -> st`

Uses of the `STEP` instruction correspond to invocations of `hw` (on the left), and the "callbacks" to the VMM which `EXEC` makes for privileged instructions correspond to invocations of `vmm` (on the right). Assuming that the majority of operations in any given program are unprivileged, the AL/`EXEC` setup results in a smaller number of calls.

3.6 An application of full abstraction to VMMs

A guest program should not be able to detect that it is executing inside a virtual machine. In the period before virtualization became popular, a similar requirement was often applied to system-level debuggers. Various methods of debugger detection and circumvention, collectively known as "anti-debugging" techniques, were proposed[7], and some viruses are known to become actively hostile when executed under a debugger. Anti-debugging tricks are also used by content-protection schemes to actively resist analysis and reverse-engineering.

Programmatic tests to detect the presence of a virtual machine monitor rely on observable discrepancies between two expressions or code blocks known to be equivalent when executed directly on hardware but behave differently in a virtual machine. Garfinkel et al. [54] define VMM *transparency* as "making virtual and native hardware indistinguishable under close scrutiny by a dedicated adversary". They argue on pragmatic grounds that transparency is not an achievable or desirable goal on real-world hardware. Indeed, some

[7]E.g.: `http://www.symantec.com/connect/articles/windows-anti-debug-reference`

VMMs (and debuggers) do not attempt to hide their presence at all, and many of those that do are unsuccessful [44, 103]. While full transparency may not be practically attainable, we argue that it is nevertheless a useful guiding principle for VMM design and potentially hardware design as well. Extensionally, transparency is covered by the "equivalence" requirement for VMMs, analogous to the self-interpreter correctness equation, i.e. that $\forall v.\ [\![p]\!](v) = [\![sint]\!](p,\, v)$. Intuitively, whether p does or does not detect the presence of *sint* is irrelevant as long as it produces the same final answer. However, the definition of equivalence is very coarse-grained.

Suppose the attacker is allowed to submit arbitrary terminating programs for execution on a "black box" computer of a given architecture which may be a physical machine or a VM. The attacker wants to determine which it is, based on the output and the running times of the programs. Consider a setup where programs must be submitted one by one: i.e. each program is executed on a randomly-chosen black box. The equivalence requirement for VMMs guarantees that the output of the program executed by a VM is identical to its output when executed natively. The running time of the program cannot be used as a distinguishing criterion either: even if the running time is excessively large, there is no way for the attacker to be sure that the program was executed on a VM rather than a very slow physical machine. Suppose instead that the attacker is allowed to submit *two programs* at a time: the programs are run consecutively on the same black box. In this case, the attacker can *compare the running times of two programs*: e.g. p_1 (with lots of privileged instructions) is likely to run *significantly* slower than p_2 (with none) on a VM vs. a physical machine. We argue that transparency is a *relational* concept that cannot be attributed to a given run of the VMM, and Abadi's full abstraction for language translations (§2.1.1) is a natural formalism for this.

Let $\mathcal{V}$ be a translation from AL to "virtualized AL" which has the same syntax, but a different interpretation of certain instructions. The translation is not semantics-preserving since, in general, $\mathcal{V}(p)$ does not exhibit the same I/O behaviour as p (for example, the ports read and written to may be different—see p. 42). Recall that a translation $\mathcal{C}$ is fully abstract whenever $e \sim_L e' \iff \mathcal{C}(e) \sim_M \mathcal{C}(e')$ where $\sim_L$ and $\sim_M$ are equivalence relations over languages L and M respectively. Let us call $\mathcal{V}$ *transparent with respect to* R iff

$$p\ R\ p' \iff \mathcal{V}(p)\ R\ \mathcal{V}(p')\ . \tag{3.7}$$

First, take R to be contextual equivalence on AL and $\mathcal{V}$ to be identity (recall that in classical virtualization a program does not need to be modified to run in a VM).

Consider an extreme example of the failure of full abstraction caused by the VMM revealing its presence. Suppose the VMM makes the first data memory cell read-only and places a special marker (e.g. the value 57) there. Let p_1 and p_2 be the instruction sequences below and note that p_2 is obtained from p_1 by eliminating a redundant `LOAD`.

p_1	p_2
`MOV   r1, 57`	`MOV   r1, 57`
`STORE (0), r1`	`STORE (0), r1`
`LOAD  r1, (0)`	—
`OUT   <0>, 29`	`OUT   <0>, 29`

With the exception of contexts that jump into the middle of the term[8], there are no contexts in AL that can distinguish p_1 and p_2. But under virtualization we have

Ctx[−] = "[−]; `JE p2,r1,57; OUT <0>,1; HLT; p2: OUT <0>,2; HLT;` .

[8]For example, Ctx[−] = "[−]; `JMP 0; ...`".

Of course, a realistic VMM will not break the semantics of the language so egregiously. However, it may fail to preserve finer relations on programs which cannot be expressed as internal language observations (an AL program cannot measure the execution time of one of its subprograms). Therefore, we will now take R to capture an *external observation* that the adversary is able to perform. For example, by letting R be $\leq_{\text{time}}$ (note that $\leq_{\text{time}}$ is non-symmetric), we ensure that $\mathcal{V}$ preserves the relative speed of programs: an external observer cannot determine whether it is observing virtualized or native executions by running p and p' and comparing their running times with $\leq_{\text{time}}$.

3.7 Related work

We give a few pointers to relevant material on virtualization and Jones optimality.

A VMM has to address some of the same scheduling and process isolation problems as an operating system kernel, and the similarities and differences between VMMs and kernels are a source of some friction [110]. The landmark paper of Popek and Goldberg [102] (see also discussion of same in [124, Chapter 8]) establishes architectural requirements for virtualization in a formal manner. Robin and Irvine [109] investigate the feasibility of secure virtual machines on the Intel Pentium. Intel's VT virtualization extensions for the x86 architecture and the Itanium are described by Neiger et al. [97]. Adams and Agesen [5] discuss the pros and cons of these hardware assists compared with previously used techniques for x86 virtualization.

Makholm [82] has a good introduction to Jones optimality. More recently, Danvy and López [36] established a link between Jones-optimal specialization and higher-order abstract syntax. Glück [56] showed that for any Jones-optimal specializer in a particular class, for any given translation, there exists an interpreter which, under specialization, will yield programs "no worse" than the translation. Gade and Glück [53] give a formal argument of Jones optimality for the specializer Unmix.

Full abstraction is best known as a notion of agreement between the denotational and operational semantics of a particular language (see §2.1.1 and [148]). Notice that full abstraction captures a notion of "continuity" in compilation: a malicious compiler, such as that of Thompson [131], must either miscompile all programs in an equivalence class or none at all. (Thompson's trojan compiler inserts spurious malicious code when compiling either itself or the `login` program.)

3.8 Conclusions and further work

We have described our efforts to relate well-established concepts from partial evaluation with an important practical application: virtualization. Applying familiar programming language theory to virtualization gives a unifying account of the two main definitions of interpretational overhead: Jones optimality for program specializers and Popek and Goldberg's VMM efficiency. We introduced AL, an assembly language for an idealized machine with I/O devices. We showed a self-interpreter for AL as well as interpreters for two alternative extensions of AL with hardware virtualization assists: AL/`STEP` and AL/`EXEC`. Finally, we showed a possible formalization of VMM transparency in terms of full abstraction.

We are intrigued about the possibility of *typed assembly languages* with first-class support for privileged instructions, as a first step towards formalizing a minimal trusted

virtual machine, much in the spirit of the work on proof-carrying code [95, 10].

Existing VMM implementations tend to conflate *policy* (e.g. how a virtual device is emulated) with *mechanism* (i.e. how privileged instructions are intercepted and the VM state maintained), but each mechanism has its downsides: binary translation VMMs are difficult to implement, simple emulation gives poor performance, and trap-and-emulate virtualization requires hardware support. Irrespective of the implementation mechanism, we still expect the VMM to have the same properties (such as transparency). What is lacking is a means of *specifying* the behaviour of the VMM from which a concrete implementation (based on either emulation, trap-and-emulate, binary translation or a combination of all three) can be instantiated. We speculate that partial evaluation could play an important role in allowing this to happen since the Futamura projections (§2.2) already give us the means to derive program transformations from interpreters. Instantiate Equation 3.7 for an interpreter *vmm* and a program specializer *mix*:

$$p \sim p' \iff [\![mix]\!](vmm, p) \sim [\![mix]\!](vmm, p') \; . \tag{3.8}$$

We intuit that full abstraction may prove to be a useful notion in designing specializers for virtualization as well as CPU instruction sets.

Chapter 4

Formally efficient program instrumentation

In the last chapter we showed how Popek and Goldberg's efficiency requirement for virtual machine monitors can be reconciled with Jones optimality. Program instrumentation is another form of augmented execution where performance can be important. There is a wealth of literature on the subject of program instrumentation, and many tools are used in practice. Implementation methods range from manually adding `printf` calls to purpose-built frameworks with virtual machines and just-in-time compilers. Historically, instrumentation was seen as a debugging aid, unsuitable for use in production systems primarily because the runtime overhead of instrumentation can slow down a program by orders of magnitude. This traditional view is challenged by modern, lightweight dynamic binary instrumentation frameworks—of which DTrace (developed by Sun Microsystems) is perhaps the best-known example—that allow instrumentation code to be selectively enabled and disabled at runtime. The performance implications of an instrumentation framework largely determine its adoption. Although there is broad informal agreement that only those parts of the program that are being instrumented should incur a performance penalty, no formal criteria exist. The key contribution of this chapter is a proposal for such a criterion. We also attempt to clarify the relation of lightweight instrumentation to Jones-optimal specialization and Popek&Goldberg-efficient virtualization.

Outline. In §4.1 we introduce IL, an instrumentation language loosely based on DTrace's "D". An IL script associates AL instructions with instrumentation code to be run when the corresponding instruction is executed. An *instrumenting function* combines an IL script with the AL program to be instrumented. We define what it means for an instrumenting function to be *faithful* in §4.2. There are many reasonable choices of instrumenting function, some clearly better than others. In §4.3 we add a breakpoint instruction (`BRK`) to AL and give a definition of *efficient* instrumentation inspired by Popek and Goldberg's definition of efficiency for VMMs. Next, in §4.4, we consider scoping of instrumentation code: i.e. allowing multiple IL scripts to be simultaneously applied to the same program. Since each IL script implicitly defines an instrumented language, we borrow some ideas from the multi-language interoperability literature. In particular, we recover the notion of *boundaries* between languages by reversing a super-instruction optimization on the combined instrumentation language. Note that §4.2 and §4.3 deal with explicit overheads of instrumentation, but "implicit" overheads remain because code optimization opportunities are lost to allow potential instrumentation. Therefore, in §4.5, we describe linguistic

$$\begin{aligned} exp &::= val \mid var \mid opvar \mid \mathsf{pc} \mid \mathsf{reg}(exp) \mid \mathsf{code}(exp) \mid \mathsf{data}(exp) \mid op\ exp^{+} \\ comm &::= comm;\ comm \mid \mathsf{if}\ exp\ \mathsf{then}\ comm \mid var \leftarrow exp \mid \mathsf{print}\ exp^{+} \\ pat &::= opcode\ opvar^{\star} \quad rule ::= pat\ \{\ comm\ \} \quad script ::= rule^{+} \end{aligned}$$

Figure 4.1: Syntax of IL.

restrictions on IL scripts to recover such lost ground. Related work is covered in §4.6. We conclude (§4.7) with an outlook to further work.

4.1 IL

Although it is more conventional to instrument programs written in high-level languages, we will continue to use the same assembly language (introduced in Chapter 2) as in the last chapter in order to emphasize the similarities between virtualization and instrumentation. Like a conditional breakpoint in a debugger, instrumentation code in principle can be triggered by an arbitrary set of runtime conditions. Indeed, an instrumentation script serves a similar purpose to a *debug script* (a small program that drives the execution of another program under a debugger). For example, the GNU debugger gdb can be set up to invoke a script every time a specific breakpoint is reached and the script is then free to perform any operation on the program being debugged. Since the performance implications of this are severe, debug scripts are not a good way of obtaining insight into running production applications. Consequently, compared to debug scripts, instrumentation scripts are significantly restricted in how they interact with the instrumented program and when they can be invoked. For obvious reasons, instrumentation scripts must always terminate. IL is a instrumentation language in the spirit of "D" [22] with syntax shown in Figure 4.1. We restrict our attention to instrumentation of individual AL instructions. An example IL script is shown below:

$$\begin{array}{ll} \texttt{STORE (}\,r_{\mathrm{dst}}\,\texttt{)},\ r_{\mathrm{src}} & \{\ x \leftarrow \mathsf{data}(12345678);\ \mathsf{print}\ x;\ \} \\ \texttt{JZ}\ r_{\mathrm{loc}},\ r & \{\ \mathsf{if}\ \mathsf{code}(\mathsf{reg}(r_{\mathrm{loc}})) = opcode(\texttt{NOP})\ \mathsf{then}\ y \leftarrow y+1;\ \mathsf{print}\ y;\ \} \end{array}$$

IL rules pattern-match on AL syntax constructors: the *operand variables* (*opvar*s) are bound to the operands of the trigger instruction. For example, in the script above, if the rule for STORE is triggered by the instruction "STORE (r1), r0", then r_{dst} and r_{src} would be bound to 1 and 0 respectively. We do not enforce any particular naming convention for operand variables, but we do require that *opvars* be syntactically distinct from ordinary variables: this is necessary to prevent assignments to *opvars* (see below). The order of the rules is not important; each opcode can only be instrumented once. Instrumentation code is run before the corresponding instruction takes effect. The pc, reg, code and data primitives give instrumentation code access to the program's state. The value of pc is set to the address of the triggering instruction.

Scripts are composed by catenation with capture-avoiding renaming of variables. For example, let t_1 be the sample script on the previous page and t_2 be the script

$$\texttt{STORE (}\,r_{\mathrm{dst}}\,\texttt{)},\ r_{\mathrm{src}}\ \ \{\ x \leftarrow \mathsf{reg}(r_{\mathrm{src}});\ \mathsf{print}\ x;\ \}\ .$$

The combined script $t_1 \oplus t_2$ is obtained by appending (*i*) the rules of t_2 to those of t_1, and (*ii*) the commands in the rule for STORE in t_2 to those in the corresponding rule in

$$
\begin{array}{rcl}
[\![e]\!]^{exp}_{\mathrm{IL}} & : & \mathit{State} \times \mathit{VEnv} \to \mathit{Val} \\
\\
[\![v]\!]^{exp}_{\mathrm{IL}} s\pi & = & v \\
[\![x]\!]^{exp}_{\mathrm{IL}} s\pi & = & \pi(x) \\
[\![\mathsf{pc}]\!]^{exp}_{\mathrm{IL}} s\pi & = & pc_s \\
[\![\mathsf{reg}(x)]\!]^{exp}_{\mathrm{IL}} s\pi & = & R_s(\pi(x)) \\
[\![\mathsf{code}(x)]\!]^{exp}_{\mathrm{IL}} s\pi & = & C_s(\pi(x)) \\
[\![\mathsf{data}(x)]\!]^{exp}_{\mathrm{IL}} s\pi & = & D_s(\pi(x)) \\
[\![op\ x_1,\ x_2]\!]^{exp}_{\mathrm{IL}} s\pi & = & op(\pi(x_1),\ \pi(x_2))
\end{array}
$$

$$
\begin{array}{rcl}
[\![c]\!]^{comm}_{\mathrm{IL}} & : & \mathit{State} \times \mathit{VEnv} \to \mathit{VEnv} \times \mathit{Output} \\
\\
[\![c_1; c_2]\!]^{comm}_{\mathrm{IL}} s\pi & = & let\ (\pi',\ o_1) = [\![c_1]\!]^{comm}_{\mathrm{IL}} s\pi\ in \\
 & & \quad let\ (\pi'',\ o_2) = [\![c_2]\!]^{comm}_{\mathrm{IL}} s\pi'\ in \\
 & & \quad (\pi'',\ o_1 +\!\!+\ o_2) \\
[\![\mathsf{if}\ x\ \mathsf{then}\ c]\!]^{comm}_{\mathrm{IL}} s\pi & = & if\ \pi(x)\ then\ [\![c]\!]^{comm}_{\mathrm{IL}} s\pi \\
 & & \quad else\ (\pi,\ \{\}) \\
[\![x \leftarrow e]\!]^{comm}_{\mathrm{IL}} s\pi & = & (\pi[x \mapsto [\![e]\!]^{exp}_{\mathrm{IL}} s\pi],\ \{\}) \\
[\![\mathsf{print}\ x]\!]^{comm}_{\mathrm{IL}} s\pi & = & (\pi,\ \pi(x))
\end{array}
$$

Figure 4.2: Semantics of $\mathrm{IL}_{\mathrm{ANF}}$.

t_1, taking care to rename x to a fresh variable name z:

$$
\begin{array}{ll}
\texttt{STORE (}\ r_{\mathrm{dst}}\ \texttt{),}\ r_{\mathrm{src}} & \{\ x \leftarrow \mathsf{data}(12345678);\ \mathsf{print}\ x;\ z \leftarrow \mathsf{reg}(r_{\mathrm{src}});\ \mathsf{print}\ z;\ \} \\
\texttt{JZ}\ r_{\mathrm{loc}},\ r & \{\ \mathsf{if}\ \mathsf{code}(\mathsf{reg}(r_{\mathrm{loc}})) = opcode(\texttt{NOP})\ \mathsf{then}\ y \leftarrow y + 1;\ \mathsf{print}\ y;\ \}\ .
\end{array}
$$

Note that the operator $\oplus$ is not commutative, but, since *opvar*s cannot be assigned to and all other variables are subject to capture-avoiding renaming, there is no danger of interference between the scripts.

4.1.1 Semantics

Recall from §2.3 that the state of an AL program is a tuple $s = (pc,\ R,\ D,\ C)$ consisting of the program counter pc, register file R and data and code memories D and C. Figure 4.2 shows the semantics of expressions and commands of an A-normal form (ANF) subset of IL, where all intermediate results are explicitly named. The environment $\mathit{VEnv} \stackrel{\mathrm{def}}{=} \mathit{Var} \to \mathit{Val}$ holds bindings of IL variables; *Output* is a sequence of port/value pairs. The functions R, D and C are total: given an out-of-range argument they return the special error value $err \in \mathit{Val}$. If err arises at any point during evaluation of a script, the behaviour of the instrumented program is undefined.

4.1.2 Instrumenting AL programs

Figure 4.3 shows a possible translation of IL expressions and commands to AL. The symbol table $\mathit{VEnv}^{\#}$ maps variable names to locations in data memory; the "state" holds addresses of four functions used to retrieve the values of the program counter, registers, and code and data memory cells of the instrumented program (i.e. $\mathit{State}^{\#}$ is an interface to an interpreter). It is easy to see how the AL code produced by this translation can be inserted into a self-interpreter for AL (e.g. Figure 3.1 on p. 40 in the previous chapter) to yield an instrumented interpreter. We therefore assume the existence of an auxiliary function $\mathcal{I}(t)$ that maps an IL script t to an instrumented interpreter for AL which is itself written in AL. An *instrumenting function* $\mathcal{C}_t(p)$ augments an AL program p using script t, producing an instrumented AL program. Given any specializer *mix* for AL, one possibility is to define $\mathcal{C}_t(p)$ as $[\![mix]\!](\mathcal{I}(t),\ p)$.

Dynamic instrumentation is the ability to add and remove instrumentation at runtime. Notice that dynamic instrumentation is a form of dynamic software updating, and every

$$\langle\!|e|\!\rangle^{exp}_{\mathrm{IL}} : State^{\#} \times VEnv^{\#} \to \mathrm{AL}$$

$$\begin{aligned}
\langle\!|v|\!\rangle^{exp}_{\mathrm{IL}} s\pi &= \texttt{MOV r0, } v \\
\langle\!|x|\!\rangle^{exp}_{\mathrm{IL}} s\pi &= \texttt{LOAD r0, } (\pi(x)) \\
\langle\!|\mathsf{pc}|\!\rangle^{exp}_{\mathrm{IL}} s\pi &= \texttt{CALL } pc_s \\
\langle\!|\mathsf{reg}(x)|\!\rangle^{exp}_{\mathrm{IL}} s\pi &= \langle\!|x|\!\rangle^{exp}_{\mathrm{IL}} s\pi\texttt{; CALL } R_s \\
\langle\!|\mathsf{code}(x)|\!\rangle^{exp}_{\mathrm{IL}} s\pi &= \langle\!|x|\!\rangle^{exp}_{\mathrm{IL}} s\pi\texttt{; CALL } C_s \\
\langle\!|\mathsf{data}(x)|\!\rangle^{exp}_{\mathrm{IL}} s\pi &= \langle\!|x|\!\rangle^{exp}_{\mathrm{IL}} s\pi\texttt{; CALL } D_s \\
\langle\!|op\ x_1,\, x_2|\!\rangle^{exp}_{\mathrm{IL}} s\pi &= \texttt{LOAD r1, } (\pi(x_1)) \\
&\quad\ \texttt{LOAD r2, } (\pi(x_2)) \\
&\quad\ \texttt{ALU}(op)\texttt{ r0, r1, r2}
\end{aligned}$$

$$\langle\!|c|\!\rangle^{comm}_{\mathrm{IL}} : State^{\#} \times VEnv^{\#} \to \mathrm{AL}$$

$$\begin{aligned}
\langle\!|c_1; c_2|\!\rangle^{comm}_{\mathrm{IL}} s\pi &= \langle\!|c_1|\!\rangle^{comm}_{\mathrm{IL}} s\pi;\ \langle\!|c_2|\!\rangle^{comm}_{\mathrm{IL}} s\pi \\
\langle\!|\mathsf{if}\ x\ \mathsf{then}\ c|\!\rangle^{comm}_{\mathrm{IL}} s\pi &= \langle\!|x|\!\rangle^{exp}_{\mathrm{IL}} s\pi \\
&\quad\ \texttt{JZ } \ell\texttt{, r0} \\
&\quad\ \langle\!|c|\!\rangle^{comm}_{\mathrm{IL}} s\pi \\
&\quad\ \ell\texttt{:}\quad \text{where } \ell \text{ is fresh.} \\
\langle\!|x \leftarrow e|\!\rangle^{comm}_{\mathrm{IL}} s\pi &= \langle\!|e|\!\rangle^{exp}_{\mathrm{IL}} s\pi \\
&\quad\ \texttt{STORE } (\pi(x))\texttt{, r0} \\
\langle\!|\mathsf{print}\ x|\!\rangle^{comm}_{\mathrm{IL}} s\pi &= \langle\!|x|\!\rangle^{exp}_{\mathrm{IL}} s\pi \\
&\quad\ \texttt{OUT <0>, (r0)}
\end{aligned}$$

Figure 4.3: Translation of $\mathrm{IL}_{\mathrm{ANF}}$ to AL.

`UPDATE` instruction marks an update point. Somewhat counter-intuitively, we treat runtime instrumentation (and later dynamic updating) as a meta-linguistic operation which can be performed only when the program reaches an `UPDATE` instruction and terminates. This requires an instrumentation and a "de-instrumentation" function for *program states*: $\mathcal{C}^{State}_t$ and $\mathcal{D}^{State}_t$ where the former instruments a program state and the latter projects the program state from a state of instrumented execution. For example, suppose a program p instrumented with script t_1 executes `UPDATE` and terminates with result *upd* s where s is a complete execution state $(pc,\, R,\, D,\, C)$. To instrument the program with script t_2, execution is restarted with state $(\mathcal{C}^{State}_{t_2} \circ \mathcal{D}^{State}_{t_1})\ s$. For the self-interpreter of Figure 3.1, the injection $\mathcal{C}^{State}$ from a machine execution state into an interpreted state is defined in the figure below; the definition of $\mathcal{D}^{State}$ is analogous.

$\mathcal{C}^{State}$ given input state $s = (pc,\, R,\, D,\, C)$.
$pc' = 0$
$R' =$ | 0 | 0 | 0 | $pc + 4 +$ offset of C | … | offset of C | offset of D | … |
$D' =$ | $R(0)$ | $R(1)$ | … | $C(0)$ | $C(1)$ | … | $D(0)$ | $D(1)$ | … |
$C' =$ Code in Figure 3.1.

Figure 4.4: $\mathcal{C}^{State}$ for the self-interpreter in Figure 3.1.

4.2 Faithful instrumentation

An instrumented interpreter is not a self-interpreter, so Jones optimality, in its original formulation, does not apply. But we argue that it is still useful to define—explicitly and independently of the interpreter—the relation between instrumented and non-instrumented ("plain") computations.

Jones [72] notes that original and specialised computations can (for reasonable specializers) be related by "execution order"-preserving maps. In the previous chapter we showed that a version of Jones optimality built on this observation—which we called "Jones optimality for traces"—can capture Popek and Goldberg's [102] efficiency criterion. Recall

that, to provide isolation between individual virtual machines, a virtual machine monitor must emulate privileged instructions (roughly, those that affect the operation of the CPU itself or other hardware) in software, and the efficiency requirement states that non-privileged instructions must be executed directly by the hardware with no intervention by the VMM. This mirrors Cantrill et al.'s [22] claim that "when DTrace is not in use, the system is just as if DTrace were not present at all". The key point is that common-case performance—i.e. execution of unprivileged instructions for VMMs and non-instrumented execution for DTrace—should not degrade. For example, instrumentation by naive interpretation is clearly not "good" because for every unit of useful work in the program, the interpreter executes many housekeeping instructions.

Let tr and tr' be the erased traces of programs p and p' respectively. Then p is erased trace-simulated by p' ($p \leq_{\text{erased}} p'$) iff tr is a subsequence of tr', i.e. $\exists f.\ \forall i.\ tr(i) = tr'(f(i))$ where f is a strictly increasing function (see §3.4). For many reasonable instrumenting functions $\mathcal{C}$, the instrumented program $\mathcal{C}_t(p)$ executes all the instructions that p does, interspersed with instrumentation code. We call $\mathcal{C}$ a *faithful instrumenting function* iff

$$\forall t.\ \forall p. \quad p \leq_{\text{erased}} \mathcal{C}_t(p)\ . \tag{4.1}$$

The advantage of erased trace simulation over stronger orderings is that it admits various implementations of the instrumenting function: although faithfulness does limit the choice of instrumenting function because of the intensional nature of $\leq_{\text{erased}}$, we argue that the restrictions are reasonable. For example, consider naive interpretation, i.e. $\mathcal{C}_t(p) = [\![mix_{\text{triv}}]\!](\mathcal{I}(t),\, p)$ where mix_{triv} is the trivial specializer for AL. The simplest interpreter implements every instruction in terms of itself: an `ADD` with an `ADD`, a `MOV` with a `MOV`, etc. It is easy to see that when this is the case Equation 4.1 is satisfied. The dispatch mechanism used by the interpreter—a `switch` statement or threading—is not important here. But one could argue that interpretation is not a viable implementation strategy for a variety of reasons: DTrace and many other frameworks use in-place binary patching to substitute instrumented instructions with jumps into the framework or breakpoints.[1] The breakpoint instruction provides rudimentary support for what the virtualization literature calls "trap-and-emulate" execution.

4.3 Breakpoints: AL/BRK

Recall that AL programs run on a finite CISC-style Harvard architecture: instructions and data are stored separately. The code store cannot be read or written to and the data store cannot be executed. In AL/`BRK`, we extend AL with a new mode of execution and four new instructions: `BRK` which invokes a breakpoint handler residing at a well-known address inaccessible by means of a jump; `BRET` used to return from the handler; `UEXEC` (for "user execute") which executes instructions displaced by `BRK`s; and `UREAD` which gives instrumentation code access to the program state. All four instructions are part of the instrumentation mechanism and may not occur in user programs. As we are not considering recursive instrumentation (i.e. "instrumentation of instrumentation code") the breakpoint handler is not re-entrant, and to execute `BRK` while in handler mode is an error that causes abnormal machine termination. Note that `BRK` cannot simply be inserted into the program text before the instruction of interest, as this causes the layout

[1] The breakpoint instruction is usually one byte long (e.g.: `0xCC` on the x86) so that it can fit into the space occupied by any other instruction.

BRK	$(pc_{\text{user}},\, R_{\text{user}},\, D_{\text{user}},\, C_{\text{user}}) * (0,\, R_{\text{instr}},\, D_{\text{instr}},\, C_{\text{instr}})$
BRET	$(pc_{\text{user}} + 4,\, R_{\text{user}},\, D_{\text{user}},\, C_{\text{user}}) * (pc_{\text{instr}},\, R_{\text{instr}},\, D_{\text{instr}},\, C_{\text{instr}})$
UREAD $r_{\text{dst}},\, x[,\, r_i]$	*match x with* PC $\Longrightarrow s_{\text{user}} * (pc_{\text{instr}} + 4,\, R_{\text{instr}}[r_{\text{dst}} \mapsto pc_{\text{user}}],\, D_{\text{instr}},\, C_{\text{instr}})$ R $\Longrightarrow s_{\text{user}} * (pc_{\text{instr}} + 4,\, R_{\text{instr}}[r_{\text{dst}} \mapsto R_{\text{user}}(R_{\text{instr}}(r_i))],\, D_{\text{instr}},\, C_{\text{instr}})$ . . . ; the cases for D and C are analogous.
UEXEC r_{loc}	Execute an instruction at r_{loc} in user mode.

Figure 4.5: Semantics of BRK, BRET, UREAD and UEXEC.

of code in memory to change, upsetting computed jumps. (And, in general, the problem of determining the target of a computed jump is undecidable.) One might imagine an implementation strategy along the following lines: prior to the program run, each potential instrumentation site is padded with NOPs which are later replaced with BRKs or calls into the instrumentation framework when necessary. However, the NOPs would constitute unacceptable overhead as they are executed even when instrumentation is off.

4.3.1 Semantics

To support BRK, a secondary program counter, register file and code and data memories are necessary in addition to a flag indicating whether the processor is executing in user or breakpoint handler mode. The additional code memory holds the instrumentation code and the data memory holds the variables. The star notation $s_{\text{user}} * s_{\text{instr}}$ borrowed from separation logic [108] signals that the two states are disjoint, and the shading indicates the active (currently executing) state. The table in Figure 4.3.1 shows the semantics of the new instructions. The modifications to the semantics of other instructions are elided. The important point is that the instructions BRET, UREAD and UEXEC above are only accessible in breakpoint handler mode. The UREAD and UEXEC instructions implement reification and reflection of user program state. The UEXEC instruction executes in user mode an instruction held in the data memory of the handler. This setup mimics real-world implementations which copy out the instruction at the instrumentation site into a "scratch" buffer; when instrumentation code is triggered, the overwritten instruction is executed from this buffer.

The new instructions can be thought of as an asymmetric inter-process communication mechanism between the user process and the more privileged breakpoint handler process. The user program state is an implicit argument to the the BRK instruction accessible by reification. We may ask whether the BRK instruction is necessary at all: since AL uses fixed-length encoding for instructions, we could just as easily use a CALL to jump into the instrumentation framework. The immediate problem with this setup is that CALL clobbers the register L; but, more importantly, the modes—much as memory protection hardware—enforce separation between the code and data memories of the program and the instrumentation code. Certainly, if isolation between user code and instrumentation can be proved statically, there is no need to enforce it in the semantics. (This is the approach to memory protection taken in the Singularity [67] operating system—see §4.6.)

4.3.2 A rationale for the design of AL/BRK

Suppose we want to instrument an AL interpreter written in an ML-like language:

```
let eval (pc, r, d, c) =
  match (c pc, ..., c (pc+3)) with
  | (O ADDrrr, W dst, W src1, W src2) ->
      let r' i = if i == dst then (r(src1) + r(src2)) else r(i)
      in eval (pc+4, r', d, c)
```

Consider instrumenting `eval`, and let `brk` be a "callback" function that contains instrumentation code, then the simplest solution is to pass `brk` to `eval` as an argument, and to propagate the accumulated instrumentation statistics:

```
let eval (pc, r, d, c) acc brk =
let acc' = brk (pc, r, d, c) acc in
  match (c pc, ..., c (pc+3)) with
  | (O ADDrrr, W dst, W src1, W src2) ->
      let r' i = if i = dst then (r(src1) + r(src2)) else (r i)
      in eval (pc+4, r', d, c) acc' brk
```

Suppose that we known that `ADD` is the only instruction that will ever be instrumented: i.e. `brk` behaves as the identity function in all other cases. An obvious optimization is to "push" the call to `brk` under the pattern-match:

```
let eval (pc, r, d, c) acc brk =
  match (c pc, ..., c (pc+3)) with
  | (O ADDrrr, W dst, W src1, W src2) ->
      let acc' = brk (pc, r, d, c) acc in
        let r' i = if i = dst then (r(src1) + r(src2)) else (r i)
        in eval (pc+4, r', d, c) acc' brk
```

Now, put `brk` in a tail-call position, passing it the continuation to invoke:

```
let eval (pc, r, d, c) acc brk =
  match (c pc, ..., c (pc+3)) with
  | (O ADDrrr, W dst, W src1, W src2) ->
      brk (pc, r, d, c) acc (fun acc' ->
        let r' i = if i = dst then (r(src1) + r(src2)) else (r i)
        in eval (pc+4, r', d, c) acc' brk )
```

Finally, note that the continuation (the last argument to `brk`) closes over the program state and the operands of `ADD`; there is no need for this since `brk` already receives the program state as an argument. Let `eval'` be the base evaluation function (which does not call `brk`) and perform a transformation similar to closure conversion:

```
let eval (pc, r, d, c) acc brk =
  match (c pc) with
  | O ADDrrr -> brk (pc, r, d, c) acc eval' (fun s acc -> eval s acc brk)
```

Note that `brk` receives the state (cf. `UREAD`), the accumulated instrumentation statistics (cf. data memory D_{instr}), the original non-instrumented evaluation function (cf. `UEXEC`) and the continuation to call.

4.3.3 IL instrumentation with AL/BRK

In this scenario, to instrument a program, every opcode of interest is replaced with the BRK instruction and the original instruction is placed in the instrumentation data memory D_{instr} for later execution with UEXEC. Note the similarity with context threading (see §3.2) where instructions are replaced by calls into the interpreter (cf. BRK) and their operands are stored in a separate table in memory (cf. D_{instr}). The prologue of the breakpoint handler (in C_{instr}) determines which instruction caused the breakpoint by reifying the user program counter with UREAD. The handler then binds *opvars* and transfers control to appropriate rule for that instruction. The implementation of the code primitive should satisfy accesses to instrumented addresses from the scratch buffer containing displaced instructions (in D_{instr}) rather than by UREAD, because otherwise we would always have $\mathsf{code}(\mathsf{pc}) = \mathit{opcode}(\mathtt{BRK})$.

4.3.4 AL/BRK vs. AL/EXEC

Both AL/BRK and AL/EXEC from the previous chapter provide support for non-standard interpretation. AL/EXEC is a basic but faithful model of the trap-and-emulate virtualization assists commonly implemented by hardware. AL/BRK is similar to AL/EXEC in many respects: both allow certain instructions to be intercepted and control to be transferred to emulation or instrumentation code, and both provide mechanisms for isolating the latter from the user program. The most obvious difference between the two languages is that AL/BRK has only a single trapping instruction. Thus instrumentation using AL/BRK also requires a pre-processing step to place breakpoints prior to execution. Further, in AL/EXEC the state of a guest virtual machine is contained in a VMCB (of which there can be many), whereas in AL/BRK the state of the instrumented program (of which there can be only one) is a separate part of a configuration in the operational semantics. A VMM written in AL/EXEC can, in principle, read in code for new guest virtual machines at runtime using AL's port I/O facility.

4.3.5 Efficiency is bounded overhead

The requirement that the instrumented program simulate the original places a *lower bound* on the number of instructions executed by the instrumented program. Efficient instrumentation is also bounded *from above*. Define $dom(t)$, the domain of script t, as the set of opcodes for which there is a rule in t. For example, the domain of the sample script on p. 50 is $dom(t) = \{\mathtt{STOREr r}, \mathtt{JZrr}\}$ where the suffix rr selects a particular version of the opcode. Note that BRK and BRET serve as mode switch indicators in the trace where execution passes from the original code into the instrumentation code and vice versa. Define a predicate h on sequence indices such that $h(tr, i)$ is true iff the instruction at index i in tr is bracketed by a BRK/BRET pair. Using h enables us to define the script-specific relation $\leq_t$ that disregards instrumentation code: i.e. $p \leq_t p'$ means the same as $p \leq p'$ modulo the contribution of t; note that $\leq_t$ does not depend on the instrumenting function used. Recall (§3.4.1) that for a VMM, no requirements are placed on the part of the trace containing emulation code. However, instrumentation code must always execute the original instruction at some point. Let tr and tr' be the traces (not erased) of the instrumented program and non-instrumented computations respectively. For a strictly increasing function f we have

$$tr \preceq_t tr' \iff \exists f.\ \forall i \text{ s.t. } \neg h(tr, i) \text{ or } tr(i+1) = \mathtt{BRET}, \text{ we have } tr(i) = tr'(f(i))\ .$$

Notice that we have assumed that the instrumentation code will `UEXEC` the original instruction immediately prior to returning (`BRET`). We call $\mathcal{C}$ an *efficient instrumenting function* iff $\forall s.\ \forall p.\ \mathcal{C}_t(p) \leq_t p$. Finally, we call $\mathcal{C}$ a *good instrumentation function* iff it is both faithful and efficient:

$$\forall t.\ \forall p. \quad p \leq_{\text{erased}} \mathcal{C}_t(p) \leq_t p\ . \tag{4.2}$$

Notice the universal quantification over IL scripts compared to existential quantification over self-interpreters used in the definition of Jones optimality. Defining good instrumentation in two parts (faithfulness and efficiency) has advantages over straightforward erased trace equality because the efficiency relation can be independently refined. Unfortunately, our definition suffers from the same drawback as the original statement of Jones optimality: it does not allow $\mathcal{C}$ to optimize the program. To satisfy equation 4.2, the trace of instrumentation code must be contained within well-bracketed `BRK`/`BRET` pairs. For example, consider the program "`MOV r0, 10; OUT <3>, r0; HLT`" and the script `OUT <`v_{port}`>`, r_{src} { if reg(0) = 9 then print 1; }. Instrumented, the instruction sequence becomes "`MOV r0, 10; BRK; HLT`". But the then-branch of the rule will never be executed since `r0` always holds 10, and the trace of the instrumented program contains unnecessary handler invocations. This is not necessarily a deficiency since it could be argued that a distinct boundary between program and instrumentation is desirable. We leave a detailed examination of this issue to further work and for now assume that programs are optimized prior to being instrumented (which is often the case in practice).

4.4 From super-instructions to language boundaries

Given IL scripts t_i for $i \in \{1, \ldots, n\}$, every script defines an instrumented language AL(i) with the same syntax as AL. So far we assumed that instrumentation is program-wide. This choice also dictated the semantics of AL/`BRK` where a single additional code memory holds the compiled code of a single instrumentation script. Consider instead allowing different scripts to be applied to different parts of the program at the same time. A program instrumented in this way can reasonably be seen as a "multi-language" program because its behaviour is determined by the combination of several instrumented languages. Consider this example:

procA	procB

`procA: MOV r3, 27`
`       STORE (r8), 94`
`       CALL procB`
	`procB: ADD r3, r3, 10`
	`       STORE (40), r3`
	`       RET`

The handling of the `CALL` instruction may differ depending on the calling convention used: whether `procB` expects to find its argument in register `r3` or on a stack pointed to by `r8`. The instrumentation code must be different in each case if it is to retrieve the value of the argument correctly. For our base language AL, let us call the individual instrumented languages AL(1), AL(2) etc. and the combined language AL($*$). In traditional multi-language interoperability work (e.g. Matthews and Findler [84]) the position of *boundaries* between languages where execution flows from one language to another can be easily determined visually, since the constituent high-level (e.g. Scheme and ML) languages have different syntax. In an instrumentation setup, however, the constituent languages have exactly the same syntax (the syntax of the base language), and so the placement of boundaries is not manifest. The interpretation of a given opcode in the combined language AL($*$) depends both on the opcode itself and the *currently active language*: the two ways of selecting the currently active language are discussed in §4.4.1 below.

4.4.1 Lexical vs. dynamic scoping of boundaries

The boundaries can be scoped *lexically* or *dynamically* by analogy with lexical vs. dynamic variable binding. Define a family of AL(∗) instructions `LEXLANG/`n where n is a number corresponding to one of the constituent language; `LEXLANG/`n specifies that all instructions up to the next lexical occurrence of `LEXLANG` are to be interpreted according to language n. Lexical scoping generalizes the familiar concept of instruction *prefixes* (a prefix modifies the behaviour of the single instruction which it precedes).

Lexical boundaries	**Prefixes**
`LEXLANG 1`	
`MOV r1, r2`	`1/ MOV r1, r2`
`MUL r3, r2, r5`	`1/ MUL r3, r2, r5`
`LEXLANG 2`	
`ADD r3, r1, r2`	`2/ ADD r3, r1, r2`

The mapping between `LEXLANG` and prefixes is an example of a *super-instruction optimization*. A *super-instruction* combines the effects of two or more instructions: e.g. an `ADD/STORE` super-instruction might write the result of an addition to memory[2]. Super-instruction optimization is a well-known technique in the implementation of interpreters and language virtual machines. Alternatively, `LEXLANG` can be thought of as an annotation on the program syntax tree.

Let us define another group of instructions: `DYNLANG/`n. Executing `DYNLANG/`n sets the currently active language so that all instructions up to the next runtime occurrence of `DYNLANG` are interpreted according to language n. Dynamic scoping is comparable to processor *modes* such as 16- vs. 32-bit or the virtual machine mode of the previous chapter (§3.2); indeed, processors with ARM Jazelle extensions are capable of executing both Java bytecode and the native instruction sets. The mode setting affects execution of all instructions from the time the mode is set to when it is changed. Instruction dispatch with `DYNLANG` is analogous to *virtual dispatch* where each derived class provides its own implementation of AL opcodes (`LEXLANG` corresponds to static dispatch):

Instructions	**Dispatch pseudo-code (`I` is global)**
`DYNLANG/1`	`I = get_I(1);`
`MOV r1, r2`	`I.MOV( r1, r2 )`
`MUL r3, r2, r5`	`I.MUL( r3, r2, r5 )`
`ADD r3, r1, r2`	`I.ADD( r3, r1, r2 )`

Indeed, the pseudo-code on the right can be read as a context threading table (§3.2). The negative performance impact of virtual calls is well-known in the literature. A virtual call can be replaced with a static call when the receiver class is known (cf. rewriting `I.MOV` into a prefixed instruction n/ `MOV`). However, note that in languages that support dynamic class loading, a guard must be placed around the optimized static call which checks that the class hierarchy has not changed in a way that invalidates the optimization. For instrumentation, this optimization corresponds to conversion from `DYNLANG` dynamic dispatch to `LEXLANG` static dispatch:

[2]So, very loosely, one might claim that CISC is a super-instruction version of RISC.

A `DYNLANG` basic block	Guarded `LEXLANG` dispatch (`r31` holds `I`)

`I.MOV( r1, r2 )`
`I.MUL( r3, r2, r5 )`
`I.ADD( r3, r1, r2 )`
`I.JMP( 128 )`
	`JE tt, 1, r31`
	. . .; Original code using virtual dispatch
	`tt: 1/MOV r1, r2`
	`    1/MUL r3, r2, r5`
	`    1/ADD r3, r1, r2`
	`    1/JMP 128`

The guard on entry to the basic block (on the right), ensures that the optimization remains sound should the value of `I` change. The static calls can be inlined in the usual way. For instance, if instrumented language "1" does not instrument `MOV`—and therefore behaves as a self-interpreter for `MOV`—then `1/MOV` can be replaced with `MOV`.

4.4.2 Multi-IL

In Multi-IL we allow the scoping of individual instrumentation rules to be defined by extending IL patterns with an annotation which specifies the range of instructions to which the pattern applies: *range* ::= *address* [⟨*count*⟩] and *pat′* ::= *pat* [@ *range*⋆]. For example, "20⟨5⟩" specifies a range of five instructions starting at address 20; a range with a missing count includes just the instruction at the start address. A pattern with a missing range applies to the entire program. Multi-IL is lexically scoped: which instrumentation rule is triggered by a given instruction is determined by its opcode and the value of the program counter only. Scripts are merged by catenation in the same way as before, but with the added condition that instrumentation ranges for a given opcode must not overlap. This requirement is easy to satisfy by splitting the ranges, as in the following example:

Individual scripts		Combined script	
`MOV` r, v { print reg(r); }	@ $0\langle 3\rangle$	`MOV` r, v { print reg(r); }	@ $0, 8$
`MOV` r, v { $x \leftarrow v$; }	@ 4	`MOV` r, v { print reg(r); $x \leftarrow v$; }	@ 4

The example above is not representative: in general, we assume that there will be many rules with the same range, corresponding to a "set" of instrumentation. One part of the program should not incur additional overhead due to instrumentation applied to another part (e.g. it may be acceptable from a performance point of view to instrument a rarely-used code module in minute detail, but this instrumentation must not negatively affect the "hot path"). The value of the user program counter pc_{user} on entry to the breakpoint handler uniquely identifies the rule to be executed, and therefore Multi-IL can be implemented using AL/`BRK`. However, in this setup the handler performs two dispatch steps, first to select the correct language (determined by the range) and then the rule:

Script			Language
`MOV` r, v	{ print reg(r); }	@ $0\langle 3\rangle$	L1
`STORE` (r), v	{ $x \leftarrow$ data(reg(r)); }	@ $0\langle 3\rangle$	
`MOV` r, v	{ $z \leftarrow v$}	@ $12\langle 2\rangle$	L2

Breakpoint handler prologue (example)

```
     UREAD r0, pc       ; Read the user PC
     GTE   r1, r0, 12   ; Is it greater than or equal to 12?
     JZ    L1, r1       ; If not, one of the first two rules applies
     LTE   r1, r0, 16   ; Is it less than or equal to 16?
     JZ    err, r1      ; If not, the handler was called in error
L2:  ...                ; Dispatch an L2 rule
L1:  ...                ; Dispatch an L1 rule
```

Of course, optimized implementations of the prologue are possible. However, a more natural solution, which also avoids having to hard-code the ranges into the prologue code, is to introduce `LEXLANG`-style prefixed breakpoint instructions (as per §4.4.1).

4.4.3 Multi-AL/`BRK`

Multi-AL/`BRK` is an extension of AL/`BRK` with prefixes for the `BRK` instruction: the semantics of `BRK` selects one of many breakpoint handlers based on the value of the prefix. The handler then selects the appropriate rule using the value of pc_{user}. In an extreme case, every Multi-IL rule can be split into a sequence of rules each of which applies to a single-instruction range: i.e. every instrumented instruction becomes a separate language. Then $n/$`BRK` directly invokes the IL rule for the displaced instruction. Compare this static dispatch to the virtual dispatch which is used in virtualization trap-and-emulate where the CPU must inspect the processor mode (host execution vs. VM execution) in order to select the opcode implementation (native vs. VMM emulation).

4.5 Optimization of dynamically instrumented code

Debugging is well-known to be antagonistic to program optimization. Optimization in the presence of dynamic instrumentation is challenging for much the same reasons: IL scripts can distinguish previously equivalent programs. Even removing a spurious `NOP` instruction may have an effect on the result of an instrumented run. Worse still, the compiler must forego profitable optimization in deference to potential future instrumentation which may never materialize. In the general case, the unoptimized execution state must be preserved or reconstructed (see §2.4.1) both at `UPDATE` points and prior to entering instrumentation code. We are interested in static, common-case optimization of dynamically instrumented programs, exploiting these observations: (*i*) instrumentation can only be applied at `UPDATE` points, (*ii*) instrumentation is applied rarely (i.e. most of the time an `UPDATE` is a no-op), and (*iii*) only a small proportion of instructions are instrumented at any given time. The trade-offs for instrumentation of production systems and debugging are different: rather than attempt to undo the effect of arbitrary program transformations on the execution state (a costly procedure) at runtime, we propose to selectively ban overly invasive instrumentation. As an example, consider the programs and script below:

Program	Optimized
`MOV r9, 3`	—
`MOV r9, 5`	`MOV r9, 5`
`MOV r7, 1`	—
`OUT <1>, r9`	`OUT <1>, r9`

Instrumentation script
`MOV` r, v { print reg(r); }
`OUT` < v >, r { print "r7 holds ", reg(7); }

The optimized program is obtained from the original by eliminating two dead assignments: the first and third `MOV`s. Running either program produces the output "5". But when instrumented this is preceded by "0 3 0 1" in the original program, and "0 0" in the optimized (disregarding port numbers and assuming the machine initializes registers to zero). The basic issue is that the optimizer is semantics-preserving with respect to the standard AL interpretation, but not the non-standard interpretation defined by the script. Various authors have proposed adding instrumentation to the program prior to optimization (see related work below) to overcome this problem. However, our proposed efficiency criterion requires a strict separation between program and instrumentation code.

The user program execution state on entry into the breakpoint handler, and consequently into the instrumentation code proper, is altered as a result of optimization. Because every instruction is a potential candidate for instrumentation, the optimizer cannot perform even the simplest peephole rewrites. A possible workaround is to declare certain instructions—e.g. all arithmetic instructions—as exempt from instrumentation. We note that instrumentation code introduces additional dependencies not present in the program itself (for instance, the rule for `OUT` in the script above makes `r7` "live" in the usual data-flow sense). Therefore, by limiting the dependence of instrumentation code on the execution state of the program, further optimizations become possible. To this end we will ascribe types to IL expressions and commands to capture their observational power (i.e. ability to distinguish execution states) as a partial equivalence relation (PER). The approach closely follows Benton's work [15] on program transformation for imperative languages using PERs to model the context in which a command or expression is executed. A PER is a binary relation that is symmetric and transitive, but, unlike a proper equivalence relation, not necessarily reflexive. Recall (§2.7) that if P and Q are PERs over X and Y respectively, then $(P \times Q)$ and $(P \Rightarrow Q)$ are PERs over $X \times Y$ and $X \to Y$ respectively:

$$
\begin{array}{rcll}
(x,\, y) & (P \times Q) & (x',\, y') & \text{iff} \quad x\ P\ x' \text{ and } y\ Q\ y' \\
f & (P \Rightarrow Q) & g & \text{iff} \quad x\ P\ x' \Longrightarrow (f\ x)\ Q\ (g\ x')\ .
\end{array}
$$

Further, given function Γ (which should be thought of as a typing context) from elements of X to PERs over Y, by a slight abuse of notation we will treat Γ as a relation over $X \to Y$ defined as follows: $f\ (\Gamma)\ g \iff \forall x \in \mathit{dom}(\Gamma).\ f(x)\ \Gamma(x)\ g(x)$.

Recall that $[\![e]\!]^{exp}_{\mathrm{IL}} \in \mathit{State} \times \mathit{VEnv} \to \mathit{Val}$ is the evaluation function for IL expressions. Let P and Q be PERs over $(\mathit{State} \times \mathit{VEnv})$ and Val respectively and define equivalence of expressions like so: $e_1 \sim_{P \Rightarrow Q} e_2 \iff [\![e_1]\!]\ (P \Rightarrow Q)\ [\![e_2]\!]$. We will write $e : P \Rightarrow Q$ as shorthand for $e \in |P \Rightarrow Q|$. Intuitively, the relation P reveals the dependence of the expression on the program state and the IL variable environment; Q describes how the value computed by the expression is going to be used by the surrounding context. Note that every expression e such that $e : \mathit{All}_{\mathit{State} \times \mathit{VEnv}} \Rightarrow \mathit{Id}_{\mathit{Val}}$ necessarily conflates all execution states and variable environments, e.g. 42, $\mathsf{reg}(7) * 0$. Such expressions have the least impact on program optimization because they are extensionally constant. Suppose that, like in the example above, the compiler optimizes away a dead `MOV` to `r7`. In that case, no guarantee can be made about the contents of the target register, since it may have been written to by a preceding `MOV` or `STORE`. The optimization remains sound in the presence of instrumentation expression e so long as e does not depend on the value: this is easily modelled by a register context Γ_R such that $\Gamma_R(\mathtt{r7}) = \mathit{All}$.

Further refinement is possible: for example, if a command treats `r7` as a boolean ("`if reg(7) then` ..."), only the truth (or falsity) of the value needs to be preserved by

optimization. Recall that $[\![c]\!]^{comm}_{\mathrm{IL}} : \mathit{State} \times \mathit{VEnv} \to \mathit{VEnv} \times \mathit{Output}$ is the evaluation function for IL commands. Equivalence of commands is defined similarly to equivalence of expressions, taking care to handle `print`: we consider two commands equivalent only if they produce exactly the same output (i.e. the same values in the same order), and so the PER over *Output* is always taken to be *Id*.

We posit per-program syntactic (or "type system"-like) restrictions on instrumentation code to enforce partial independence from the program state. We plan to explore this direction in further work.

4.5.1 From dynamic instrumentation to software updating

Both dynamic software updating and dynamic binary instrumentation modify a running program. Like instrumentation, a dynamic update can only take place at certain points in the program. Like instrumentation, there is anecdotal evidence that dynamic updates co-exist poorly with (static) program optimization. Further, switching instrumentation on is patently a kind of dynamic update since the non-instrumented code is replaced with instrumented code ("dynamic binary patching"). Whereas with DSU it is clear that the update applies to the object program itself, in the case of instrumentation it is reasonable to argue that it is the *interpreter* that is updated—the program text remains unchanged but the behaviour of some instructions (like `CALL`) is altered. But we can use partial evaluation to derive the updated code from the original code and the updated interpreter: (*i*) annotate every basic block with the instrumentation set that applies to the instructions in the block (so the boundaries between languages correspond to basic block boundaries), (*ii*) specialise instrumented interpreter with respect to the appropriate basic blocks based on the annotations, and finally (*iii*) update the program by replacing the original basic blocks with the instrumented basic blocks.

4.6 Related work

Kishon and Hudak [77] use partial evaluation first to instrument a standard interpreter with the instrumentation actions and then specialize the resulting instrumented interpreter with respect to the program (thereby instrumenting the program statically). The authors use benchmarking to quantify the performance impact of instrumentation.

Instrumentation code often relies on compilation details (e.g. arguments residing at specific locations on the stack, having a valid stack pointer so that a stack trace can be built, etc.) that the compiler must take care to preserve. The problem is less pronounced if, instead of a compiler, a specializer is used together with an instrumented interpreter (as in Kishon and Hudak's work above). In the absence of good specializers for most languages, a practical alternative is to insert instrumentation code into the program before compilation (see §2.6). Doing so also means that the instrumented program benefits form the optimizations that the compiler might be able to perform. This approach is taken by Tolmach and Appel [132], Wallach and Felten [143] and Allwood et al. [8] among others.

Siskind and Pearlmutter's [122] `map-closure` construct which allows identifiers in the environment of a closure to be rebound, thus enabling a kind of non-standard interpretation. The relationship of `map-closure` to non-standard interpretation techniques developed by the partial evaluation community is, to our knowledge, unexplored.

DTrace was originally developed for Solaris by Cantrill et al. [22], and has since been ported to FreeBSD and Mac OS X. VMware's VProbes [136] framework is similar in

many respects. As far we are aware, neither system has a formal semantics. Cantrill et al. [22] note in passing the similarity between aspect weaving and instrumentation. Indeed, the "Hello, world" example of aspect-oriented programming (the logging aspect) is a form of program instrumentation. However, the payload of the aspect is written in the same general-purpose programming language as the rest of the program. Unlike aspects, instrumentation is not intended as a generic execution augmentation mechanism; most conspicuously, instrumentation code is "passive" and must not modify the program state. These implied restrictions are made explicit in special-purpose instrumentation languages like IL.

Tamches and Miller [129] describe KernInst, a dynamic instrumentation framework for Solaris on UltraSPARC, and one of the first of its kind. The paper includes a detailed quantitative analysis of the overheads of KernInst. In their system, an instrumented instruction is replaced with a jump to a "code patch" which contains the instrumentation code followed by the displaced instruction and a jump back to the original code path. KernInst perform a liveness analysis on the kernel binary to find registers that are dead at the instrumentation point; dead registers can be used by instrumentation code without needing to be saved and later restored. In AL/`BRK` the instrumentation and user states are architecturally distinct: intuitively, the save/restore functionality is implemented in hardware. However, it is interesting to note that translating AL/`BRK` to plain AL (so that `BRK` becomes `JMP`) would require a similar analysis to be performed. This is an expression of an extremely general idea familiar from type checking, that runtime checks (e.g., checking that instrumentation code does not write to registers or data memory belonging to the user program) can often be replaced by static verification. For example, the Singularity operating system [67] introduced the concept of a *Software Isolated Process (SIP)*. Static verification guarantees that SIPs occupying the same address space will not interfere with one another. This means that memory protection hardware can be "switched off".

Typed assembly languages were popularised by Morrisett et al. [91]. Necula [95] introduces proof-carrying code and, with Lee, certifying compilation [96]. Shao [118] presents a common language, based on F_ω, explicitly as a substrate for language integration. Rudiak-Gould et al. [111] propose an intermediate language to study interaction between CBV (ML-like) and CBN (Haskell-like) computation. Trifonov and Shao [133] describe $\mathcal{R}$, an intermediate language also designed to support interoperability between HOT languages. In $\mathcal{R}$, expressions (e) require machine *resources* (ρ) to produce *effects* (ε). The typing judgement has the form $\rho; \Delta; \Gamma \vdash_e e : \tau; \varepsilon$. Resources modularise the computational models of the languages: some use stack-allocated activation records—others heap-allocated, some have a mutable store—others do not. (For example, the "mutable store" resource is required for an expression to write to the store.) The authors give the language an operational semantics, but note in concluding that a monadic semantics would also be appropriate in view of the known similarities between effects and monads [140]. It would be interesting to see whether a similar setup can be used to give semantics to a family of instrumented languages, all compiled to the same monadic base language.

4.7 Conclusions and further work

This chapter has started to apply programming language theory to an emerging class of instrumentation tools—like DTrace and VProbes—where the instrumentation code is written in a domain-specific language. We have specifically focused on performance guar-

antees, proposing a definition for efficient instrumentation adapted from a well-known virtualization efficiency criterion. We do not claim that our proposed criterion is applicable to all instrumentation frameworks and under all circumstances. Rather, it is a first attempt to capture a property that has hitherto received little attention in the literature. In the latter part of the chapter, we used partial equivalence relations to make explicit the dependence of instrumentation code on intermediate program execution states—the finer the relation, the greater the extent to which the script impedes optimization.

4.7.1 Further work

Certified instrumentation code together with a verified framework would allow the consumer to check both the performance and security impact of untrusted third-party instrumentation. This is important because malicious or incompetent instrumentation code can compromise security by—directly or indirectly—leaking values of secret variables. DTrace has a very coarse-grained capability system governing *what* a given UNIX user can instrument rather than properties of the instrumentation code. In addition to performance guarantees, we believe that pervasive instrumentation of production systems will require a finer, language-based permissions model. We speculate that robust declassification [152]—an idea developed in information-flow security—may prove to be a useful concept in limiting the security impact of instrumentation and debugging code. Zdancewic and Myers [152] require that an active attacker not be able to trick the program into revealing any more information than it already does by declassification. Myers et al. [93] model active attack by allowing the attacker to fill holes (cf. instrumentation sites) in the original program with his/her own code. The language used for the these hostile code fragments is a subset of the language the program itself is written in.

Chapter 5

Dynamic typing, boundaries and con-freeness

A *dynamically-updatable program* is a program whose code and data type definitions can be changed at runtime. Every so often a dynamically-updatable program reaches an *update point*, usually by virtue of executing a special "update" command. If no update is pending, the program continues as if nothing happened (the "`update` $\equiv$ `skip`" case). However, when an update *is* available, a decision has to be taken on whether or not it can be applied safely and, if so, exactly *how* it should be applied: the updated program contains a mixture of "old" and "new" code and type definitions. The problem of keeping dynamically-updatable programs type-safe in spite of dynamic updates has received considerable attention in recent years [65, 126, 94]. Unfortunately, the mechanics of applying an update, along with definitional, syntactic safety checks like con-freeness (described below), have been built directly into the semantics of current dynamic software updating (DSU) calculi. A progress-and-preservation soundness result follows but does not reveal what *semantic* properties hold at a "safe" update point. As a consequence of this, to our knowledge there are no DSU calculi amenable to proofs of program equivalence of the kind necessary to enable correct optimizing compilation. For instance, consider these two commands:

$$(\texttt{update; x:=f(0)+42}) \stackrel{?}{=} (\texttt{update; x:=42})\ .$$

The commands are equivalent as long as (i) `f(0)` always returns zero (e.g. `f` could be the identity function) and (ii) we promise never to apply an update containing an `f` that violates (i). In the previous chapter we described how the possibility of future instrumentation impedes static compiler optimizations. Similarly, in DSU there is a clear trade-off here between optimizability vs. updatability. We argue that to avoid a priori restrictions on optimization, the semantics of a DSU language should remain standard and largely orthogonal to the mechanism by which updates are separated into "good" and "bad", and applied. We will focus on Stoyle et al.'s approach [126] to safety of updates in particular as it is both unusual (compared with a traditional type system) and quite well-known. A brief summary follows (see §5.1 for details). Take the state of a program at an update point together with a pending update. Their combination (which includes code, data and type definitions and so is, in effect, a continuation) is con-t-free if old code never relies concretely (by e.g. accessing fields of a labelled record) on the definition of data type t. It is then safe to change the definition of t at a con-t-free point: the semantics of Proteus (Stoyle et al.'s DSU calculus) rejects an update if it fails a runtime *con-free check*.

Contributions & outline. Throughout this chapter, we will use a language called HL, a simple imperative language with an `update` command. Borrowing from Matthews and Findler's work on multi-language interoperability [84], we split the semantics of HL into separate evaluation functions for old and new code (§5.2). To accept a pending update, a HL program *terminates* exposing its entire intermediate execution state: this allows us to keep the language semantics largely free of the details of updating. Once an update is applied, the program is restarted. HL supports definition of named types and a restricted form of runtime type analysis. (The definition of a named type does not change in the course of normal execution, but only due to an update.) Every named type is interpreted as a universal type analogous to Dynamic [2]; runtime type analysis allows code that uses named types to remain well-typed following an update. We identify an extensional ("semantic") counterpart to con-freeness of which the original is an over-approximation (§5.4). Doing so makes explicit the relationship between con-freeness and familiar dependency-based program properties: liveness and secrecy. We show that there are no runtime execution monitors that are complete with respect to semantics con-freeness. In place of the con-free check, we define a type system, closely based on Volpano et al.'s typing rules for secure information flow (§5.5). We argue that $\mathbf{con}_t$ and $\mathbf{abs}_t$ contribute to interpretational overhead and draw a connection to the tag-elimination problem encountered in specializing interpreters for strongly-typed languages (§5.6). Finally, con-t-freeness can be seen dually as either a restriction on runtime states (i.e. only those where old code does not rely on t) or as a restriction on *updates* (i.e. those that do not change the definition of t). We sketch an approach to program optimization of dynamically-updatable programs (a problem not addressed in existing DSU calculi) based on restricting updates that can be applied at a given point (§5.7). Equivalence of dynamically-updatable programs is a bisimulation in a labelled transition system where states are programs and transitions are dynamic updates. Restrictions that are too onerous can be relaxed using a variant of dynamic deoptimization, a technique due to Hölzle et al. [66] for source-level debugging of optimized programs. Conclusions (§5.9) follow a survey of related work (§5.8).

5.1 Preliminaries: con-freeness

The PROTEUS calculus of Stoyle et al. allows named data types to be defined: a definition of the form $\mathbf{type}\ t = \tau$ introduces a named type t whose *representation type* is τ. Such definitions are generative, i.e. t and τ are not type synonyms and values of the two types cannot be used interchangeably. To this end, the coercions $\mathbf{abs}_t : \tau \to t$ and $\mathbf{con}_t : t \to \tau$ are provided for each named type t. An expression e of type t is wrapped with a coercion $\mathbf{con}_t$ if the value of e is used "concretely", i.e. if the use depends on the representation type of t. For example, suppose a variable x has type pair which is a type of labelled tuples with fields "fst" and "snd", i.e. $\mathbf{type}\ \mathsf{pair} = \{\text{fst} : \mathsf{int},\ \text{snd} : \mathsf{int}\}$. By projecting out a field, the expression $(\mathbf{con}_{\mathsf{pair}}\ x).\text{fst}$ relies on the underlying representation type of x, therefore x must be wrapped with $\mathbf{con}_{\mathsf{pair}}$. Expressions of type τ are injected into t by $\mathbf{abs}_t$; for example, $\mathbf{abs}_{\mathsf{pair}}\ \{\text{fst} : 20,\ \text{snd} : 9\}$ has type pair. Note that, via its argument, $\mathbf{abs}_t$ implicitly depends on the representation type of t

A point in the execution of a program is *con-t-free* with respect to an update if no old code that may execute from that point onwards contains occurrences of $\mathbf{con}_t$; "con-free" is shorthand for "con-t-free for all named types t". To preserve type safety, updates to the definition of t are only permitted at con-t-free points. Occurrences in old code of $\mathbf{abs}_t$ for updated types t are wrapped in a user-supplied "typed transformer function".

$$
\begin{aligned}
\textit{prog} &::= \textit{tdef}^* \ \textit{vdef}^* \ \textit{fdef}^* \ \textit{comm}; \ [\texttt{out}\ \textit{var}^+] \qquad \textit{tnam}, \textit{var}, \textit{fun} \text{ drawn from countable sets of names.}\\
\textit{tdef} &::= \texttt{type}\ \textit{tnam} = \textit{typ}\ (\texttt{and}\ \textit{tnam} = \textit{typ})^*\\
\textit{val}' &::= \textit{val} \mid \texttt{mk-}\textit{tnam}\ \textit{val}'\\
\textit{vdef} &::= \texttt{var}\ \textit{typ}\ \textit{var} := \textit{val}'\\
\textit{fdef} &::= \texttt{fun}\ \textit{tnam}\ \textit{fun}(\textit{tnam}\ \textit{var})\ \{\ \textit{vdef}^*;\ \textit{comm};\ \texttt{return}\ \textit{exp};\ \}\\
\textit{typ} &::= \texttt{int} \mid \texttt{sum}(\textit{typ}, \textit{typ}) \mid \texttt{prod}(\textit{typ}, \textit{typ}) \mid \textit{tnam}\\
\textit{scheme} &::= * \mid \texttt{int} \mid \texttt{sum}(\textit{scheme}, \textit{scheme}) \mid \texttt{prod}(\textit{scheme}, \textit{scheme}) \mid \textit{tnam}\\
\textit{exp} &::= \textit{val} \mid \textit{var} \mid \textit{op}\ \textit{exp}^+ \mid \texttt{fst/snd}\ \textit{exp} \mid \texttt{inl/inr}\ \textit{exp} \mid (\textit{exp}, \textit{exp}) \mid \texttt{mk-}\textit{tnam}\ \textit{exp}\\
\textit{comm} &::= \textit{comm}; \textit{comm} \mid \texttt{while}\ \textit{exp}\ \texttt{do}\ \textit{comm} \mid \texttt{skip} \mid \textit{var} := \textit{exp} \mid \textit{var} := \textit{fun}(\textit{exp})\\
&\mid\ \texttt{if}\ \textit{exp}\ \texttt{then}\ \textit{comm}\ \texttt{else}\ \textit{comm} \mid \texttt{crash} \mid \texttt{update}\ n\\
&\mid\ \texttt{case}\ \textit{exp}\ \texttt{of}\ \{\ \texttt{inl}\ \textit{var} \Rightarrow \textit{comm};\ \texttt{inr}\ \textit{var} \Rightarrow \textit{comm}\ \}\\
&\mid\ \texttt{reprcase}\ \textit{exp}\ \texttt{of}\ \{\ (\textit{scheme}\ \textit{var} \Rightarrow \textit{comm})^+\ \}\ [\texttt{else}\ \textit{comm}]
\end{aligned}
$$

Further syntactic restrictions: (i) names of global variables, functions and types must be unique and (ii) global variables must have named types.

Figure 5.1: Syntax of HL.

(The authors use an operational semantics, so the program continuation is explicit.) The rationale behind this definition is intuitively appealing: old code, written without any knowledge about future updates, makes assumptions about the definitions of types which may become invalid. For example, consider the following code:

$$
\begin{aligned}
&\textbf{type}\ \mathsf{t} = \mathsf{int}\ \textbf{in}\\
&\textbf{fun}\ \mathsf{inc}(x : \mathsf{t}) : \mathsf{int} = (\textbf{con}_{\mathsf{t}}\ x) + 1\ \textbf{in}\\
&\textbf{let}\ x = \textbf{update}\ \textbf{in}\ \ldots
\end{aligned}
$$

Any dynamic update which changes the representation of `t` but not the definition of inc will fail the con-free check at **update**. However, the syntactic con-free check, which is part of the operational semantics of Proteus, is definitional: there is no independent statement of what it means for an update point to be con-free other than passing the check. The check applies to old code only: new code may use both old and new named types freely. The program is, in effect, partitioned into two "modules": old code with old type definitions and new code with both old and new type definitions.

Note: Stoyle et al. also define a static *updatability analysis* that locates con-free *program points* (as opposed to runtime states). Whenever we refer to con-freeness, we mean the former notion defined by the runtime con-free check.

5.2 HL

HL is a simple imperative language with support for dynamic updates and runtime type analysis. The syntax of HL is shown in Figure 5.1. A program consists of a series of type, variable and function definitions. Functions are single-entry single-exit: `return` is always the last command in a function body. (Note that a function call is a command rather than an expression; expression evaluation always terminates.) There is a single base type (integers) and type constructors for binary sums and products. HL's booleans are C-like, a special case of integers: zero is falsity, any other integer is truth. As in Proteus, types can be given a name: a definition like `type` $t = \tau$ introduces a new named type t

with type τ as the underlying representation type. The expression (mk-t e), analogous to ($\mathbf{abs}_t$ e), makes a new value of the named type out of a value of its representation type. The `reprcase` command, which is more general that $\mathbf{con}_t$, eliminates values of named types; `reprcase` is described below. Named types may occur on the right hand side of a named type definition, as in:

```
type t1 = sum(int, int) and t2 = prod(int, t1)
```

Note that this is not merely a notational convenience: `t2` as defined above is not equivalent to `prod(int, sum(int, int))`. Type equality is nominal rather than structural. Recursive definition of named types is allowed.

5.2.1 Runtime type analysis

The `reprcase` command allows the representation type of a named type to be scrutinised. The representation type of a value is matched against one or more "type schemes", types with possible wild-cards; `reprcase` executes the earliest branch where a match is found, binding a variable to a value of the representation type. For example,

```
type t1 = int and t2 = prod(int, int);
fun t2 f(t1 x) {
    var t2 z:=mk-t2(0,0)
    reprcase x of {
        int y => z:=mk-t2(y,y);
    }
    return z;
}
```

Each named typed t is modelled as a separate universal type like Dynamic (see §2.5.1). Over the course of the program run, the representation of the type may be altered by updates; using `reprcase`, these changes can be handled gracefully by existing code. Indeed, in dynamically-typed languages, it is standard practice for a function to accept arguments of different types: `read` might accept a file name (string) or file descriptor (integer). HL's `reprcase` is more powerful than a $\mathbf{con}_t$ coercion but simpler than Abadi et al.'s [2] **typecase**: the latter can bind type variables, whereas we use a single wild-card symbol. Assuming we have an expression e of type t with representation τ, $\mathbf{con}_t$ e can be approximated as `reprcase` e `of {` τ `_ => skip; * _ => crash; }`. Intuitively, a program that passes the syntactic con-free check is guaranteed not to crash. Note that $\mathbf{con}_t$ is part of the internal syntax of PROTEUS, has a very specific purpose and cannot be used for general runtime type analysis. Syntactic con-t-freeness in PROTEUS specifically checks for the presence of $\mathbf{con}_t$ coercions in old code, whereas semantic con-freeness (§5.4) does not treat `reprcase` specially. Finally, of the two constructs ($\mathbf{con}_t$ and **typecase**), **typecase** is the better-known [2, 62].

5.2.2 Dynamic semantics

The small-step operational semantics of HL is shown in Figure 5.4. The pre-domain of values contains enough elements to interpret the types of HL:

$$Val \cong \mathbb{Z} + (Val + Val) + (Val \times Val) + \{tyerr\} .$$

If an operator is applied to arguments of the wrong type, the result is the special value *tyerr* which belongs to no type. The program as a whole does not become stuck: a type error which does not contribute to the final result of the program is semantically harmless (even though many type systems will reject programs which might encounter such type errors). Similarly, `fst` e and `snd` e evaluate to *tyerr* if e evaluates to anything other than a product; the `case` and `reprcase` commands do nothing when applied to a value of the wrong type (sum for `case` and named type for `reprcase`); `if` and `while` treat any value other than zero, including non-integer values, as truth.

When referring to a HL "program", we mean a combination of command, state and function and type environments. The mutable state $s \in \mathit{State} = \mathit{Mem} \times \mathit{Stack}$ of a running program consists of a global memory $\mathit{Mem} \stackrel{\text{def}}{=} \mathit{Var} \rightharpoonup (\mathit{Val} \times \mathit{Typ})$ and a stack $\mathit{Stack} \stackrel{\text{def}}{=} \mathbb{N} \rightharpoonup \mathit{Mem}$. Every variable carries a tag identifying its type. The stack is a list of activation frames, each containing bindings for local variables, with the bottom-most (currently active) frame at the head. We let m_g and m_ℓ ("global" and "local") range over *Mem* and k over *Stack*.

In addition to the state, the evaluation function receives the command to execute and two environments—the *function environment* and the *named type environment* map between the names and definitions of functions and named types respectively—and returns an answer in $\Omega_\perp$ (see below):

$$[\![c]\!]^{\mathit{comm}}_{\mathrm{HL}} : \mathit{State} \to \mathit{FunEnv} \to \mathit{TypEnv} \to \Omega_\perp \ .$$

Both `crash` and `update` terminate the program. By analogy with debugger breakpoints, and as in the previous chapters' language AL, `update` preserves the entire intermediate execution state; `crash` only signals that the program terminated abnormally and can be thought of as putting the program into a "stuck state". For normal termination, the notation `out` $x_1, \ldots, x_n$ indicates which global variables are live at the end of the program. (Other variables are "dead" and their final values are undefined.) The predomain of answers is defined as follows:

$$\Omega \cong \mathit{Mem} + \mathbf{1} + \mathbf{U} \quad \text{where } \mathbf{U} \stackrel{\text{def}}{=} (\mathbb{N} \times \mathit{comm} \times \mathit{State} \times \mathit{TypEnv} \times \mathit{FunEnv}) \ .$$

HL programs can thus exhibit three possible behaviours apart from non-termination:

Termination mode	**Cause**	**Summand of Ω**
Normal	—	*Mem*
Crash	`crash`	$\mathbf{1}$
Update point	`update`	$\mathbf{U}$

The numeric argument of the `update` command identifies the program point corresponding to the update point; `update` commands should therefore be uniquely numbered, but this is not a formal requirement. We call an element of $\mathbf{U}$ a *reified configuration* and write it as $\mathbf{U}_n\langle c,\ (m_g,\ k),\ T,\ F\rangle$. The *comm* component of a reified configuration is the continuation of the program after the update point. Note that a named type environment is sufficient to determine which branch will be taken by a `reprcase`. Since the type environment changes infrequently—ordinary execution between updates can only modify the memory and stack—we would like to optimize away the dead branches of `reprcase` statements, in effect specializing the program with respect to its type environment. We will come back to this observation in §5.6, but first we describe dynamic updates in HL and how they are applied.

5.3 Dynamic updates

Our goal is to understand dynamic updating as "analysis + transformation" of programs in *as standard a semantics as possible*: in existing DSU calculi, including PROTEUS, the mechanics of updating are built into the definition of the language. In HL, an update is applied by modifying the intermediate state of a program that is exposed when an update point is reached; execution then continues from the new, updated state. We believe this design offers several advantages over a "hard-coded" semantics for **update**. First, keeping update policy separate from the language makes it possible to compare different DSU proposals (the language is a controlled variable); second, the essential content of analyses like the con-free check is easier to grasp when these are formulated over a standard semantics; finally, an optimizing compiler written for the standard semantics is guaranteed to work with all possible current and future update policies.

For the purposes of this chapter we make the following basic assumptions about updates which are in line with prior work: (*i*) an update can add, change and remove definitions of named types, global variables and functions, (*ii*) an update to a function definition takes effect when the function is next invoked (a function active at the time of the update continues to execute its old code), and (*iii*) a pending update is applied when the program reaches an update point by executing `update` (when no update is pending, `update` is a no-op). Note that the second requirement is easily satisfied by an implementation that copies ("clones") function bodies immediately on call. Alternatively, the implementation may also choose to copy the definition if and when it is updated. The trade-offs mirror those of implementing a `fork` system call: the memory of the parent process can be either copied immediately on `fork` (wasteful if `fork` is followed immediately by `exec`) or lazily by Copy-on-Write.

5.3.1 Applying an update

Applying an update is a meta-level operation. Suppose we have a program with function environment F_{old} and type environment T_{old} that has reached an update point and terminated with the result (i, c, s). If no update is pending, the program is restarted with the old function and type environments: $[\![c]\!]\ s\ F_{\text{old}}\ T_{\text{old}}$ (`update` is then equivalent to `skip`). Suppose that an update is pending and that the update defines (or redefines) the functions in F_{new} and the named types in T_{new}. Define the operator $\uplus$ which combines and old function or type environment with a new one:

$$(f_{\text{new}} \uplus g_{\text{old}})(x) = \begin{cases} \mathit{inl}\ f_{\text{new}}(x) & x \in \mathit{dom}\ f_{\text{new}} \\ \mathit{inr}\ g_{\text{old}}(x) & \text{otherwise}\ . \end{cases}$$

By slight abuse of notation we shall disregard the constructors *inl* and *inr* in most cases.

Recall that global variable types and function argument and return types must all be named. To change the type of a global variable, the underlying representation is changed. Similarly, if, say, function `f` now takes two arguments instead of one, its argument type `t` is changed accordingly: from (e.g.) `type t = int` to `type t = prod(int, int)`.

Variables in memory and in frames on the stack which have a type in T_{new} must be converted to the new representation. Let $\mathcal{U}_{\mathit{State}} : \mathit{State} \to \mathit{State}$ be the conversion function that replaces all values of types in T_{new} in memory and in the stack frames with the corresponding values in the new representation[1].

[1] $\mathcal{U}_{\mathit{State}}$ must perform a deep conversion.

Though existing values of update named types are converted to the new representation by $\mathcal{U}_{State}$, old code that constructs values of an update type is now ill-typed. This is because it still attempts to mk-t values of each updated named type t using the old representation of t. Stoyle et al.'s original approach is to replace occurrences of $(\mathbf{abs}_t\ e)$ in old code, wrapping the expression e with a user-provided function that maps values from the old representation of the type t to the new. We follow a similar approach: to keep old code well-typed, every (mk-t e) where e is in the old representation of t is replaced with (mk-t e') such that e' is in the new representation. The conversion from e to e' is necessarily simpler than Stoyle et al.'s since a function call in HL cannot be part of an expression. Let $\mathcal{U}_{comm}$ and $\mathcal{U}_{FunEnv}$ be the corresponding conversion functions for commands and function environments respectively. The function environment and the type environment of the new, updated program are, respectively, $(F_{\text{new}} \uplus (\mathcal{U}_{FunEnv}\ F_{\text{old}}))$ and $(T_{\text{new}} \uplus T_{\text{old}})$. The result of the updated program is

$$[\![\mathcal{U}_{comm}\ c]\!]\ (\mathcal{U}_{State}\ s)\ (F_{\text{new}} \uplus (\mathcal{U}_{FunEnv}\ F_{\text{old}}))\ (T_{\text{new}} \uplus T_{\text{old}})\ .$$

5.3.2 HL language boundaries

The Proteus con-free check (described in §5.1) treats old and new code in a program differently. The new code is privileged, in the sense that it can use values of any of the updated named types concretely. In HL, we make the boundaries between old and new code explicit: as will become clear in the next section, the distinction between old and new code is semantically important. We split the valuation function $[\![\cdot]\!]_{\text{HL}}$ into two mutually-recursive functions $[\![\cdot]\!]^{\text{old}}_{\text{HL}}$ and $[\![\cdot]\!]^{\text{new}}_{\text{HL}}$ for old and new code respectively. This setup is intended to resemble Matthews and Findler's "Operational Semantics for Multi-Language Programs" [84]. Note that after an update execution always starts in the old code immediately following the `update` command: in Matthews and Findler's terminology, the "top-level context" belongs to the old-code language.

5.4 Semantic con-freeness

The relationship between a semantic notion of con-freeness and the original syntactic definition is best illustrated by analogy with semantic versus syntactic *liveness* of variables. In the λ-calculus, recall that a variable x is semantically dead in expression e and environment $\rho \in Var \rightharpoonup Val$ if its value has no bearing on the result of evaluating e in ρ:

$$\forall v, v' \in V.\ [\![e]\!]\ \rho[x \mapsto v] = [\![e]\!]\ \rho[x \mapsto v']\ . \tag{5.1}$$

On the other hand, x is syntactically dead in e if there are no program paths in e that contain a use of x that is not dominated by a definition. Syntactic liveness is a computable (dataflow) over-approximation of semantic liveness. Semantic liveness is conveniently expressed using partial equivalence relations, binary relations that are symmetric and transitive but, unlike proper equivalence relations, not necessarily reflexive (see §2.7). For a variable x, let $[\![e]\!]_x\ \rho = \lambda v.\ [\![e]\!]\ \rho[x \mapsto v]$; then x is dead iff $([\![e]\!]_x\ \rho) : All_{Val} \Rightarrow Id_{\Omega_\perp}$ where $\Omega_\perp$ is the domain of results. This can be extended straightforwardly to a set of variables $X \subseteq Var$. Let $\Phi \in Var \rightharpoonup Per(V)$ map a variable to a PER over values; Φ can be used to define a relation over environments: $\rho\ \Phi\ \rho' \iff \forall x \in dom\ \Phi.\ \rho(x)\ \Phi(x)\ \rho'(x)$. Now, define a Φ as follows:

$$\Phi \stackrel{\text{def}}{=} x \mapsto \begin{cases} All_V, & x \in X \\ Id_V, & \text{otherwise}\ . \end{cases}$$

The variables in X are semantically dead iff $[\![e]\!] : \Phi \Rightarrow Id_{\Omega_\perp}$.

5.4.1 Definition of semantic con-freeness

Let $\Phi : \mathit{TNam} \rightharpoonup \mathit{Per}(\mathit{Typ})$ map the name of a type to a PER over types. This is also a PER: two type environments $T, T' \in \mathit{TypEnv}$ are related by Φ when they are pointwise-related:

$$T \; \Phi \; T' \iff \forall t \in \mathit{dom}\ \Phi.\ T(t)\ \Phi(t)\ T'(t)\ . \tag{5.2}$$

New code can examine the representation of both new and old named types freely, so let us introduce another argument to the evaluation functions giving

$$[\![c]\!]^{\text{new}}_{\text{HL}},\ [\![c]\!]^{\text{old}}_{\text{HL}} : \mathit{State} \to \mathit{FunEnv} \to \mathit{TypEnv} \to \mathit{TypEnv} \to \Omega_\perp\ . \tag{5.3}$$

The first and second type environments are used by $[\![c]\!]^{\text{new}}_{\text{HL}}$ (new code evaluation function) and $[\![c]\!]^{\text{old}}_{\text{HL}}$ (old code evaluation function) respectively to look up the representation of a named type. A command c is *semantically con-free* for named types in T_{new} iff

$$E[\![\mathcal{U}_{comm}\ c]\!]\ (\mathcal{U}_{State}\ s)\ (F_{\text{new}} \uplus (\mathcal{U}_{FunEnv}\ F_{\text{old}}))\ (T_{\text{new}} \uplus T_{\text{old}})\ : \Phi \Rightarrow Id_{\Omega_\perp} \tag{5.4}$$

where

$$\Phi(t) = \begin{cases} All_{\mathit{Typ}} & t \in \mathit{dom}\ T_{\text{new}} \\ Eq_{T_{\text{old}}(t)} & \text{otherwise}\ . \end{cases}$$

Varying the representations of the new named types in old code (while keeping those of old ones the same) has no effect on the final result of a con-free program. Note that, although old code cannot inspect the representation of a new named type directly, it *can* do so indirectly by calling out to new code instead.

5.4.2 Runtime enforcement

Gray et al. [59] and later Matthews and Findler [84] used contracts to control value flow across the boundary between a statically-typed and a dynamically-typed language. Consider Matthews and Findler's notation ($\mathbf{e_1}$ ($^{\tau}MS$ $\mathtt{e}_2$)) where an ML expression $\mathbf{e_1}$ is applied to the result of evaluating a Scheme expression $\mathtt{e}_2$. We have an outwardly similar notion of a "mixed" program where boundaries are drawn between old and new code rather than code written in different languages, and a natural question to ask is whether semantic con-freeness can be checked using contracts. The contract we want to enforce is that old code should not depend on new named type definitions.

Since we are in an imperative setting, we consider predicates on program states. Intuitively, contracts are a form of *execution monitoring*: the decision about whether or not a violation has occurred is taken solely based on the history of execution so far, i.e. a finite prefix of the program trace. Neither execution monitors, nor contracts have a way of divining future possible executions. Volpano [137] showed that there are no sound and complete execution monitors for secrecy, and we follow a similar argument for semantic con-freeness. The key point is that observations are made on the *final result* of the program. To illustrate, consider a command `z:=f(x)` where `z` is a global variable of type `t2` and `f` is a function defined as follows:

```
fun t2 f(t1 x) {
    var t2 r:=mk-t2 0;
```

```
        update 0;
        reprcase x of {
            prod(int, int) _ => r:=mk-t2 1;
        }
        return r;
    }
```

Suppose type `t1` was defined as `prod(int, int)` to begin with, but we now want to redefine it as `int` at the update point. Syntactic con-freeness forbids the update at update point 0 because of the presence in the continuation of a `reprcase` on `x` (which has type `t1`). However, if the variable `z`, which gets the return value of `f`, is *semantically dead*, then the change in the representation of `t1` at the update point has no effect on the result of the program. In other words, the program after the update is *semantically* con-free but not *syntactically* con-free.

Consider implementing an execution monitor for semantic con-freeness. The set of traces of an execution monitor must be prefix-closed. Thus, a monitor which disallows the assignment `z:=f(x)` is incomplete (because `z` might be dead), and, conversely, a monitor which allows the assignment is unsound (e.g. `z:=f(x)` is the last statement in the program). Therefore we conclude that *no runtime monitor is sound and complete with respect to semantic con-freeness.*

5.5 The con-free check as a type system

In the previous section, we noted that con-freeness, like semantic liveness and non-interference, is a dependency property. There are many type systems in the literature that enforce secrecy, i.e. are sound with respect to non-interference: a well-typed program is guaranteed not to leak information about the values of high-security variables. In this section we introduce a type system intended to generalize the con-free check, and whose rules are based closely on Volpano et al.'s [137] type system for secure information flow.

A typing context Γ gives variables, functions and named types a *security level*, which is either "low" or "high" for variables and named types, or an arrow between security levels for functions:

$$\Gamma ::= x : \ell, \Gamma \mid f : \ell \to \ell', \Gamma \mid t : \ell, \Gamma \mid \varepsilon \qquad \ell \in \{L, H\} .$$

The judgment $\Gamma \vdash x : \ell$ means that $\Gamma(x) = \ell$ and similarly for functions and named types. Judgement for expressions and commands have the form $\Gamma \vdash e : \ell \ exp$ and $\Gamma \vdash c : \ell \ comm$ respectively. Informally, the judgement $\Gamma \vdash e : L \ exp$ means that e does not access any high-security information. Conversely, the judgement $\Gamma \vdash c : H \ comm$ means that c does not *assign* to any low-security variables (and therefore can be trusted with high-security information). This simple formulation relies on expressions being side-effect free: in HL, a function call is not an expression (see grammar on p. 67).

For example, if `x` is high-security variable, the command "`if x then z:=1 else z:=0`" introduces a *flow* from `x` to `z`. If `z` is a low-security variable, this is a *downward flow*: information about the value of a high-security variable is communicated to an attacker through a low-security variable, thereby violating secrecy.

There are therefore three straightforward sub-typing rules:

$$\frac{\Gamma \vdash t : L}{\Gamma \vdash t : H} \qquad \frac{\Gamma \vdash e : L \ exp}{\Gamma \vdash e : H \ exp} \qquad \frac{\Gamma \vdash c : H \ comm}{\Gamma \vdash c : L \ comm} .$$

Expressions:

$$\frac{\Gamma \vdash x : \ell}{\Gamma \vdash x : \ell\ exp} \qquad \frac{}{\Gamma \vdash z : \ell\ exp} \qquad \frac{\Gamma \vdash e_1 : \ell\ exp,\, e_2 : \ell\ exp}{\Gamma \vdash e_1\ op\ e_2 : \ell\ exp} \qquad \frac{\Gamma \vdash e : \ell\ exp}{\Gamma \vdash \texttt{inl}\ e : \ell\ exp} \qquad \frac{\Gamma \vdash e : \ell\ exp}{\Gamma \vdash \texttt{inr}\ e : \ell\ exp}$$

$$\frac{\Gamma \vdash e_1 : \ell\ exp,\, e_2 : \ell\ exp}{\Gamma \vdash \texttt{(}e_1,\, e_2\texttt{)} : \ell\ exp} \qquad \frac{\Gamma \vdash e : \ell\ exp}{\Gamma \vdash \texttt{fst}\ e : \ell\ exp} \qquad \frac{\Gamma \vdash e : \ell\ exp}{\Gamma \vdash \texttt{snd}\ e : \ell\ exp} \qquad \frac{\Gamma \vdash e : \ell\ exp}{\Gamma \vdash \texttt{mk-}t\ e : \ell\ exp}$$

Commands:

$$\frac{\Gamma \vdash e : \ell\ exp,\, c : \ell\ comm}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c : \ell\ comm} \qquad \frac{\Gamma \vdash e : \ell\ exp,\, c_1 : \ell\ comm,\, c_2 : \ell\ comm}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \ell\ comm}$$

$$\frac{\Gamma \vdash c_1 : \ell\ comm,\, c_2 : \ell\ comm}{\Gamma \vdash c_1;\ c_2 : \ell\ comm} \qquad \frac{\Gamma \vdash x : \ell,\, e : \ell\ exp}{\Gamma \vdash x := e : \ell\ comm} \qquad \frac{\Gamma \vdash x : \ell',\, e : \ell\ exp,\, f : \ell \to \ell'}{\Gamma \vdash x := f(e) : \ell'\ comm}$$

$$\frac{\Gamma \vdash e : \ell\ exp,\, x_1 : \ell,\, x_2 : \ell,\, c_1 : \ell\ comm,\, c_2 : \ell\ comm}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ \{\ \texttt{inl}\ x_1 \Rightarrow c_1;\ \texttt{inr}\ x_2 \Rightarrow c_2;\ \}\ : \ell\ comm}$$

$$\frac{\Gamma \vdash t : \ell,\, e : \ell\ exp \quad e \text{ has type } t \quad \forall i.\ \forall \tau \in insts(\sigma_i).\ \Gamma[x_i \to \ell] \vdash c_i : \ell\ comm \quad \Gamma \vdash c : \ell\ comm}{\Gamma \vdash \texttt{reprcase}\ e\ \texttt{of}\ \{\ \sigma_1\ x_1 \Rightarrow c_1;\ \ldots;\ \sigma_n\ x_n \Rightarrow c_n;\ \}\ \texttt{else}\ c : \ell\ comm}$$

Auxiliary function (instantiations of a type scheme): $insts \in scheme \to \mathcal{P}(typ)$.

Figure 5.2: Con-freness typing for HL.

The `skip` command can be assigned either security level, but both `update` and `crash` are low-security commands, since both implicitly assign to a low-security pseudo-variable (the program result) whose value is visible to the user, and therefore:

$$\frac{}{\Gamma \vdash \texttt{update} : L\ comm} \qquad \frac{}{\Gamma \vdash \texttt{crash} : L\ comm} \qquad \frac{}{\Gamma \vdash \texttt{skip} : H\ comm}.$$

For example, assume e is an expression of high-security named type `t`. The command "`reprcase` e `of { ... } else crash`" is ill-typed because `crash` is a low-security command. Note that because type representations are not first-class entities, there are no explicit (lo:=hi) flows for named types, only *implicit* flows via `reprcase`. The remaining rules are shown in Figure 5.2. To simplify presentation, the type system tracks only security levels, and not data types (`int`, `prod`, etc.). For a well-typed program, *varying the representations of high-security types will not affect the values of low-security variables.*

The con-free check can be recovered as a type-check. Unusually, in this case type checking must be done at runtime, when an update point is reached. First, types that are part of a dynamic update are assigned the high-security label. Second, global variables, old types and old functions (and their local variables) are given low-security labels. Third, new functions are not type-checked and are assigned a low-security label ($L \to L$). This last condition is necessary to allow old code to examine new named types "by proxy" by calling new functions (this would not be permissible in a conventional non-interference setup, since it constitutes a downward flow). The effect of these requirements is to forbid occurrences in old code of `reprcase` e `of { ... }` where e belongs to a new named type.

5.6 Interpretational overhead of DSU

When an interpreter is written in a typed language, a universal data type is commonly used to represent values of the interpreter language (see §2.5.1), for example:

```
type valu = I of int | F of (valu -> valu)
```

A type constructor in the object language becomes a data constructor for the universal type. An interpreter injects values into the universal type by notionally tagging each value with its object-language type. For example, consider fragments of an OCaml interpreter for call-by-value λ-calculus shown below.

```
let rec eval exp env = match exp with
  | Const n        -> I n
  | Var x          -> env x
  | Lambda (x, e') -> F (fun v -> eval e' (update env (x, v)))
```

A value is untagged by projecting it back into the meta-language type:

```
  | App (e', e'') ->
      (match eval e' env with
         | F f -> f (eval e'' env)
         | _   -> failwith "Type error.")
```

Untagging may fail in general, in the sense that the value is not of the expected object-language type, causing the program to become stuck due to a type error. When the interpreted language is strongly-typed, its type system guarantees the absence of such errors. But if the type system of the meta-language is not expressive enough to encode the types of object-language terms, the interpreter must still carry out needless tagging and untagging. Further, when the interpreter is specialized with respect to the program by means of conventional partial evaluation, the specialized program will also inherit the tagging and untagging code. Immediately, this means that achieving Jones-optimal specialization for strongly-typed language is problematic (see [82]); for our purposes, it is enough to observe that tagging and untagging operations in the residual program constitute interpretational overhead. However, a useful side-effect is that untagging occurs in the residual program in places where the value is used concretely in Stoyle et al.'s sense. We have already remarked that, in a given HL program, the branch taken by every `reprcase` command is known statically: removing the other branches has no effect on the final result of the program, so tagging (`mk`) and untagging (`reprcase`) ostensibly amount to overhead added to permit updates.

5.7 Equivalence of updatable programs

Software for embedded devices must be both updatable and optimized for speed and size to fit in the limited compute and storage capacity of the device. However, optimizing compilation of dynamically-updatable programs presents numerous challenges: Bierman et al. [19] remark that "dynamic rebinding or update primitives invalidate general use of standard optimisations". Establishing the correctness of optimizing program transformations in the presence of updates is also a prerequisite for verifying compilers and any other program manipulation tools.

One immediate problem is that updates are applied at the granularity of functions. So, for example, long-running loops must be manually converted into tail-recursive functions and the compiler somehow prevented from doing tail-call elimination. In practice, performing the tail-call through a function pointer rather than directly is usually sufficient to foil the compiler. Further—to improve availability of a program to updates, update points need to be placed in parts of the program that are executed frequently, even though this is precisely where they will preclude the most effective of optimizations.

Optimizing a dynamically-updatable program using a specific type and function environment may also alter its behaviour with respect to future updates, so the soundness of the optimizations must be reconsidered at each update point. Recall that any given `update` command behaves as `skip` most of the time. But, consider the effect of replacing `skip` with `update` in the following sequence of commands:

```
x:=5; skip; y:=f(z)
```

Assume that global variable `x` is dead in the remainder of the program. Two transformations can be applied to the code as it stands: dead assignment elimination to get rid of `x:=5` and inlining of `f`. However, if `update` is substituted for `skip`, neither optimization is sound because an update may introduce uses of `x` and/or alter the definition of `f`. By the same token, if a tail-recursive function is optimized to a loop, the semantics of the program with respect to updating changes. With tail-recursion, an update to the function definition takes effect on the next recursive invocation. In the loop-based program, the call site no longer exists and the update will never take effect. Therefore, it is necessary to first define what it means for two dynamically-updatable programs to be equivalent.

As a starting point, consider the general problem of defining equivalence of programs that perform input (see 2.3.3). Let the domain of results be $\Omega \cong (\mathit{Val} + (\mathit{Val} \to \Omega))_\perp$ and $[\![\cdot]\!] : \mathit{prog} \to \Omega$. Let $\mathcal{L}$ be a labelled transition system $(S, \Lambda, \rightsquigarrow)$ where states are program results ($S = \Omega$) and the actions ranged over by α are input values ($\Lambda = \mathit{Val}$). A transition from one state to another is possible as long as the state is a resumption (i.e. a continuation, capable of accepting input): $\omega \overset{v}{\rightsquigarrow} \omega' \overset{\text{def}}{=} (\omega = \mathit{inr}\ \kappa) \wedge (\kappa\ v = \omega')$. Recall that a relation $R \subseteq S \times S$ is a *bisimulation* iff

$$\begin{array}{l} \forall t,\, s \in S.\ \forall \alpha \in \Lambda. \\ \forall s'.\ s \overset{\alpha}{\rightsquigarrow} s' \Rightarrow \exists t'.\ t \overset{\alpha}{\rightsquigarrow} t' \ \ \wedge (s',\, t') \in R\, \wedge \\ \forall t'.\ t \overset{\alpha}{\rightsquigarrow} t' \ \Rightarrow \exists s'.\ s \overset{\alpha}{\rightsquigarrow} s' \ \ \wedge (t',\, s') \in R\ . \end{array} \tag{5.5}$$

Bisimulation is a convenient notion of equivalence for communication-centric languages [87]. However, bisimulation is too coarse-grained if we want to treat states as functions and actions as inputs (rather than the other way round) and to encode assumptions about the input values. We generalize equation 5.5 slightly by making it parametric with respect to a secondary relation $P \subseteq \Lambda \times \Lambda$ over actions. The relation $R(P)$ is defined as follows:

$$\begin{array}{l} \forall t,\, s \in S.\ \forall (\alpha,\, \alpha') \in P\ . \\ \forall s'.\ s \overset{\alpha}{\rightsquigarrow} s' \Rightarrow\ \exists t'.\ t \overset{\alpha'}{\rightsquigarrow} t' \ \ \wedge (s',\, t') \in R\, \wedge \\ \forall t'.\ t \overset{\alpha}{\rightsquigarrow} t' \ \Rightarrow \exists s'.\ s \overset{\alpha'}{\rightsquigarrow} s' \ \ \wedge (t',\, s') \in R\ . \end{array} \tag{5.6}$$

Notice that this definition closely mirrors that of PERs for functions (2.7):

$$s\ (P \Rightarrow R)\ t \text{ iff } \alpha\ P\ \alpha' \Longrightarrow (s\ \alpha)\ R\ (t\ \alpha')\ .$$

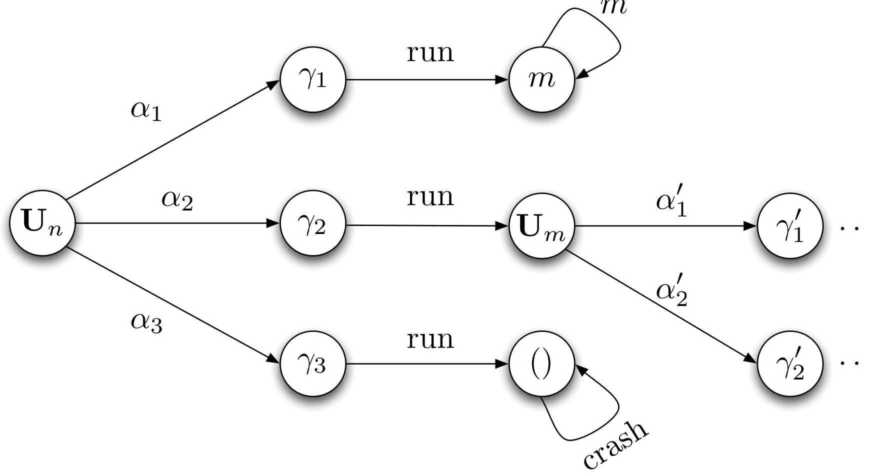


Figure 5.3: Update transition system.

Since, intuitively, a dynamic update is a particular kind of input, we extend this notion of equivalence to dynamically-updatable programs by letting the set of actions be the set $\mathbb{N} \times \mathit{Update}$; the first component of an action is the number of the update point where the update—the second component—can be applied.

A reified configuration $\mathbf{U}_n\langle c, (m_g, k), T, F\rangle$ (see Figure 5.4) consisting of a continuation, program state and type and function environments, transitions to a non-terminal configuration by accepting an update $(n, u) \in \Lambda$. This is represented by a transition of the form $\mathbf{U}_n\langle \ldots \rangle \overset{(n,u)}{\rightsquigarrow} \gamma$. A non-terminal configuration γ transitions to a program answer (in $\Omega_\perp$) by accepting the action "run". E.g.

$$\mathbf{U}_n\langle \ldots \rangle \overset{(n,u)}{\rightsquigarrow} \gamma \overset{\text{run}}{\rightsquigarrow} \mathbf{U}_m\langle \ldots \rangle \overset{(m,u')}{\rightsquigarrow} \ldots \overset{\text{run}}{\rightsquigarrow} m\ .$$

A pseudo-action m is only accepted by the terminal configuration m; similarly for the action "crash" (Figure 5.3). The transition system does not capture the operational semantics of HL itself, and only models updates. The special *empty update* $(n, \emptyset)$, reserved for the situation where no updates are pending, reinstates the configuration unmodified, i.e. $\mathbf{U}_n\gamma \overset{(n,\emptyset)}{\rightsquigarrow} \gamma$. An HL compiler is free to optimize the program so long as the semantics of the program are preserved: this includes behaviour with respect to future updates, i.e. the optimized and original programs are bisimilar (it is not be possible to distinguish between the original and optimized programs by applying updates).

5.7.1 Optimization

Disregarding safety checks (like con-freeness) for the moment, any update can potentially be applied at any update point. This burden on the compiler can be lessened by *restricting the possible updates*. These restrictions are captured in the relation P in Equation 5.6. But it need not be necessary to restrict updates if the optimization can be undone when an update point is reached. At runtime, when an update is pending, we have two options: either (1) reject the update if it would invalidate the optimizations or (2) undo the effect of the optimizations on the program state. (For the example on the previous page, updates that do not change the definition of `f` and do not introduce uses of `x` are acceptable.) For

example, consider the optimization below which exploits knowledge of the representation type of named type `t` to unbox `x`:

Original	Optimized
`type t = int;`	
`var t x:=mk-t 5, int z:=3;`	`var int x:=5, int z:=3;`
`update;`	`update;`
`reprcase x of { int y => z:=y; }`	`z:=x;`

Previously-dead branches of `reprcase` can be re-inserted at an update point if an update is applied that changes the definitions of the affected named types.

We adopt deoptimization [66], a technique from the debugging literature, to relax update restrictions without sacrificing optimizability. (Despite widespread adoption, dynamic deoptimization has, to our knowledge, never been formalized.) At each update point n we introduce a function δ that maps from a configuration of an optimized program to a partially unoptimized configuration, allowing more updates to be applied. Let $\mathbf{U}_n\gamma_0$ be the reified configuration of the unoptimized program at update point n and $\mathbf{U}_n\gamma_{\mathrm{opt}}$ be the corresponding configuration in the optimized program, then δ is a *deoptimization* iff $\mathbf{U}_n\gamma_0$ and $\mathbf{U}_n\delta(\gamma_{\mathrm{opt}})$ are bisimilar.

There are are no computational reflection facilities in HL, so δ cannot be implemented in HL itself. Operationally, this means that δ is a low-level assembly subroutine in the interpreter (or virtual machine) that is executing the HL program. Alternatively, δ can be thought of as a debugging table that contains enough information to recover the interpreted state (see §6.3). The function δ_n for each update point n is constructed at the time the program is optimized, which may be either prior to the program start or at an update point. We rule out the "cheating" δ functions that ignore their input and re-run the original program up to the update point (cf. a cheating specializer, §2.2).

To give an intuition of how this works, suppose that a one-time assignment `z:=42` to a dead global variable `z` is eliminated at compile-time. This limits the updates applicable at an update point that occurs after the assignment to those that do not contain uses of `z`. We can define a deoptimization to undo the effects on memory of this optimization:

$$\delta\ \langle c,\ (m_g,\ m_\ell :: k),\ T,\ F\rangle = \langle c,\ (m_g[\mathtt{z} \mapsto (42,\ \mathtt{int})],\ m_\ell :: k),\ T,\ F\rangle\ .$$

Here, we will consider deoptimization in a small subset of HL (called $\mathrm{HL}_{\mathrm{ANF}}$) without functions or named types with the following syntax (which enforces A-normal form).

5.7.2 Deoptimization in $\mathrm{HL}_{\mathrm{ANF}}$

The syntax of $\mathrm{HL}_{\mathrm{ANF}}$ is shown below:

$$\begin{aligned} \mathit{triv} &::= \mathit{val} \mid \mathit{var} \quad \mathit{exp} ::= \mathit{triv} \mid \mathit{op}\ \mathit{triv}^+ \\ \mathit{comm} &::= \mathit{comm};\mathit{comm} \mid \mathtt{while}\ \mathit{triv}\ \mathtt{do}\ \mathit{comm} \mid \mathtt{skip} \mid \mathit{var} := \mathit{exp} \\ &\quad\mid\ \mathtt{if}\ \mathit{triv}\ \mathtt{then}\ \mathit{comm}\ \mathtt{else}\ \mathit{comm} \mid \mathtt{update}\ n \\ \mathit{prog} &::= \mathit{comm};\ \mathtt{out}\ \mathit{triv}^+ \end{aligned}$$

Let $\longrightarrow$ ("reduces to") be a small-step reduction relation for the language: that is $\gamma \rightarrow \gamma'$ where γ is a non-terminal configuration while γ' is either another non-terminal configuration or a result in Ω. A non-terminal configuration is a pair $\langle c,\ m\rangle$ of command c and memory $m \in \mathit{Mem} \stackrel{\mathrm{def}}{=} \mathit{Var} \rightarrow \mathit{Val}$. A result $\omega \in \Omega$ is either a memory, or a reified configuration $\mathbf{U}_n\langle c,\ m\rangle$ or $\mathbf{U}_n m$ where n is the number of the update point. A configuration is

reified by the `update` command; a reified configuration cannot be reduced further. The semantics of `update` is shown below:

$$\begin{array}{rll} \langle \texttt{update}\ n,\ m\rangle \longrightarrow \mathbf{U}_n m & & \\ \langle c_1\texttt{;}\ c_2,\ m\rangle \longrightarrow \mathbf{U}_n\langle c_1'\texttt{;}\ c_2,\ m'\rangle & \text{if } \langle c_1,\ m\rangle \longrightarrow \langle c_1',\ \mathbf{U}_n m'\rangle \\ \langle c_1\texttt{;}\ c_2,\ m\rangle \longrightarrow \mathbf{U}_n\langle c_2,\ m'\rangle & \text{if } \langle c_1,\ m\rangle \longrightarrow \mathbf{U}_n m' \end{array}$$

The valuation function $[\![\cdot]\!] : \mathit{prog} \to \mathit{Mem} \to \Omega_\perp$ is defined as follows:

$$[\![c\texttt{;}\ \texttt{out}\ (x_1,\ \ldots,\ x_n)]\!]\ m = \begin{cases} m'|_{\{x_1,\ldots,x_n\}} & \langle c,\ m\rangle \longrightarrow^* m' \\ \mathbf{U}_n m' & \langle c,\ m\rangle \longrightarrow^* \mathbf{U}_n m' \\ \mathbf{U}_n\langle c',\ m'\rangle & \langle c,\ m\rangle \longrightarrow^* \mathbf{U}_n\langle c',\ m'\rangle \\ \perp & \langle c,\ m\rangle \uparrow . \end{cases}$$

Now, let $T \in \mathit{prog} \to \mathit{prog}$ be an optimization; then $\delta \in \mathbf{U} \to \mathbf{U}$ is the corresponding dynamic deoptimization iff $[\![p]\!] = (\delta)^* \circ [\![T(p)]\!]$ where the lifting $(\delta)^* : \Omega_\perp \to \Omega_\perp$ acts as identity on all normal results, i.e.

$$(\delta)^*(\omega) = \begin{cases} \delta(\omega) & \text{if } \omega = \mathbf{U}_n\langle c,\ m\rangle \text{ or } \omega = \mathbf{U}_n m \\ \omega & \text{otherwise.} \end{cases}$$

The deoptimization is specific to both the optimization and the program. For example, consider the program $p =$ "`x:=7; y:=9; update 1;`" and let transformation T be dead assignment elimination. Assuming m_0 is an initial memory that maps every variable to zero, we have:

$$\begin{array}{rl} [\![p]\!]\ m_0 & = \mathbf{U}_1\ m_0[\texttt{x} \mapsto 7,\ \texttt{y} \mapsto 9] \\ [\![T(p)]\!]\ m_0 & = [\![\texttt{update 1}]\!]\ m_0 = \mathbf{U}_1\ m_0\ . \end{array}$$

Then $\delta(x) = \mathit{match}\ x\ \mathit{with}\ \mathbf{U}_1 m \Longrightarrow \mathbf{U}_1 m[\texttt{x} \mapsto 7,\ \texttt{y} \mapsto 9]$. In general, the deoptimization must not only reconstruct the state, but also the continuation. We assume that the original program code is always available.

5.7.3 Other semantics for updating

The `update` command is conveniently defined in a continuation semantics (see §2.3.3); an example is given below (assuming $[\![\cdot]\!]^{\mathit{exp}}$ is a valuation function for both trivial expressions a and ordinary expressions e):

$$\begin{array}{rcl} [\![\cdot]\!] & : & \mathit{comm} \to (\mathit{Mem} \to \Omega_\perp) \to \mathit{Mem} \to \Omega_\perp \\ [\![\texttt{while}\ a\ \texttt{do}\ c]\!] & = & \lambda k.\ \mathit{fix}_{\mathit{Mem}\to\Omega_\perp}\ \lambda f.\lambda m.\ \mathit{if}\ m([\![a]\!]^{\mathit{exp}}\ m) \neq 0\ \mathit{then}\ ([\![c]\!]\ f\ m)\ \mathit{else}\ (k\ m) \\ [\![\texttt{if}\ a\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] & = & \lambda k.\lambda m.\ \mathit{if}\ m([\![a]\!]^{\mathit{exp}}\ m) \neq 0\ \mathit{then}\ ([\![c_1]\!]\ k\ m)\ \mathit{else}\ ([\![c_2]\!]\ k\ m) \\ [\![x\ \texttt{:=}\ e]\!] & = & \lambda k.\lambda m.\ k(m[x \mapsto [\![e]\!]^{\mathit{exp}}\ m]) \\ [\![\texttt{skip}]\!] & = & \lambda k.\lambda m.\ k(m) \\ [\![\texttt{update}\ n]\!] & = & \lambda k.\lambda m.\ \mathbf{U}_n(k,\ m) \\ [\![c_1\texttt{;}\ c_2]\!] & = & [\![c_1]\!] \circ [\![c_2]\!]\ . \end{array}$$

However, in this semantics the continuation k is an opaque object: the only thing we can do with a continuation is apply it to a value. An operational semantics exposes the continuation as program text. Another semantics may use a different internal representation of the continuation and memory: e.g. a flow-chart interpreter holds the entire control flow graph in memory. The continuation is then identified with the current execution location,

a pair of basic block number and an offset into the basic block. An operational semantics arguably exposes too much internal detail, whereas a denotational continuation semantics like that above does not provide enough. In future work, we plan to investigate definition of software updating that is independent of the underlying semantics.

5.8 Related work

5.8.1 Dynamic typing

Abadi, Cardelli, Pierce, and Rémy [3] further discuss technical issues which arise when Dynamic is used in various contexts and in conjunction with other type-systemic features. They consider: ML (a language with implicit prenex-quantified polymorphism), the second- and higher-order polymorphic λ-calculi (System F and F_ω, respectively), abstract datatypes, and subtyping. The development revolves around correct scoping and binding rules for pattern variables in the `typecase` construct.

More recently, Siek and Taha [120] describe the calculus $\lambda^{?}_{\rightarrow}$, a simply-typed λ-calculus with a syntactically optional Dynamic type (?). Coercions (*casts*) are added by translation to the intermediate calculus $\lambda^{\langle\tau\rangle}_{\rightarrow}$. The approach is termed *gradual typing*. The authors recover the untyped λ-calculus and the simply-typed λ-calculus in the annotation-free and fully-annotated cases respectively. Siek and Taha [121] extend these ideas to the Abadi-Cardelli object calculus. Gray et al. [59] describe an approach to interoperability between Java and Scheme (a dynamically-typed language) by adding a dynamic type to Java and using inferred contracts to preserve type-safety. The application of contracts is structured using *mirrors*, a construct proposed by Bracha and Ungar [20]. Of practical necessity, language implementations provide means to perform cross-language calls by way of a Foreign Function Interface (FFI). Furr and Foster show how to statically verify type-safety of calls between OCaml and C [51] and Java and C [52], in both cases via a unified type system.

In their work on *soft typing*, Cartwright and Fagan [25] take the view that a type-checker should never reject a program. The authors augment a purely applicative subset of ML with union types[2] and recursive types and present a type reconstruction algorithm. Coercions are inserted to restore well-typedness of badly-typed programs.

Washburn and Weirich [146] use information-flow annotations at the type system level in a language with runtime type analysis with the goal of recovering some form of parametricity (data abstraction). Their type system is significantly more sophisticated and interesting than HL's, but their language does not support DSU.

5.8.2 Con-freeness and dependency

Two excellent surveys—[112] and [114]—give an overview of the recent research in language-based information-flow security. Hicks et al. [64] considered the impact of dynamic updates to the information-flow policy of programs written in a language with security types. (The code of the programs is not updated.) The authors remark that Stoyle et al.'s con-free check is analogous to dynamic checking of permission tags. We have explained *why* we should expect enforcement mechanisms for secure information flow and con-freeness to be similar (because the underlying semantic properties are). Our imperative language is

[2]For instance, $\lambda \mathtt{x}.\mathtt{if}\ \mathtt{x} = 42\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ 0 : \mathbf{int} \rightarrow \mathbf{int} + \mathbf{bool}$.

quite different from Hicks et al.'s security-typed functional language: e.g. HL has named types, runtime type analysis and a notion of a security level for a type.

PERs are often used to capture a notion of dependency: some example applications of PERs are given in Appendix B. Benton [15] built an equational theory (DDCC) to support proofs of optimizing program transformations on top of a PER semantics for a simple imperative language.

5.8.3 Optimization

Source-level debugging of optimized programs poses similar challenges as that of updating them (see §2.4.1). Tolmach and Appel [132] observe: "Optimizing compilers may make any changes to a program so long as the observable behavior of the program remains the same. Unfortunately, debuggers must expose the internal behavior of the original program, which may be altered by optimization". Their solution is to add instrumentation code to the program prior to compilation. Although standard optimizations can be used on the instrumented program, runtime overheads remain substantial.

Virtual methods in object-oriented (OO) languages that support dynamic class loading provide a simple DSU facility. Virtual method inlining [38] in this setting causes similar problems to those discussed above. If Class Hierarchy Analysis (CHA) indicates that the receiver of a virtual call is always known, then it is safe to inline the method at the call site. This optimization is speculative in nature: dynamic class loading may invalidate the results of CHA. (See also the discussion in §4.4.1). HotSpot implements dynamic deoptimization to cope with this scenario. Dynamic class loading is part of the language definition in languages like Java and must be supported by the implementation (both compiler and runtime). In contrast, HL does not define what constitutes a valid update, leaving an implementation free to allow some updates but not others.

Subramanian et al. [127] described an extension of Jikes (a Java VM) called JVOLVE that supports dynamically updatable programs. JVOLVE cannot update a method if the method itself is active or if any method that has it inlined is active, something which Hölzle et al.'s [66] dynamic deoptimization proposal for SELF can handle. Interestingly, the implementation of SELF clones the code of the function when creating an activation frame, so when a function definition is changed, those functions that are already active continue to execute old code—which exactly matches DSU behaviour.

5.9 Conclusions and further work

We have presented a rational reconstruction of con-freeness, a property introduced by Stoyle et al. [126] in their work on type-safety for dynamically-updatable programs. In the setting of a simple imperative language with runtime type analysis, we identified an extensional ("semantic") counterpart of the original property that is independent of any particular updating mechanism. In order to formalize it, we found it necessary to split the semantics of our language into separate evaluation functions for old and new code in a program, suggesting a possible connection with Matthews and Findler's [84] work on multi-language interoperability. In contrast to previous calculi for dynamic updating, the mechanics of updating are not part of the semantics of our language. We argue the benefits of this approach for program optimization (well-known to be antagonistic to updatability) and propose an observational equivalence for updatable programs. We defined a non-interference type system inspired by the original con-free check: old code may perform

runtime type analysis on values of new types provided this does not introduce downward flows to observable program results. This is useful for typing e.g. instrumentation code.

In future work, we plan to investigate a notion of "blame" [46, 141] which arises naturally in dynamically-updatable programs: old code that is part of a con-t-free program cannot be blamed for dependencies on t of the program as a whole, as all such dependencies ultimately originate in new code. As far as we are aware, there is no accepted formal definition of dynamic deoptimization: it would be interesting to extend and generalize the definitions in §5.7.2.

Configuration

$\langle c,$	Command
$m_g,$	Global memory
$m_\ell ::$	Current activation frame
$k,$	Call stack
$T,$	Type environment
$F\rangle$	Function environment

Mem. lookup

$$(m_g,\, m_\ell)(x) = \begin{cases} m_\ell(x) & x \in dom(m_\ell) \\ m_g(x) & \text{otherwise} \end{cases}$$

$$(m_g,\, m_\ell)[x \mapsto v] = \begin{cases} (m_g,\, m_\ell[x \mapsto v]) & x \in dom(m_\ell) \\ (m_g[x \mapsto v],\, m_\ell) & \text{otherwise} \end{cases}$$

Shorthands:

$$\bullet \equiv m_g,\, m_\ell :: k,\, T,\, F$$
$$\bullet[x \mapsto v] \equiv m'_g,\, m'_\ell :: k,\, T,\, F \text{ such that } (m'_g,\, m'_\ell) = (m_g,\, m_\ell)[x \mapsto v]$$

$$\begin{array}{rcll}
\langle \texttt{while}\ e\ \texttt{do}\ c,\, \bullet\rangle & \longrightarrow & \bullet & \text{if } [\![e]\!]_{\text{HL}}\ \bullet = (0,\, \texttt{int}) \\
\langle \texttt{while}\ e\ \texttt{do}\ c,\, \bullet\rangle & \longrightarrow & \langle c;\ \texttt{while}\ e\ \texttt{do}\ c,\, \bullet\rangle & \text{otherwise} \\
\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\, \bullet\rangle & \longrightarrow & \langle c_2,\, \bullet\rangle & \text{if } [\![e]\!]_{\text{HL}}\ \bullet = (0,\, \texttt{int}) \\
\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2,\, \bullet\rangle & \longrightarrow & \langle c_1,\, \bullet\rangle & \text{otherwise} \\
\langle \texttt{skip},\, \bullet\rangle & \longrightarrow & \bullet & \\
\langle \texttt{case}\ e\ \texttt{of}\ \{\ \texttt{inl}\ x_1 \Rightarrow c_1;\ \ldots\ \},\, \bullet\rangle & \longrightarrow & \langle c_1,\, \bullet[x_1 \mapsto (v,\, \tau_1)]\rangle & [\![e]\!]_{\text{HL}}\ \bullet = (inl\ v,\, \texttt{sum}(\tau_1,\, \tau_2)) \\
\langle \texttt{case}\ e\ \texttt{of}\ \{\ \ldots;\ \texttt{inr}\ x_2 \Rightarrow c_2\ \},\, \bullet\rangle & \longrightarrow & \langle c_2,\, \bullet[x_2 \mapsto (v,\, \tau_2)]\rangle & [\![e]\!]_{\text{HL}}\ \bullet = (inr\ v,\, \texttt{sum}(\tau_1,\, \tau_2)) \\
\langle \texttt{case}\ e\ \texttt{of}\ \{\ \ldots\ \},\, \bullet\rangle & \longrightarrow & \bullet & \text{otherwise} \\
\langle x := e,\ \bullet\rangle & \longrightarrow & \bullet[x \mapsto [\![e]\!]_{\text{HL}}\ \bullet] &
\end{array}$$

$$\begin{array}{l}
\langle \texttt{reprcase}\ e\ \texttt{of} \\
\quad \{\ \sigma_1 x_1 \Rightarrow c_1;\ \ldots;\ \sigma_n x_n \Rightarrow c_n;\ \}\ \texttt{else}\ c,\, \bullet\rangle \longrightarrow \langle c_i,\, \bullet[x_i \mapsto (v,\, T(t))]\rangle \\
\qquad \text{where } [\![e]\!]_{\text{HL}}\ \bullet = (v,\, t) \\
\qquad \text{and } i = min\ match(\sigma_1,\, \ldots,\, \sigma_n,\, T(t))
\end{array}$$

$$\begin{array}{rcll}
\langle \texttt{reprcase}\ e\ \texttt{of}\ \{\ \ldots\ \}\ \texttt{else}\ c,\, \bullet\rangle & \longrightarrow & \langle c,\, \bullet\rangle & \text{otherwise} \\
\langle c_1;\ c_2,\, \bullet\rangle & \longrightarrow & \langle c'_1;\ c_2,\, \bullet'\rangle & \text{if } \langle c_1,\, \bullet\rangle \longrightarrow \langle c'_1,\, \bullet'\rangle \\
\langle c_1;\ c_2,\, \bullet\rangle & \longrightarrow & \langle c_2,\, \bullet'\rangle & \text{if } \langle c_1,\, \bullet\rangle \longrightarrow \bullet' \\
\langle \texttt{crash};\ c,\, \bullet\rangle & \longrightarrow & () & \\
\langle \texttt{update}\ n;\ c,\, \bullet\rangle & \longrightarrow & \mathbf{U}_n\langle c,\, (m_g,\, m_\ell :: k),\, T,\, F\rangle & \\
\langle \texttt{return}\ e,\, \bullet\rangle & \longrightarrow & m_g,\, m_\ell[\texttt{\$ret} \mapsto v] :: k,\, T,\, F & \\
 & & \quad \text{where } v = [\![e]\!]_{\text{HL}}\ \bullet & \\
\langle \textit{pop-frame}\ x,\, m_g,\, m_\ell^{\text{stale}} :: (m_\ell :: k),\, T,\, F\rangle & \longrightarrow & m'_g,\, m'_\ell :: k,\, T,\, F & \\
 & & \quad \text{where } (m'_g,\, m'_\ell) = (m_g,\, m_\ell)[x \mapsto m_\ell^{\text{stale}}(\texttt{\$ret})] & \\
\langle x := f(e),\ \bullet\rangle & \longrightarrow & \langle c; \textit{pop-frame}\ x,\, m_g,\, m'_{\text{proto}} :: (m_\ell :: k),\, T,\, F\rangle & \\
 & & \quad \text{where } F(f) = (x_{\text{arg}},\, m_{\text{proto}},\, c) & \\
 & & \quad \text{and } m_{\text{proto}}[x_{\text{arg}} \mapsto [\![e]\!]_{\text{HL}}\ \bullet] &
\end{array}$$

Notes

pop-frame is internal syntax to pop the activation frame of a function that has returned
`$ret` is a local variable which holds the return value
$\mathbf{U}_n\langle c,\, (m_g,\, m_\ell :: k),\, T,\, F\rangle$ is a *reified configuration* (see §5.7.2)

Figure 5.4: Operational semantics of HL commands.

Chapter 6

Information flow and (de)compilation

In both dynamic software updating and dynamic instrumentation, the dependence of future updates or instrumentation code on intermediate program execution states is a limiting factor for static optimization. In Chapter 4, we used PERs to gauge the "invasiveness" and potential optimization impact of instrumentation code, and similarly, at the end of Chapter 5, we considered restricting allowable dynamic updates to enable more aggressive static optimization. We showed that deoptimization can be used to lift or relax some of the restrictions. Recall that deoptimization, originally due to Hölzle et al. [66], is a program state transformation that reconstructs an unoptimized program state from an optimized program state. In this chapter, which is more speculative than the preceding ones, we frame a related problem, decompilation, in terms of *information-flow security*. Recall (§2.1) that decompilation is the process of recovering the source code of a program from its machine code. We will ignore the problem of *disassembling* machine code, i.e. turning machine code into a human-readable assembly listing [79].

Optimizing compilation has a normalizing effect: many semantically equivalent but syntactically distinct expressions are translated to the same assembly code. Therefore, intuitively, the better the compiler is at its job, the more difficult it will be for an adversary to recover the original source code. For instance, if the compiler implements constant propagation and folding, an instruction like "`MOV r0, 4`" could correspond to a number of source language phrases: e.g., assuming register `r0` holds the value of variable `x`, we could have "`x:=4`", "`x:=2+2`" or "`y:=5; x:=y-1`". We will assume that the source code is a secret, but the machine code is made publicly available. The *information flow* from the input source code to the output machine code is characterized by transformations the compiler performs.

In the second part of the chapter, we look at a possible semantics for *debug tables*: data generated by the compiler to help a debugger map from the optimized execution state back to the source-level state.

Outline. In §6.1 we define what it means for a compiler to have *secure information flow*. This view naturally accommodates randomized compilation [49] and gives a novel perspective on superoptimization [83] as a security-enhancing transformation (§6.2). No compiler for a Turing-complete language has zero information flow: this is a trivial result. In §6.3, we consider translation from the previous chapter's $\mathrm{HL}_{\mathrm{ANF}}$ language to a stack assembly language called SAL. We propose an operational model of debug tables as

"boundaries" between optimized (compiled) and unoptimized (interpreted) code. Related work is discussed in §6.4, and §6.5 concludes.

6.1 Normalization and decompilation

For a compiler $\mathcal{C}$ from language L to language M (both Turing-complete), define *the kernel of* $\mathcal{C}$ as the induced equivalence relation on L-programs[1]:

$$ker(\mathcal{C}) = \{(p, p') \mid \mathcal{C}(p) =_\alpha \mathcal{C}(p')\} \ . \tag{6.1}$$

Better-optimizing compilers are more normalizing and therefore have bigger kernels[2]. Analyses that underlie optimizing program transformations are expected to be sound, but, as a consequence of Rice's theorem, they are almost never complete, so no perfectly optimizing compilers for Turing-complete languages can exist: see, for example, p. 378 of Appel's textbook [9]. Therefore, for any given $\mathcal{C}$, it must be the case that $ker(\mathcal{C}) \subset \sim_L$ where $\sim_L$ is the observational equivalence relation on L-programs (note strict inclusion). Most sensible compilers produce the same output for inputs that are equivalent up to renaming, so therefore $=_\alpha \subseteq ker(\mathcal{C})$.

A decompiler $\mathcal{D}$ (§2.1) is a translation from M back to L which recovers a program semantically equivalent to the compiler's input:

$$\forall p \in L.\ (\mathcal{D} \circ \mathcal{C})(p) \in [p]_{\sim_L} \ . \tag{6.2}$$

Although decompilation can be thought of and implemented as "reverse compilation", this leads to trivial solutions: for example, $\mathcal{D}(p) = [\![mix_{\text{triv}}]\!](int^L_M, p)$ where mix_{triv} is a trivial specializer (§2.2) and int^L_M is an interpreter for the target language M written in the source language L. In practice, decompilers exploit knowledge of the compiler's internal structure. However, even the best possible decompiler can only recover the original source code up to the kernel relation of the compiler:

$$\forall p \in L.\ (\mathcal{D} \circ \mathcal{C})(p) \in [p]_{ker(\mathcal{C})} \ . \tag{6.3}$$

Further selection between equivalent source programs should be left to the user, since the decompiler's choice will necessarily be arbitrary, reflecting the internal structure of the decompiler rather than the program. Equations 6.2 and 6.3 have a natural interpretation in terms of information-flow security which we examine below.

6.1.1 Motivation: non-interference and full abstraction

Non-interference is usually defined as zero information flow from high-security inputs to low-security outputs. Suppose we partition the variables of a program into low-security and high-security. Let $=_{\text{low}}$ be a be a binary relation on states such that $s_1 =_{\text{low}} s_2$ iff the low-security parts of s and s' are the same:

$$s_1 =_{\text{low}} s_2 \quad \text{iff} \quad \forall x.\ s_1^{\text{low}}(x) = v \iff s_2^{\text{low}}(x) = v \ .$$

[1]Note that $\mathcal{C} \in |R|$ where R is the PER ($ker(\mathcal{C}) \Rightarrow =_\alpha$).

[2]We assume $\mathcal{C}$ is implemented on a finite machine, the set of input programs is finite and therefore $ker(\mathcal{C})$ is also finite.

The "observational power of an attacker" [112] is captured by a relation $=_{\text{att}}$ on program results. For example, if the attacker is able to observe the low-security part of the output, we would define $=_{\text{att}}$ to coincide with $=_{\text{low}}$. Intuitively, an attacker should remain oblivious to variations in high-security inputs. Non-interference for command c is defined as follows:

$$s_1 =_{\text{low}} s_2 \Longrightarrow (\llbracket c \rrbracket\ s_1) =_{\text{att}} (\llbracket c \rrbracket\ s_2)\ . \tag{6.4}$$

Full abstraction for language translations [1] expresses a similar idea to non-interference: recall (§2.1.1) that a compiler $\mathcal{C}$ from L to M is fully abstract iff

$$e_1 \sim_L e_2 \iff \mathcal{C}(e_1) \sim_M \mathcal{C}(e_2) \tag{6.5}$$

where $\sim_L$ and $\sim_M$ are contextual equivalence relations over L and M. Abadi [1] notes that the forward implication "means that the translation does not introduce information leaks". Let us rewrite Equation 6.4 and the forward implication of Equation 6.5 in PER notation:

$$\llbracket c \rrbracket : =_{\text{low}} \Rightarrow =_{\text{att}} \qquad \mathcal{C} : \sim_L \Rightarrow \sim_M \ .$$

The correspondence $(\llbracket c \rrbracket, =_{\text{low}}, =_{\text{att}}) \leftrightarrow (\mathcal{C}, \sim_L, \sim_M)$ is immediate. Note that full abstraction concerns internal observations of the target language, whereas non-interference usually deals with external observations. The distinction becomes less pronounced if the target language can make very fine-grained observations. For example, if the target language M has a Gödelizing context (see §2.4), then, without loss of generality, we can represent a decompiler as an M-context, i.e. $\mathcal{D}[-] \in \mathbf{Ctx}_M$. Consider decompilation as a form of attack in the sense of Equation 6.4: the low-security part of the compiler's input is the extensional (I/O) meaning of the program and the high-security part is the program text. The relation $=_{\text{att}}$ encodes the degree of access that the attacker has to the program's machine code as well as the degree of sophistication of the attacker. Even if direct access is not possible (e.g. the machine code is part of a tamper-proof embedded system), the attacker may still be able to recover traces of individual program runs: for example, Vermoen et al. [135] obtain the bytecode trace of a Java program by power analysis. Similarly, in most web applications, parts of the program reside on the web server, while others run on the client. It is usually safe to assume that a would-be attacker cannot scrutinize server-side code.

6.1.2 Secure information flow for a compiler

A program that obeys an information flow policy is said to have *secure information flow*. A compiler $\mathcal{C}$ has secure information flow with respect to an adversary defined by $=_{\text{att}}$ iff

$$\mathcal{C} : \sim_L \Rightarrow =_{\text{att}} \ . \tag{6.6}$$

Less normalizing compilers are in principle more susceptible to attack. Assuming a decompiler that can distinguish all target programs, we instantiate $=_{\text{att}}$ to textual equality. A *compiler $\mathcal{C}$ has zero information flow* iff

$$\mathcal{C} : \sim_L \Rightarrow =_\alpha \ . \tag{6.7}$$

A compiler that has zero information flow leaks only the equivalence class of its input programs: from the output of $\mathcal{C}$ an attacker can recover the original program to within a

semantic equivalence class. However, this requires a perfectly optimizing compiler $\mathcal{C}$. We conclude that no compiler can have zero information flow.

The power of the attacker lies in the kinds of program analyses and transformations that decompiler is capable of performing. This can be rendered by substituting an appropriate relation for $=_{\text{att}}$: by definition, the compiler $\mathcal{C}$ can "withstand" an attacker represented by the relation R as long as $\mathcal{C} : \sim_L \Rightarrow R$. The obvious downside of this approach is its reliance on assumptions about the attacker[3]. Following Clark et al. [31], given function f and a relation R over its co-domain, define the relation $f^{-1}(Q)$ (the kernel of f with respect to R) as follows:

$$x\ f^{-1}(R)\ x' \iff (f\ x)\ R\ (f\ x')\ . \tag{6.8}$$

Note that, by definition (Equation 6.1), $ker(\mathcal{C}) = \mathcal{C}^{-1}(=_\alpha)$ and so

$$p_1\ \mathcal{C}^{-1}(=_\alpha)\ p_2 \iff \mathcal{C}(p_1) =_\alpha \mathcal{C}(p_2)\ .$$

For a naive non-optimizing compiler, $\mathcal{C}^{-1}(=_\alpha)$ coincides with $=_\alpha$. We are ultimately interested in finding a characterization for the implicitly-defined relation $\mathcal{C}^{-1}(R)$ given an attacker relation R. However, defining R requires a relational model of decompilation (describing which machine code fragments the decompiler considers to be equivalent), rather than the more practical operational approach favoured in the literature. We leave this to future work.

6.2 Applications

6.2.1 Superoptimization

Given a sequence of assembly instructions, "superoptimization" [83] finds, by exhaustive search, the shortest sequence of instructions that has the same effect. Let $sopt(\cdot)$ be a superoptimizer and let $\sim_{\text{term}}$ relate semantically equivalent straight-line terminating programs, then $sopt : \sim_{\text{term}} \Rightarrow =_\alpha$. Massalin [83] notes that a superoptimizer frequently comes up with sequences of instructions that exploit the instruction set of the target machine in unexpected ways which a human programmer is unlikely to consider or immediately comprehend. In this light, superoptimization could be thought of as a kind of program obfuscation: a deliberate attempt to obscure the workings of the program.

6.2.2 Randomized compilation

Software diversity [49] techniques are used to counter this. Randomized compilation [32, 49] gives a measure of protection against software exploits. Much like with non-deterministic encryption schemes, the idea here is to thwart learning by the adversary. Since each binary of a program compiled with a randomizing compiler is slightly different, knowledge gained from another binary and hard-coded into the exploit becomes useless.

As a simplification, assume that an exploit either applies to a given program, causing it to misbehave in some way, or fails to apply. An exploit implicitly defines a relation ($=_{\text{exp}}$) on binary programs which relates the intended target program p_{tgt} only to itself and all remaining programs to each other, i.e. $p_{\text{tgt}} =_{\text{exp}} p_{\text{tgt}}$ and $p =_{\text{exp}} p'$ for all p, p' such that $p \neq p_{\text{tgt}}$ and $p' \neq p_{\text{tgt}}$. This means that the equivalence class of p_{tgt} is a singleton. Even

[3] Similar in spirit to the probabilistic-polynomial-time (PPT) restriction usually made in cryptography.

a minor change in the compilation process is enough to produce a sufficiently altered program p'_{tgt} outside the equivalence class. Of course, if the exploit targets a widely-used library function, every program using the function is potentially vulnerable. Let $\mathcal{E} \in \mathit{prog} \to \mathbb{B}$ be a function which attempts to apply an exploit to its argument program, returning truth ($\mathtt{tt}$) on success and falsity ($\mathtt{ff}$) on failure. Then the exploit relation is induced by $\mathcal{E}$, i.e.:

$$\begin{aligned} p =_{\mathcal{E}} p' &\quad \text{iff } \mathcal{E}(p) = \mathcal{E}(p') = \mathtt{tt} \\ p =_{\mathcal{E}} p' &\quad \text{iff } \mathcal{E}(p) = \mathcal{E}(p') = \mathtt{ff} \ . \end{aligned}$$

Compare randomized compilation with Joshi and Leino's equational characterization of non-interference [74]. Let HH be the "havoc on high" function that randomizes the high-security portion of its argument. Non-interference for a program S is captured by the following equation (where $\doteq$ is semantic equivalence):

$$HH;\ S;\ HH \doteq S;\ HH\ . \tag{6.9}$$

The occurrences of HH on the right-hand side of S conceal the final values of high-security variables. Intuitively, this means that the values of low-security outputs remain the same when the high-security inputs are set to arbitrary values. If S does not have any high-security outputs, we can rewrite the equation to

$$HH;\ S \doteq S\ . \tag{6.10}$$

For example, semantic "deadness" of a variable x can be expressed by letting HH randomize the value of x. Assuming S implements a compiler, we let HH randomize the *program text* (a high-security input) without changing its meaning: i.e. for an L-program p, we have $HH(p) \sim_L p$. For a compiler $\mathcal{C}$ such that $HH;\ \mathcal{C} \doteq \mathcal{C}$, the effect of randomizing compilation cannot be achieved by randomizing the input source code. Intuitively, this is because the compiler is required to "undo" the randomizing rewrites. There is an antagonism between normalization and randomization.

6.2.3 Adaptive compilation: a proposal

In an equational theory, expressions of a language are related at a particular type, i.e. we say that "$1 = 5 : \mathsf{bool}$" (read as "one is equal to five at type bool"), but, of course, "$1 \neq 5 : \mathsf{int}$". Benton [15] espouses this style of relational reasoning for program analyses and transformations since whether (and how) a phrase may be optimized usually depends on the surrounding context. Specifically, Benton notes that many optimizations are only valid in a particular context; similarly, Benton and Zarfaty [17] stress the importance of being able to "link soundly with code compiled from other high-level languages" as well as "library routines that are written directly in a low-level language". Recall (§2.1.1) that expressions e and e' are contextually equivalent ($e \sim e'$) whenever

$$\forall \mathsf{Ctx}[-].\quad \mathsf{Ctx}[e] \cong \mathsf{Ctx}[e']$$

where $\mathsf{Ctx}[-]$ ranges over contexts of the language and $\cong$ is some observation (say, convergence). Let us generalize the definition of zero information flow for a compiler $\mathcal{C}$ from L to M from the previous section with respect to a particular M-context $\mathsf{Ctx}[-]$:

$$\mathcal{C}\ : \sim_L \Rightarrow \sim_{\mathsf{Ctx}[-]}\ . \tag{6.11}$$

$$
\begin{aligned}
\llparenthesis x \rrparenthesis &= \texttt{LOAD}\ x \\
\llparenthesis v \rrparenthesis &= \texttt{IMM}\ v \\
\llparenthesis e_1\ op\ e_2 \rrparenthesis &= \llparenthesis e_1 \rrparenthesis;\ \llparenthesis e_2 \rrparenthesis;\ op \\
\llparenthesis x := e \rrparenthesis &= \llparenthesis e \rrparenthesis;\ \texttt{STORE}\ x \\
\llparenthesis \texttt{skip} \rrparenthesis &= \texttt{NOP} \\
\llparenthesis \texttt{if}\ a\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rrparenthesis &= \llparenthesis a \rrparenthesis;\ \texttt{JZ}\ \ell;\ \llparenthesis c_1 \rrparenthesis;\ \texttt{JMP}\ \ell';\ \ell\text{:}\ \ \llparenthesis c_2 \rrparenthesis;\ \ell'\text{:} \\
\llparenthesis \texttt{while}\ a\ \texttt{do}\ c \rrparenthesis &= \ell\text{:}\ \ \llparenthesis a \rrparenthesis;\ \texttt{JZ}\ \ell';\ \llparenthesis c \rrparenthesis;\ \texttt{JMP}\ \ell;\ \ell'\text{:} \qquad \text{where } \ell,\, \ell' \text{ fresh.} \\
\llparenthesis c_1;\ c_2 \rrparenthesis &= \llparenthesis c_1 \rrparenthesis;\ \llparenthesis c_2 \rrparenthesis \\
\llparenthesis \texttt{out}\ a_1,\, \ldots,\, a_n \rrparenthesis &= \texttt{OUT}\ a_1,\, \ldots,\, a_n
\end{aligned}
$$

Figure 6.1: Translation from HL_{ANF} to SAL.

Writing $e \sim_{\mathsf{Ctx}[-]} e'$ simply means that $\mathsf{Ctx}[e] \cong \mathsf{Ctx}[e']$. Expanding Equation 6.11 gives:

$$\forall e,\, e'.\quad e \sim_L e' \implies \mathsf{Ctx}[\mathcal{C}(e)] \cong \mathsf{Ctx}[\mathcal{C}(e')]\ .$$

If we view contexts as types, the original definition is polymorphic in the sense that we quantify over all possible types/contexts. In a compositional semantics, the meaning of a phrase depends solely on the meanings of its sub-phrases, i.e. the result of the translation is independent of any possible surrounding M-context. A natural question to ask is whether, in compilation, the translation should be parameterised by the target context, allowing the compiler to adapt its optimization effort to the attacker's capabilities. In some respects this is already the case in practice, since command-line options (like GCC's `-fomit-frame-pointer`) change the output of the compiler in a way that is observable with a debugger. However, to our knowledge, this behaviour is not formalised to any extent, and accounts in the literature assume, as we have until now, that a compiler is simply a function from source programs to target programs.

6.3 An operational view of debug tables

SAL is a stack assembly in the spirit of Java bytecode or PostScript, with syntax shown below; we omit the straightforward semantics.

$$
\begin{aligned}
&\mathit{insn} ::= \texttt{IMM}\ \mathit{val} \mid \texttt{DUP} \mid \texttt{POP} \mid \mathit{op} \mid \texttt{LOAD}\ \mathit{var} \mid \texttt{STORE}\ \mathit{var} \mid \texttt{JZ}\ \mathit{lbl} \mid \texttt{JMP}\ \mathit{lbl} \mid \texttt{NOP} \\
&\mathit{prog} ::= \mathit{insn}^\star;\ \texttt{OUT}\ \mathit{triv}_1,\, \ldots,\, \mathit{triv}_n \qquad\qquad \text{where } \mathit{triv} ::= \mathit{val} \mid \mathit{var}\ .
\end{aligned}
$$

We leave the base type of values unspecified. The `OUT` instruction outputs the values of its arguments—which can be either variables or immediate values—at the end of the program run; for a variable argument, it can be thought of as declaring the variable live. The translation from HL_{ANF} (a generic first-order imperative language from §5.7.2) to SAL is shown in Figure 6.1. For simplicity, we assume that HL_{ANF} and SAL share the same base type of values and support the same family of operators. The translation is naive (sometimes known as "compilation by macro-expansion"), and optimization is possible both in the front- and back-end of the compiler. We will consider constant propagation in HL_{ANF} and replacement of repeated `LOAD`s with a `DUP` (a peephole rewrite) in SAL. The optimizations add *debug information* to their output which includes a mapping from assembly to source line number as well as *debug tables*. A *debug table entry* usually takes the form of a triplet: a variable, function or type name, metadata (e.g. variable `x` is

stored in register `r0`) and the scope of the entry. Our particular design for debug tables is described below.

Consider the program "`x:=7; if x then y:=12 else y:=13; out y;`". Following constant propagation, the assignment `x:=7` becomes dead, and so we have:

```
       {x:7}
       IMM 7;  JZ ff;        -- if x
       IMM 12; STORE y;      -- then y:=12
       JMP done
ff:    IMM 13; STORE y;      -- else y:=13
done:  OUT y                 -- out y
```

In the example above, the debug table `{x:7}` consists of the single entry `x:7` which gives the value of `x` at that point in the program. The notation x`:?` is used when the value of x cannot be statically determined; the notation x:y means that variable x contains the value of some other variable y. The contents of the stack can also optionally be described by treating stack locations as pseudo-variables with names beginning with the letter "s", e.g. `s0:42` means that the top of the stack contains the literal value 42, and `s1:z` means that the first element from the top holds the value of variable `z`.

Note that the translation in Figure 6.1 preserves variables names: an HL_{ANF} variable `x` is mapped to a SAL variable `x`. After removing the dead `else`-branch and propagating the value of `y` to the `out` instruction, we have:

```
{x:7, y:12}; OUT 12;
```

Debug tables of this kind are traditionally seen as passive and declarative: they constitute additional information that may be interpreted by a debugger or a program comprehension tool. Here, we suggest an alternative operational interpretation.

6.3.1 Language boundaries

In many ways the effect of debug tables is similar to deoptimization at update points (§5.7.2). Unsurprisingly, the semantics of simple table entries—entries of the form x:v—can be defined as a relation over values, with the semantics of the whole table built up in the usual way (as a map from variable names and stack positions to relations over values). Let x_i range over variable names and stack locations, then

$$[\![\{x_1{:}v_1,\ x_2{:}v_2,\ \ldots,\ x_n{:}?\}]\!] = \{x_1 \mapsto Eq(v_1),\ x_2 \mapsto Eq(v_2),\ \ldots,\ x_n \mapsto All\}\ .$$

So, for example, $[\![$ `{x:7}` $]\!] = \{$`x` $\mapsto Eq(7)\}$ and $[\![$ `{x:?, y:12}` $]\!] = \{$`x` $\mapsto All$, `y` $\mapsto Eq(12)\}$. However, entries like `s0:x` do not define a relation between two states in any obvious way but rather supply auxiliary information showing where on the stack the value of a given variable may be found. The trick is to define `s0:x` as a relation over an *abstract state*, i.e. program state under an abstract interpretation: `{s0:x}` $= \{0 \mapsto Eq(\texttt{x})\}$.

We have already come across the idea of language boundaries in §4.4 and §5.3.2. Suppose that we now want to mix execution of optimized SAL code and naively-compiled HL_{ANF} code; in this setup, a debug table becomes the boundary where execution can transfer between SAL and HL_{ANF}. Consider the table `{x:7, y:12}` in the example above: intuitively, it encodes the pre-conditions for the optimizations that were performed by the compiler. Any HL_{ANF} or SAL code can be spliced at the site of the debug table so long as, on exit, `x` can be assumed to contain 7 and `y` the value 12. Now, suppose we have the following SAL code for the command `x:=x+x`:

```
LOAD x; {s0:x} ; DUP; {s0:x, s1:x} ; ADD; {s0:x} ; STORE x;
```

Intuitively, each table defines a mapping from the SAL program state to an HL_{ANF} program state. The table is sufficient to generate *prologue* and *epilogue* "thunks" to be inserted at the point of the debug table: the thunks map from the optimized to the unoptimized program state (prologue) and back (epilogue). The prologue can be thought of as a de-instrumentation (§4.1.2) or a de-optimization (§5.7.2) function, but here, rather than being meta-language operations, the implementation is in SAL. For example, consider splicing the command `x:=4` just before the `ADD` in the example above: the prologue, body code and epilogue are, respectively, "`LOAD x; POP;`", "`IMM 4; STORE x;`" and "`LOAD x; DUP;`". Thus, a debug table can be thought of as defining a *calling convention* between optimized and unoptimized code.

6.4 Related work

Shamir and van Someren [117] suggested that

> "Given suitable tools we can present the [cryptographic] key as a constant in the computation which is carried out using that key and then we can optimise the code given that constant. This will cause the key to be intimately intertwined with the code which uses it."

The last sentence above we find particularly intriguing: to what extent is data "intimately intertwined with the code" by optimizing compilation? Though we are not aware of any existing work on information-flow security for program transformations, several authors have published on closely-related problems. Recent work by Dalla Preda and Giacobazzi [33] on shows that program obfuscation and program optimization are related phenomena.

It is well-known in folklore that low-level languages are more difficult to analyse than higher-level language. Logozzo and Fähndrich [81] describe a "precision loss" when analysing low-level bytecode versus source code, but existing literature on decompilation [6, 26, 30, 75, 86, 92, 58] tends to focus on specific techniques likely to be effective in practice as opposed to the fundamental limitations of decompilers.

We are also not aware of any formal account of the uses (and, more importantly, limits) of *dynamic analysis*—which instrumentation enables—for reverse engineering. Dalla Preda et al. [34] proposed a theoretical framework for the obfuscation vs. *static* analysis arms race. The observational power of the attacker attempting to reverse-engineer an obfuscated binary is determined by the static analyses at the attacker's disposal.

6.5 Conclusions

We have considered a notion of non-interference for compilers. The normalizing behaviour of optimizing compilers fits neatly into a model of information flow based on partial equivalence relations. It is immediate that *any* compiler for a Turing-complete language leaks more than just the equivalence class of its input programs. Decompilation—a translation which recovers the source of a program given its binary—provided a motivating example. In future work, we also plan to investigate application of information-flow security to program transformations used in software protection, such as obfuscation [13] and watermarking.

We have implicitly assumed that the attacker is interested in recovering the original source program. In practice, however, this is rarely the case: an attacker would be content with any "readable" source program. In their work on optimizing compilation, Tate et al. [130] introduced the *Program Expression Graph (PEG)*, an intermediate program representation that captures equivalences between programs. Compiler passes "saturate" the PEG with equalities, before a final selection is made based on a "profitability heuristic". This appears to be a sensible approach to decompilation as well: the "profitability heuristic" is readability and the final selection can be deferred to the user (equipped with a PEG manipulation tool).

Many non-trivial programs are language processors. Indeed, it was shown[4] that sendmail rewriting rules are Turing-complete! We conjecture that work on information-flow security for program transformations and translations could be applicable to many other kinds of programs.

[4] `http://okmij.org/ftp/Computation/index.html#sendmail-Turing`

Chapter 7

Conclusions and further work

The main contributions of this dissertation are: (*i*) a unified approach to efficiency in hardware virtualization and program instrumentation, based on Jones optimality, (*ii*) a scheme for static optimization of dynamically instrumented and dynamically updated programs by constraining the dependence of future instrumentation or updates on intermediate program states, and (*iii*) a semantic exposition of con-freeness, Stoyle et al.'s safety property for dynamically updatable software. We do not see these problems in isolation but as different facets of the concept of interpretational overhead.

Outline. In this chapter, we revisit the key themes of the dissertation (§7.1, §7.2 and §7.3) and suggest directions for future work (§7.4).

7.1 Common-case performance

Good performance is critical for many kinds of low-level software. Many such systems—hardware virtualization, program instrumentation and dynamic software updating among others—are most directly understood in terms of non-standard interpretation. For example, the similarity between the operation of a VMM and an instrumented interpreter are apparent: both alter the meaning of specific language phrases. Several performance requirements have been proposed for system software, but both the formal criterion for VMMs due to Popek and Goldberg [102], and the informal assurance given by Cantrill et al. [22] for their DTrace instrumentation framework emphasize the *common case*:

Application	Common case
Virtualization	Execution of unprivileged instructions
Program instrumentation	Instrumentation is disabled
Dynamic software updating	No updates are applied

For a VMM, execution of unprivileged instruction is the common case, whereas privileged instructions occur rarely in reasonable programs, so it is acceptable to penalise execution of privileged instructions if this leads to better performance for the rest. Similarly, it is reasonable to make the process of adding and removing instrumentation or applying a dynamic software update costly, if this improves performance in the common case. The non-standard interpretation matches the standard one *most of the time*, and therefore we would like to maximally exploit the hardware which was built with the standard interpretation in mind.

7.2 Indirection without a performance penalty

There is arguably little difference between a language and a programming interface of a library or the system call interface of an operating system kernel: indeed, domain-specific languages are frequently designed to enforce the usage requirements of a programming interface (e.g. `open` must be called before `close`). This similarity is especially apparent with the aid of the context threading table (CTT), a program representation used by context-threaded interpreters [18] (see 3.2), which substitutes instructions in the program with calls to interpreter routines that implement those instructions:

AL code	**CTT**	(Operands)
`MOV  r1, 3`	`CALL doMOVri`	`1, 3`
`ADD  r2, r1, 5`	`CALL doADDrri`	`2, 1, 5`
`OUT <0>, r2`	`CALL doOUTir`	`0, 2`

A collection of `do` procedures constitutes the programming interface of an interpreter. Therefore, a CTT can be thought of simply as client code, making use of this interface, or, alternatively, as a *domain-specific embedded language (DSEL)*: not a separate language in its own right, but rather a “sub-language” of the *host* (AL, in this case) whose programs have a particular structure. In host languages with more sophisticated type systems, this structure can often be enforced with types, as in the `printf` typing example in §2.1.2. Engler et al. [40] similarly use programmer-written compiler extensions to enforce domain-specific constraints in system code. However, the focus in the literature has been primarily on correctness rather than performance guarantees.

In practice, a programming language is intimately tied to its optimizing compiler. Abstraction features in high-level languages (virtual dispatch, function pointers) are often perceived as costly, because in many cases they impede optimization. Indeed, programmers are liable to avoid using these features, to the detriment of readability, code quality etc. However, the programmer’s intuitions and expectation about the compiler’s ability to, e.g., optimize virtual calls into static calls, are borne out of experience rather than any formal assurance. Veldhuizen and Lumsdaine [134] came up with the idea of *guaranteed optimization*: the compiler promises that some abstraction mechanisms are “free”, i.e. the indirection is always optimized away. Since a non-standard interpretation introduces a layer of indirection, we can draw a parallel from this line of work to ours. The common-case requirements that have been proposed—by Popek and Goldberg and Cantrill et al.—stipulate that the overhead of indirection *must* be eliminated, whether by optimization (as in binary translation VMMs) or through additional hardware support. Documentation for the VProbes [136] instrumentation framework uses the telling phrase “free when disabled”. Serendipitously, this is very much in the spirit of the Jones optimality criterion in partial evaluation. Recall from Chapter 2 that a program specializer is called Jones-optimal if it is capable of removing a layer of interpretational overhead completely:

$$\exists sint.\ \forall p. \quad [\![mix]\!](sint, p) =_{\alpha} p\ .$$

In other words, if we have a Jones-optimal specializer, we can introduce a layer of interpretation for “free” as long as we use a self-interpreter that does not implement any custom behaviour like, for example, instrumentation (the “disabled” case). We have generalised and extended the definitions to handle partial removal of overhead (“most of the time”): remember that a VMM must still emulate some instructions.

7.3 Architectural support motivated by optimization

Programming language design is often dictated by architectural constraints of the target machine. Conversely, machine designs and instruction sets expand to provide better support for important language features: for example, virtualization extensions, Transmeta's Code Morphing [78] technology and branch-prediction buffers all provide hardware support for non-standard interpretation, albeit in slightly different ways. Co-evolution of language and machine is clearly beneficial to the end-user (at least in terms of raw performance improvement) but does not necessarily lead to an optimal design for either language or machine in the long run: seen as an optimisation process, it is more likely to lead to a local rather than a global maximum. In the functional programming languages community there is a tradition of deriving abstract machines from the semantics of a higher-level language [35, 7] and we have attempted to emulate this approach in several places in this dissertation (§3.2, §4.3.2, §4.4) to recover trap-and-emulate support, breakpoints and super-instructions from a more basic setup.

7.4 Further work

There are two broad areas of further work that follow on naturally from this dissertation.

First, modern runtime environments use a mixture of interpretation and direct execution. This includes emulators, language virtual machines (like the JVM or .NET), instrumentation and runtime verification tools, and virtual machine monitors. However, we believe that semantics and, perhaps more importantly, the design, of low-level languages (and low-level domain-specific languages) is an area that has not received sufficient attention. As an example of a worthwhile goal, consider a dedicated typed assembly language, with support for program staging, specifically intended for writing virtual machines. Crucially, the compiler, aware of the meaning of the special primitives in the language, can do a better job at optimizing the resulting code. For communication-centric hardware, the devices are as much part of the language as the CPU. The meaning of an `IN r1, <42>` instruction depends on what device is attached to I/O port 42. Indeed, it is not uncommon to have a processor on-board a system device that is no less important than the CPU: e.g. GPUs, specialised network and RAID processors, FPGAs and even general-purpose CPUs (on the SunPCi series of x86 coprocessor cards[1]).

Virtualization, instrumentation and sandboxing are all special cases of a general class of systems that function by "slightly" modifying the way a program is executed. Implementation of such tools is especially challenging (see e.g. [109, 99, 150]) on x86 processors due to presence of self-modifying code (von Neumann vs. Harvard architecture) exacerbated by a CISC instruction set (vs. fixed-length encoding RISC), where merely disassembling the binary is a difficult problem due to possibly-overlapping instruction sequences. At the same time, hardware support for non-standard interpretation of this kind is meagre and, where it exists at all, is highly specialised: such as, for example, Intel's VT virtualization assists [97] (or even Virtual 8086 mode!). Indeed, some opcodes are already executed differently depending on the mode the machine is in: 32-bit vs. 64-bit mode or real vs. protected mode. Virtualization assists and single-step execution support are point solutions to the general problem of providing hardware support for non-standard interpretation. Unified architectural support for non-standard interpretation has many

[1] `http://www.sun.com/desktop/products/sunpcipro/`

benefits over and above that of simplifying implementation of debuggers, virtual machines and other similar tools.

Appendix A

Notation

`MOV`	AL instruction (§2.3)
x	variable
e	expression
c	command
p	program
t	IL script (§4.1)
$\mathcal{I}(t)$	AL interpreter with instrumentation according to t
v	value
s	state
pc	program counter (instruction pointer)
k	continuation
$=_\alpha$	α-equivalence
$\sim_L$	a semantic equivalence relation for language L
$\mathsf{Ctx}[-]$	a one-hole context (§2.1.1)
$[\![\cdot]\!]_L$	valuation function for L
$\mathcal{I}[\![\cdot]\!]_L$	instrumented valuation function for L (§2.6)
int_M^L	interpreter for M written in L (§2.2)
$sint_L$	self-interpreter for L
mix	partial evaluator
$\mathcal{C}$, $\mathcal{D}$	compiler and decompiler
P, Q, ...	partial equivalence relations (PERs) over X, Y, ... (§2.7)
$P \times Q$	PER over $X \times Y$
$P + Q$	PER over $X + Y$
$P \Rightarrow Q$	PER over $X \to Y$

Appendix B

Applications of PERs

B.1 Types

A semantics of types is said to be *extrinsic* if it is defined over an untyped model of the language[1]. One approach is to interpret types as PERs over the universal domain. This works for both the simply-typed and polymorphic λ-calculi among others—see Gunter's [61, pp. 266 and 374] and Reynolds' [106, pp. 327 and 390] textbooks for details. We will briefly recap the simply-typed case here: suppose we have types $\tau ::= \mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \tau \to \tau$ and a call-by-value pre-CPO $V \cong \mathbf{1} + \mathbb{Z} + (V \to V_\perp)$. With injections omitted, the meaning $[\![\tau]\!]$ of a type τ is a PER on V defined as follows:

$$\begin{array}{rcl} [\![\mathsf{unit}]\!] & = & \{(\texttt{()},\ \texttt{()})\} \\ [\![\mathsf{int}]\!] & = & Id_{\mathbb{Z}} \\ [\![\mathsf{bool}]\!] & = & \{(z,\ z') \mid z \neq 0 \text{ and } z' \neq 0\} \cup \{(0,\ 0)\} \\ [\![\tau_1 \to \tau_2]\!] & = & [\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]_\perp . \end{array}$$

The denotation of an expression e is $[\![e]\!] \in \mathbf{Env} \to V_\perp$ where $\mathbf{Env} = Var \to V$ is the CPO of variable environments. Two environment $\rho, \rho' \in \mathbf{Env}$ are related under a typing context Γ, written $\rho \sim_\Gamma \rho'$, iff $\forall x \in dom\ \Gamma.\ \rho(x)\ [\![\Gamma(x)]\!]\ \rho'(x)$. An equational typing judgement $\Gamma \vdash e = e' : \tau$ is valid iff $[\![e]\!]\ (\sim_\Gamma \Rightarrow [\![\tau]\!])\ [\![e']\!]$. A relational semantics of types is more directly useful to the compiler writer than a syntactic type discipline with a "progress and preservation" result. To see why, consider that the expression $e_1 = \lambda x.\mathbf{if}\ x > 0\ \mathbf{then}\ 42\ \mathbf{else}\ 0$ can replaced by $e_2 = \lambda x.x$ in a program context where its value is used as a boolean but *not* when it is used as an integer. This is directly reflected in the PER semantics: in any typing context Γ the two expressions are equivalent at type $\mathsf{int} \to \mathsf{bool}$, i.e. $\Gamma \vdash e_1 = e_2 : \mathsf{int} \to \mathsf{bool}$, but *not* at $\mathsf{int} \to \mathsf{int}$. It is also worth pointing out now that PERs which are *not* interpretations of any of the types in the language are often the more interesting ones: e.g. $True = \{(z,\ z') \mid z > 0 \text{ and } z' > 0\}$. For instance, $[\![e_1]\!]\ (\sim_\Gamma \Rightarrow (True \Rightarrow [\![\mathsf{int}]\!]))\ [\![\lambda x.42]\!]$, so in a program context where we know that x is true (say, "$e_1\ 7$") the **if**-expression is semantically equivalent to its **then**-branch.

For polymorphic calculi, Wadler demonstrated the utility of Reynolds' *abstraction theorem* [105] in "Theorems for Free!" [138]. In a nutshell, expressions inhabiting a particular type obey certain algebraic laws the validity of which can be proved using the relational semantics of the type alone. The same approach works equally well in an imperative setting[2]. Benton [15] built an equational theory to support proofs of optimizing program

[1] Versus *intrinsic* where the language only assigns meaning to well-typed terms.

[2] And why shouldn't it? The meta-language (continuous functions over CPOs) is still the same.

transformations on top of a relational semantics for a **While** language. In the same paper, Benton also introduces *relational Hoare logic*. Recall that, in Hoare logic,

$$[\![\{p\}\ c\ \{q\}]\!] = \forall s \in \mathit{State}.\ [\![p]\!]\ s \Longrightarrow ([\![c]\!]\ s = \bot \vee [\![q]\!]\ ([\![c]\!]\ s))$$

where $s \in \mathit{State}$ is a state, $[\![p]\!], [\![q]\!] \in s \to \mathbb{B}$ the meanings of pre- and post- conditions p and q respectively, and $[\![c]\!] \in s \to s_\bot$ the denotation of command c. For some given p and q we can construct relations $P \stackrel{\text{def}}{=} \{(s,\ s') \mid [\![p]\!]\ s \wedge [\![p]\!]\ s'\}$ and Q (analogously), and see that $\{p\}\ c\ \{q\}$ iff $[\![c]\!] : P \Rightarrow Q_\bot$.

So far we have looked at cases where use of PERs—to give a "static semantics on steroids"—is convenient but not essential. Milner famously noted that "well-typed programs don't go wrong". The traditional approach is then to define "wrong" as a stuck state in the dynamic semantics of the language and prove a syntactic "progress and preservation" theorem that guarantees that no well-typed program ever gets stuck. A type system like this captures a *safety* property that corresponds to a set of program traces and can be enforced by an execution monitor. Volpano [137] and Schneider et al. [115] show that *secrecy* cannot be defined in this way. Indeed secrecy is just one example of a host of similar properties all of which share a notion of *dependency*. Abadi et al. [4] developed the Dependency Core Calculus (DCC) as a common language capable of expressing all of these. The semantics of DCC is given in a category where the objects are CPOs and relations over them and a morphism $f : A \to B$ must be in the domain of $(R_A \Rightarrow R_B)$.

B.2 Non-interference

In information-flow security, non-interference is usually defined as as zero information flow from high-security inputs to low-security outputs. Let c be a command and s_1, s_2 be starting states consisting of bindings for low- and high-security variables. Let $[\![\cdot]\!] : \mathit{State} \to \mathit{State}_\bot$ be the evaluation function. Non-interference is satisfied whenever

$$s_1\ P\ s_2 \Longrightarrow [\![c]\!]\ s_1\ Q\ [\![c]\!]\ s_2 \qquad \text{(B.1)}$$

where two states are equivalent up to P when their low-security parts are equal. The relation Q captures the "observational power of an attacker" [112]. Sabelfeld and Sands [113] gave a presentation of non-interference in terms of PERs (including non-deterministic information flow using PERs over power-domains) and formalized its relationship to Joshi and Leino's [74] equational characterization (§6.2.2). For example, let $[\![c]\!] : (\mathbb{Z} \times \mathbb{Z}) \to (\mathbb{Z} \times \mathbb{Z})_\bot$ where the first element of each pair is the value of the single low-security variable and the second element the value of the high-security variable. Then c has zero information flow iff

$$[\![c]\!] : (\mathit{Id} \times \mathit{All}) \Rightarrow (\mathit{Id} \times \mathit{All})_\bot\ .$$

Hunt and Mastroeni [69] compare the PER model of non-interference to *abstract non-interference* of Giacobazzi and Mastroeni [55], which is a generalization of non-interference based on abstract interpretation.

B.3 Static analyses

We will describe the use of CPO *projections* and PERs for formalizing *strictness* and *binding-time* analyses. Recall that a continuous function $\alpha : D \to D$ on a CPO D is

$$\begin{array}{lll} \text{a } \textit{retraction} & \text{iff} & \alpha \circ \alpha = \alpha, \\ \text{a } \textit{projection} & \text{iff} & \alpha \circ \alpha = \alpha \text{ and } \alpha \sqsubseteq \mathsf{id}_D \ . \end{array}$$

Given a function $f = [\![e]\!]$, a denotation of some expression e, the goal of *projection* (or *context*) analysis is to solve the equation $\alpha \circ f \circ \beta = \alpha \circ f$ for projections α and β. The intuition here is that β describes the way f uses its arguments in a *context* α. The idea is due to Wadler and Hughes [142] who first applied it to strictness analysis: recall that, in a lazy language, an expression e is said to be "strict" in its (sole) argument iff $[\![e]\!] \perp = \perp$. Hunt [68] showed that PERs subsume projections for strictness analysis.

Let e be an expression with type $\tau_1 \to \ldots \to \tau_n$. Each formal argument to e is further annotated with S (for "static") or D (for "dynamic"). A *binding-time analysis* (BTA) determines whether the value of e is independent of the values of its dynamic arguments. For example, let $e = \lambda x : \mathsf{int}^{\mathsf{S}}.\ \lambda y : \mathsf{int}^{\mathsf{D}}.\ x$ and note that $\forall x.\ \forall y, y'.\ e\ x\ y = e\ x\ y'$, that is, $e : \mathit{Id}_{\mathbb{Z}} \Rightarrow \mathit{All}_{\mathbb{Z}} \Rightarrow \mathit{Id}_{\mathbb{Z}}$. Hunt and Sands [70] observed the correspondence between *Id* and static arguments and *All* and dynamic ones and gave a PER semantics to an abstract interpretation for binding-time analysis. A concretisation map at type τ, γ_τ, assigns to each point a in the abstract domain a PER $D^{\#}_{\tau}$ over the concrete domain D_τ. Soundness is expressed as:

$$[\![e]\!]^{\#}\ a_1\ \ldots\ a_n = b \quad \Longrightarrow \quad [\![e]\!] : \gamma_{\tau_1}(a_1) \Rightarrow \ldots \Rightarrow \gamma_{\tau_n}(a_n) \Rightarrow \gamma_\tau(b)\ .$$

For instance, the PER corresponding to the top (*dynamic*) element of $D^{\#}_{\mathsf{int}\to\mathsf{int}}$ is $\mathit{All}_{D_{\mathsf{int}\to\mathsf{int}}}$, whereas that for the bottom (*static*) is $\mathit{All}_{D_{\mathsf{int}}} \Rightarrow \mathit{Id}_{D_{\mathsf{int}}}$.

The authors also note the similarity of binding-time analysis to live variable analysis, a classical data flow analysis [100] usually presented as a backwards analysis: given a set of variables that are live at the end of the program (observable outputs), the analysis determines, for every statement, a conservative approximation of the set of variables that are live at the end of it. Suppose we restrict the state of our programs to two integer variables: $[\![\cdot]\!] \in \mathbb{Z} \times \mathbb{Z} \to (\mathbb{Z} \times \mathbb{Z})_\perp$. When c terminates the first variable will contain its output. The second variable is auxiliary and we do not care about its value, so we can optimize c to c' as long as $[\![c]\!]\ (\mathit{Id}_{\mathbb{Z}} \Rightarrow (\mathit{Id}_{\mathbb{Z}} \times \mathit{All}_{\mathbb{Z}})_\perp)\ [\![c']\!]$.

Bibliographical abbreviations

APLAS	Asian Symposium on Programming Languages and Systems
ASPLOS	International Conference on Architectural Support for Programming Languages and Operating Systems
CC	International Conference on Compiler Construction
CGO	Symposium on Code Generation and Optimization
ECOOP	European Conference on Object-Oriented Programming
ENTCS	Electronic Notes in Theoretical Computer Science
ESOP	European Symposium on Programming
FPCA	Conference on Functional Programming Languages and Computer Architecture
HotOS	Workshop on Hot Topics in Operating Systems
ICALP	International Colloquium on Automata, Languages and Programming
ICFP	International Conference on Functional Programming
LFP	ACM Conference on LISP and Functional Programming
LICS	Symposium on Logic in Computer Science
LNCS	Lecture Notes in Computer Science
OOPSLA	Conference on Object-Oriented Programming Systems, Languages, and Applications
OSDI	Symposium on Operating Systems Design and Implementation
PEPM	Symposium/Workshop on Partial Evaluation and Semantic-Based Program Manipulation
PLDI	Conference on Programming Language Design and Implementation
POPL	Symposium on Principles of Programming Languages
PPDP	International Conference on Principles and Practice of Declarative Programming
SAS	Static Analysis Symposium
SOSP	Symposium on Operating System Principles
TOPLAS	Transactions on Programming Languages and Systems
VEE	International Conference on Virtual Execution Environments

Bibliography

[1] Martín Abadi. Protection in programming-language translations. In *Proceedings of ICALP*, volume 1443 of *LNCS*, pages 868–883. Springer, 1998.

[2] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *TOPLAS*, 13(2):237–268, 1991.

[3] Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, 1995.

[4] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of POPL*, pages 147–160. ACM, 1999.

[5] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of ASPLOS*, pages 2–13. ACM, 2006.

[6] Mads Sig Ager, Olivier Danvy, and Mayer Goldberg. A symmetric approach to compilation and decompilation. In *The Essence of Computation*, volume 2566 of *LNCS*, pages 296–331. Springer, 2002.

[7] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical Report RS-03-14, BRICS, March 2003.

[8] Tristan O. R. Allwood, Simon Peyton Jones, and Susan Eisenbach. Finding the needle: stack traces for GHC. In *Proceedings of ACM Symposium on Haskell*, pages 129–140. ACM, 2009.

[9] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[10] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of LICS*. IEEE Computer Society, 2001.

[11] Jeff Arnold and M. Frans Kaashoek. Ksplice: automatic rebootless kernel updates. In *Proceedings of EuroSys*, pages 187–198. ACM, 2009.

[12] Lennart Augustsson. Cayenne — a language with dependent types. In *Proceedings of ICFP*, pages 239–250. ACM, 1998.

[13] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the International Cryptology Conference*, volume 2139 of *LNCS*, pages 1–18. Springer, 2001.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of SOSP*, pages 164–177. ACM, 2003.

[15] Nick Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of POPL*, pages 14–25. ACM, 2004.

[16] Nick Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4):503–542, 2005.

[17] Nick Benton and Uri Zarfaty. Formalizing and verifying semantic type soundness of a simple compiler. In *Proceedings of PPDP*, pages 1–12. ACM, 2007.

[18] Marc Berndl, Benjamin Vitale, Mathew Zaleski, and Angela Demke Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Proceedings of CGO*, pages 15–26. IEEE Computer Society, 2005.

[19] Gavin M. Bierman, Michael W. Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-timeλ. In *Proceedings of ICFP*, pages 99–110. ACM, 2003.

[20] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of OOPSLA*, pages 331–344. ACM, 2004.

[21] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. In *Proceedings of VEE*, pages 137–147. ACM, 2007.

[22] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference*, pages 15–28. USENIX Association, 2004.

[23] L. Cardelli. Phase distinctions in type theory. Manuscript, 1988. `http://lucacardelli.name/Papers/PhaseDistinctions.pdf`.

[24] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

[25] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of PLDI*, pages 278–292. ACM, 1991.

[26] Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula. Analysis of low-level code using cooperating decompilers. In *Proceedings of SAS*, volume 4134 of *LNCS*, pages 318–335. Springer, 2006.

[27] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of VEE*, pages 35–44. ACM, 2006.

[28] Anton Chernoff and Ray Hookway. DIGITAL FX!32: Running 32-bit x86 applications on Alpha NT. In *Proceedings of the USENIX Workshop on Windows NT*, pages 9–16. USENIX Association, 1997.

[29] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proceedings of SAS*, volume 2694 of *LNCS*, pages 1–18. Springer, 2003.

[30] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Proceedings of the International Conference on Software Maintenance*, pages 228–237. IEEE, 1998.

[31] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantitative information flow, relations and polymorphic types. *J. Log. Comput*, 15(2):181–199, 2005.

[32] Frederick B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[33] Mila Dalla Preda and Roberto Giacobazzi. Semantic-based code obfuscation by abstract interpretation. In *Proceedings of ICALP*, volume 3580 of *LNCS*, pages 1325–1336. Springer, 2005.

[34] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya K. Debray. A semantics-based approach to malware detection. In *Proceedings of POPL*, pages 377–388. ACM, 2007.

[35] Olivier Danvy. A rational deconstruction of Landin's SECD machine. Technical Report RS-03-33, BRICS, October 2003.

[36] Olivier Danvy and Pablo E. Martínez López. Tagging, encoding, and Jones optimality. In *Proceedings of ESOP*, volume 2618 of *LNCS*, pages 335–347. Springer, 2003.

[37] Søren Debois. Imperative-program transformation by instrumented-interpreter specialization. *Higher-Order and Symbolic Computation*, 21(1-2):37–58, 2008.

[38] David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of ECOOP*, volume 1628 of *LNCS*, pages 258–278. Springer, 1999.

[39] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the International Conference on Embedded Software*, pages 255–264. ACM, 2008.

[40] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of OSDI*, pages 1–16. USENIX Association, 2000.

[41] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of PLDI*, pages 278–288. ACM, 2003.

[42] Boris Feigin and Alan Mycroft. Decompilation is an information-flow problem, 2008. URL `http://clip.dia.fi.upm.es/Conferences/PLID08/slides/boris.pdf`. Presented at the International Workshop on Programming Language Interference and Dependence, Valencia, Spain.

[43] Boris Feigin and Alan Mycroft. Jones optimality and hardware virtualization: a report on work in progress. In *Proceedings of PEPM*, pages 169–175. ACM, 2008.

[44] Peter Ferrie. Attacks on virtual machine emulators. Symantec Advanced Threat Research, 2006. URL `http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf`.

[45] Robert Bruce Findler and Matthias Blume. Contracts as pairs of projections. In *Proceedings of the International Symposium on Functional and Logic Programming*, volume 3945 of *LNCS*, pages 226–241. Springer, 2006.

[46] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of ICFP*, pages 48–59. ACM, 2002.

[47] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of CGO*, pages 241–252. IEEE Computer Society, 2003.

[48] Cormac Flanagan. Hybrid type checking. In *Proceedings of POPL*, pages 245–256. ACM, 2006.

[49] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *Proceedings of HotOS*, pages 67–72. IEEE Computer Society Press, 1997.

[50] Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of LFP*, pages 348–355. ACM, 1984.

[51] Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. In *Proceedings of PLDI*, pages 62–72. ACM, 2005.

[52] Michael Furr and Jeffrey S. Foster. Polymorphic type inference for the JNI. In *Proceedings of ESOP*, volume 3924 of *LNCS*, pages 309–324. Springer, 2006.

[53] Johan Gade and Robert Glück. On Jones-optimal specializers: A case study using Unmix. In *Proceedings of APLAS*, volume 4279 of *LNCS*, pages 406–422. Springer, 2006.

[54] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of HotOS*. USENIX Association, 2007.

[55] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of POPL*, pages 186–197. ACM, 2004.

[56] Robert Glück. The translation power of the Futamura projections. In *Perspectives of Systems Informatics*, volume 2890 of *LNCS*, pages 133–147. Springer, 2003.

[57] Mayer Goldberg. Gödelization in the lambda calculus. *Inf. Process. Lett*, 75(1-2):13–16, 2000.

[58] Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. Improving the decompilation of Java bytecode to Prolog by partial evaluation. *ENTCS*, 190(1):85–101, 2007.

[59] Kathryn E. Gray, Robert Bruce Findler, and Matthew Flatt. Fine-grained interoperability through mirrors and contracts. In *Proceedings of OOPSLA*, pages 231–245. ACM, 2005.

[60] David Gries. *Compiler Construction for Digital Computers*. John Wiley, 1971.

[61] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press, 1992.

[62] Robert Harper and J. Gregory Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of POPL*, pages 130–141. ACM, 1995.

[63] William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *Proceedings of the International Conference on Computer Languages*, pages 122–131. IEEE, 1998.

[64] M. Hicks, S. Tse, B. Hicks, and S. Zdancewic. Dynamic updating of information-flow policies. In *Proceedings of the Workshop on Foundations of Computer Security*, 2005. URL `http://www.cse.chalmers.se/~andrei/FCS05/`.

[65] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of PLDI*, pages 13–23. ACM, 2001.

[66] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of PLDI*, pages 32–43. ACM, 1992.

[67] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[68] Sebastian Hunt. PERs generalise projections for strictness analysis (extended abstract). In *Proceedings of Glasgow Workshop on Functional Programming*. Springer, 1991.

[69] Sebastian Hunt and Isabella Mastroeni. The PER model of abstract non-interference. In *Proceedings of SAS*, volume 3672 of *LNCS*, pages 171–185. Springer, 2005.

[70] Sebastian Hunt and David Sands. Binding time analysis: A new PERspective. In *Proceedings of PEPM*, pages 154–165. ACM, 1991.

[71] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. *New Generation Comput.*, 6(2&3):291–302, 1988.

[72] Neil D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52:307–339, 2004.

[73] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[74] Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Science of Computer Programming*, 37(1–3):113–138, 2000.

[75] Shin-ya Katsumata and Atsushi Ohori. Proof-directed de-compilation of low-level code. In *Proceedings of ESOP*, volume 2028 of *LNCS*, pages 352–366. Springer, 2001.

[76] Andrew Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3):311–317, 2006.

[77] Amir Kishon and Paul Hudak. Semantics directed program execution monitoring. *Journal of Functional Programming*, 5(4):501–547, 1995.

[78] Alexander Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corporation, 2000.

[79] Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of USENIX Security Symposium*, pages 255–270. USENIX Association, 2004.

[80] Sheng Liang, Paul Hudak, and Mark P. Jones. Monad transformers and modular interpreters. In *Proceedings of POPL*, pages 333–343. ACM, 1995.

[81] Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *Proceedings of CC*, volume 4959 of *LNCS*, pages 197–212. Springer, 2008.

[82] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In *Proceedings of the International Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 1924 of *LNCS*, pages 129–148. Springer, 2000.

[83] Henry Massalin. Superoptimizer – a look at the smallest program. In *Proceedings of ASPLOS*, pages 122–126. ACM, 1987.

[84] Jacob Matthews and Robert Bruce Findler. Operational semantics for multi-language programs. In *Proceedings of POPL*, pages 3–10. ACM, 2007.

[85] Microsoft Corporation. Using hotpatching technology to reduce servicing reboots, 2010. URL `http://technet.microsoft.com/en-us/library/cc787843(WS.10).aspx`. Accessed on 19 March 2010.

[86] Jerome Miecznikowski and Laurie J. Hendren. Decompiling Java bytecode: problems, traps and pitfalls. In *Proceedings of CC*, volume 2304 of *LNCS*, pages 111–127. Springer, 2002.

[87] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[88] John C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141–163, 1993.

[89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of LICS*, pages 14–23. IEEE Computer Society, 1989.

[90] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93 (1):55–92, July 1991.

[91] J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of POPL*, pages 85–97. ACM, 1998.

[92] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of ESOP*, volume 1576 of *LNCS*, pages 208–223. Springer, 1999.

[93] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 14(2):157–196, 2006.

[94] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of POPL*, pages 37–49. ACM, 2008.

[95] George C. Necula. Proof-carrying code. In *Proceedings of POPL*, pages 106–119. ACM, 1997.

[96] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of PLDI*, pages 333–344. ACM, 1998.

[97] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, 2006. `http://dx.doi.org/10.1535/itj.1003.01`.

[98] Nicholas Nethercote and Alan Mycroft. Redux: A dynamic dataflow tracer. *ENTCS*, 89 (2), 2003.

[99] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of PLDI*, pages 89–100. ACM, 2007.

[100] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

[101] Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. In *Proceedings of IFIP TC1 3rd International Conference on Theoretical Computer Science*, pages 437–450. Kluwer, 2004.

[102] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[103] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In *Proceedings of 10th International Conference on Information Security*, volume 4779 of *LNCS*, pages 1–18. Springer, 2007.

[104] Norman Ramsey. Embedding an interpreted language using higher-order functions and types. In *Proceedings of the ACM SIGPLAN Workshop on Interpreters, Virtual Machines and Emulators*, pages 6–14. ACM, 2003.

[105] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Paris, France*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[106] John C. Reynolds. *Theories of programming languages.* Cambridge University Press, 1998.

[107] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprint of a 1972 paper.

[108] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of LICS*, pages 55–74. IEEE Computer Society, 2002.

[109] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of USENIX Security Symposium.* USENIX Association, 2000.

[110] Timothy Roscoe, Kevin Elphinstone, and Gernot Heiser. Hype and virtue. In *Proceedings of HotOS.* USENIX Association, 2007.

[111] Ben Rudiak-Gould, Alan Mycroft, and Simon Peyton Jones. Haskell is not not ML. In *Proceedings of ESOP*, volume 3924 of *LNCS*, pages 38–53. Springer, 2006.

[112] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.

[113] Andrei Sabelfeld and David Sands. A Per model of secure information flow in sequential programs. In *Proceedings of ESOP*, volume 1576 of *LNCS*, pages 40–58. Springer, 1999.

[114] Andrei Sabelfeld and David Sands. Declassification: dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

[115] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In *Informatics. 10 Years Back. 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101. Springer, 2001.

[116] Peter Sestoft. Replacing function parameters by global variables. In *Proceedings of FPCA*, pages 39–53. ACM, 1989.

[117] Adi Shamir and Nicko van Someren. Playing 'hide and seek' with stored keys. In *Proceedings of Financial Cryptography*, volume 1648 of *LNCS*, pages 118–124. Springer, 1999.

[118] Zhong Shao. Typed common intermediate format. In *Proceedings of the Conference on Domain-Specific Languages*, pages 89–102. USENIX, 1997.

[119] Mark Shields, Tim Sheard, and Simon L. Peyton Jones. Dynamic typing as staged type inference. In *Proceedings of POPL*, pages 289–302. ACM, 1998.

[120] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Proceedings of Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[121] Jeremy G. Siek and Walid Taha. Gradual typing for objects. In *Proceedings of ECOOP*, volume 4609 of *LNCS*, pages 2–27. Springer, 2007.

[122] Jeffrey Mark Siskind and Barak A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *Proceedings of POPL*, pages 71–76. ACM, 2007.

[123] Brian Cantwell Smith. Reflection and semantics in Lisp. In *Proceedings of POPL*, pages 23–35. ACM, 1984.

[124] James E. Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[125] Guy L. Steele, Jr. Building interpreters by composing monads. In *Proceedings of POPL*, pages 472–492. ACM, 1994.

[126] Gareth Stoyle, Michael W. Hicks, Gavin M. Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Proceedings of POPL*, pages 183–194. ACM, 2005.

[127] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a VM-centric approach. In *Proceedings of PLDI*, pages 1–12. ACM, 2009.

[128] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *Proceedings of PEPM*, pages 203–217. ACM, 1997.

[129] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of OSDI*, pages 117–130. USENIX Association, 1999.

[130] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of POPL*, pages 264–276. ACM, 2009.

[131] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[132] Andrew P. Tolmach and Andrew W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[133] Valery Trifonov and Zhong Shao. Safe and principled language interoperation. In *Proceedings of ESOP*, volume 1576 of *LNCS*, pages 128–146. Springer, 1999.

[134] Todd L. Veldhuizen and Andrew Lumsdaine. Guaranteed optimization: Proving nullspace properties of compilers. In *Proceedings of SAS*, volume 2477 of *LNCS*, pages 263–277. Springer, 2002.

[135] Dennis Vermoen, Marc F. Witteman, and Georgi Gaydadjiev. Reverse engineering Java Card applets using power analysis. In *Proceedings of the First IFIP TC6 / WG 8.8 / WG 11.2 International Workshop on Information Security Theory and Practices*, volume 4462 of *LNCS*, pages 138–149. Springer, 2007.

[136] VMware, Inc. VProbes programming reference, 2010. URL `http://www.vmware.com/pdf/ws7_f3_vprobes_reference.pdf`. Accessed on 23 August 2010.

[137] Dennis M. Volpano. Safety versus secrecy. In *Proceedings of SAS*, volume 1694 of *LNCS*, pages 303–311. Springer, 1999.

[138] Philip Wadler. Theorems for free! In *Proceedings of FPCA*, pages 347–359. ACM, 1989.

[139] Philip Wadler. The essence of functional programming. In *Proceedings of POPL*, pages 1–14. ACM, 1992.

[140] Philip Wadler. The marriage of effects and monads. In *Proceedings of ICFP*, pages 63–74. ACM, 1998.

[141] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of ESOP*, volume 5502 of *LNCS*, pages 1–16. Springer, 2009.

[142] Philip Wadler and R. J. M. Hughes. Projections for strictness analysis. In *Proceedings of FPCA*, volume 274 of *LNCS*, pages 385–407. Springer, 1987.

[143] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 52–65. IEEE Computer Society, 1998.

[144] Mitchell Wand. The theory of fexprs is trivial. *Lisp and Symbolic Computation*, 10(3): 189–199, 1998.

[145] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. In *Proceedings of LFP*, pages 298–307. ACM, 1986.

[146] Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information-flow. In *Proceedings of LICS*, pages 62–71. IEEE Computer Society, 2005.

[147] John Whaley. System checkpointing using reflection and program analysis. In *Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001, Kyoto, Japan, September 25-28, 2001, Proceedings*, volume 2192 of *LNCS*, pages 44–51. Springer, 2001.

[148] Glynn Winskel. *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[149] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of POPL*, pages 214–227. ACM, 1999.

[150] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: a sandbox for portable, untrusted x86 native code. *Communications of the ACM*, 53(1):91–99, 2010.

[151] Mathew Zaleski, Marc Berndl, and Angela Demke Brown. Mixed mode execution with context threading. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research*, pages 305–319, Toronto, Ontario, Canada, October 2005. IBM.

[152] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, page 15. IEEE Computer Society, 2001.