Extensible Untrusted Code Verification

by

Robert Richard Schneck

B.S. (Duke University) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Logic and the Methodology of Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor George Necula, Chair
Professor Ras Bodik
Professor Thomas Henzinger
Professor Christos Papadimitriou
Professor Jack Silver

Spring 2004

Abstract

Extensible Untrusted Code Verification

by

Robert Richard Schneck
Doctor of Philosophy in Logic and the Methodology of Science

University of California, Berkeley

Professor George Necula, Chair

Various mechanisms exist for enforcing that untrusted code satisfies basic but essential security properties, such as memory safety. A security mechanism needs to be trustworthy, so that users can feel secure that the mechanism can not in some way be tricked by malicious or erroneous code. It is increasingly important that security mechanisms be flexible enough to handle software systems written in more than one language. Finally a security mechanism must be a practical and usable tool; it must scale to handle realistic software systems. Standard security enforcement techniques using intermediate languages run on virtual machines are disappointing; in particular, the intermediate languages are too fixed to handle a wide variety of source languages in a natural way, even when the intermediate language is designed with flexibility in mind.

In this dissertation I propose a security enforcement mechanism called the Open Verifier. The Open Verifier allows a producer of untrusted code to include with the code an untrusted verifier called an extension. The trusted framework of the Open Verifier works together with the untrusted extension to produce a complete trustworthy verification. The code producer can tailor the extension to the particular source language and compilation strategy used to produce the untrusted code, ensuring the flexibility of the system. At the same time the trusted framework is kept reasonably simple and small, and easy to trust.

In order to produce a trustworthy verification from an untrusted verifier, the extension is required to emit intermediate results which can be checked by the trusted components of the system. In fact, the extension must produce the proofs of obligations produced by the trusted framework. The heart of this dissertation is the architecture and logic of that interaction. Additionally, to

show that the Open Verifier is a practical and usable tool, I describe by example the process of producing an extension for a realistic language, highlighting in particular the proof development strategies.

Contents

1 Introduction **1**

1.1 Overview of the Open Verifier 8

1.1.1 The Basic Idea 8

1.1.2 Structure of the Verification 9

1.1.2.1 Local Invariants 11

1.1.2.2 Parsing 12

1.1.2.3 Local Safety Conditions, Next States, and Initial State 13

1.1.2.4 Bringing It All Together 14

1.1.3 Building the Extension 15

2 The Logical Basis of the Open Verifier **17**

2.1 Logical Preliminaries 17

2.2 The Simplified Formal Development 18

2.3 Indexing 22

2.3.1 The Motivation for Indexing 22

2.3.2 Indexed Local Invariants 24

2.4 Augmented Decoder Input 26

2.5 Global Invariants 28

2.6 The Soundness Theorem 30

2.7 The Type Locinv 35

2.7.1 Existential Variables, Registers, and Assumptions 35

2.7.2 Progress Continuations 36

2.7.3 Locinvs, Again 37

2.7.3.1 A Final Restriction 40

2.7.4 The Decoder 41

2.7.5 Coverage Proof Rules 42

2.7.6 First-Order Logic 51
2.7.6.1 Types 53

3 The Implementation of the Open Verifier 54
3.1 A Simple Machine and Safety Policy 54
3.1.1 The Machine State 55
3.1.2 Machine Transitions 55
3.1.3 Memory Safety 58
3.1.4 Execution Parameters 58
3.1.5 Execution Parameters and Extension Lemmas 60
3.1.6 Labels 61
3.1.7 Verification on Actual Machines 62
3.2 Safety Policies 64
3.2.1 More about the SAL Safety Policy 64
3.2.2 Generalizing the Safety Policy 65
3.3 The Decoder 66
3.4 The Algorithm 70
3.4.1 Trusted Components 70
3.4.2 The Director 71
3.4.2.1 The Open Verifier Algorithm 72
3.4.3 The Proof Checker 74
3.4.4 The Initializer 75
3.5 Further Implementation Issues 77
3.5.1 Termination 77
3.5.2 Memory Safety of the Extension 78
3.5.3 Annotations 78
3.6 What Do We Trust? 79

4 Extensions 81
4.1 Cool 82
4.1.1 Programs in Cool 82
4.1.2 Compiling Cool 83
4.1.3 Verifying Cool 85
4.2 The Cool Extension 89
4.2.1 Local Invariants for Cool 90
4.2.1.1 Handling Recursive Types 91
4.2.1.2 The Cool Typing Predicate 92

4.2.1.3 Invariants 93
4.2.1.4 The Form of Cool's Local Invariants 94
4.2.2 Proofs for Cool 95
4.2.2.1 Memory Read 96
4.2.2.2 Dynamic Dispatch 98
4.2.3 Completing the Cool Extension 102
4.2.3.1 The Initial Coverage 102
4.2.3.2 Cool's Run-time Functions 103
4.3 Stack Handling 104
4.3.1 Memory Regions 105
4.3.2 Stack Frames and Stack Preservation 106
4.3.3 Stack Slots and Stack Pointers 107
4.3.4 Stack Overflow 108
4.3.4.1 The Stack on a 1MB Page 108
4.3.4.2 Guard Pages 109
4.4 Function Calls 112
4.4.1 Using Progress Continuations 112
4.4.2 Another Approach to Returns 115

5 Conclusions 117
5.1 Evaluating the Open Verifier 117
5.1.1 Trustworthiness 118
5.1.2 Flexibility 120
5.1.3 Scalability and Usability 120
5.2 Related Work 121
5.2.1 Virtual Machines 121
5.2.2 Typed Assembly Language (TAL) 122
5.2.3 Foundational Proof-Carrying Code (FPCC) 122
5.3 Future Research 125

Bibliography 127

Acknowledgements

It is a pleasure to thank my advisor, George Necula. His encouragement and guidance made this work possible, and indeed it reflects our long-time joint effort. I am especially grateful to George for his willingness to take on a student with an unusual background, unusual circumstances, and unusual prospects.

Bor-Yuh Evan Chang joined our project and provided many valuable insights in our conversations. Evan has supervised the construction of the Cool extension. Several sections of this dissertation were developed from joint work with George and Evan, notably parts of the Introduction, Section 3.4.2, Section 4.1.3, and Section 4.2. Evan in particular designed Figure 3.4, and initially worked out the examples of Section 4.2.

Kun Gao provided a great service in his work on the GUI used for the Open Verifier. It has proven an invaluable development tool. Kun has also worked on the Cool extension.

Adam Chlipala worked on the TAL extension, and our conversations have been very helpful. Adam provided many useful corrections to an early draft of this dissertation.

When I use "we" in this dissertation, I intend to indicate my view of the consensus opinion among those working on the implementation of the Open Verifier: George, Evan, Kun, Adam, and me.

At Berkeley I was very lucky to have Jeremy Bem and Russell O'Connor as fellow students in the Logic Group, who also think that computation needs to be taken seriously to understand logic. They were both very influential in my understanding of and education in logic.

I would like to thank Professor Ras Bodik for extensive comments on a draft of this dissertation.

Finally, I must thank my wife, Claire McConnell, for everything.

Chapter 1

Introduction

Various mechanisms exist for enforcing that untrusted code satisfies basic but essential security properties, such as memory safety. Without memory safety, untrusted code can interfere with the enforcement of higher-level security properties. In comparing particular enforcement mechanisms, the important metrics are *trustworthiness* and *flexibility* on the one hand, and what I will call *scalability* on the other. *Trustworthiness* means that we would like to believe that the enforcement mechanism actually works; that it is unlikely to have (potentially exploitable) bugs, etc. This can generally be measured in a very simple way by considering the amount of trusted code required to use the enforcement mechanism, its *trusted code base* (TCB)—the more lines of code required, the more likely it is to have bugs. *Flexibility* means that we do not want to restrict the structure of the untrusted code, or require that it come from one source language or compilation strategy. This reflects the fact that software systems tend to have components written in more than one language, particularly their runtime support routines. There is also value in flexibility with respect to the safety policy being enforced, so that the same mechanism can be adapted to handle more complicated issues such as resource usage.

By *scalability* I mean that we want a security mechanism which is practical; it needs to be useful for realistic (and in particular large) programs. This encompasses a number of factors, notably efficiency of *execution* (for mechanisms using dynamic checks on the program) and efficiency of *verification* (for mechanisms using static checks). For mechanisms which require the code producer to provide extra information along with the untrusted code—such as the proofs of proof-carrying code—we also must consider efficiency of *transmission*, and

usability or efficiency of *development*: how much of a burden is placed on the code producer to develop the extra information needed to pass the security mechanism.

Virtual machines. Consider, for example, enforcement strategies which require the whole untrusted program to be expressed in a single "trusted" typed intermediate language, such as the Java Virtual Machine Language (JVML) [24] or the Microsoft Intermediate Language (MSIL) [16, 17]. These virtual machines illustrate a kind of trade-off between trustworthiness and efficiency. The semantics of the intermediate language will typically require some dynamic checking such as array-bounds checks. It is easiest to implement the dynamic checking by running the code with a trusted interpreter. The significantly reduced execution speed under interpretation has led to the use of just-in-time compilers (JITs). However, a compiler is significantly more complex (and thus more difficult to trust) than an interpreter, particularly as more and more optimizations are used to improve the execution speed.

Where these approaches are really found lacking is in flexibility. Each of these intermediate languages is a good target for one or more corresponding source-level languages. Programs written in other source languages can be compiled into the trusted intermediate language but often in unnatural ways with a loss of expressiveness and performance [7, 18, 9]. Most importantly, these languages enforce security by means of a fixed type system and so can only apply to that part of the untrusted code which conforms to the given type system; this will often not be the case for low-level or high-performance runtime support routines.

The design of MSIL is intended to allow it to better handle multi-lingual software. MSIL contains support for multiple languages and also permits the straightforward compilation of low-level languages, such as C and C++, because it incorporates a low-level sublanguage. The results of such compilations are, however, not directly verifiable and thus less privileged and more limited in how they can interact with verified code. Moreover, this flexibility increases the complexity of the intermediate language. For example, MSIL includes eight distinct forms of function calls, including direct, virtual, interface virtual, and indirect calls, along with tail versions of these. Finally, not surprisingly, MSIL is still not perfect for every imaginable source language. The ILX project [40] suggests that MSIL could be extended with, among other features, two addi-

tional forms of function call to better support the compilation of functional higher-order languages. Similar extensions have been proposed for supporting parametric polymorphism [10, 20]. There is a strong temptation to create an intermediate language type-system that is as expressive as possible because it is hard to change the type system after many copies of the virtual machine are deployed.

Proof-carrying code. Proof-carrying code (PCC) [32, 29, 30] allows a code producer to associate to a program a machine-checkable proof of its safety. As a security enforcement mechanism, PCC is typically envisioned as allowing the code producer to send executable code, along with an encoding of a proof, to the code consumer. The code consumer then uses some trusted software to analyze the untrusted code and produce a safety theorem, a proof of which witnesses the safety of the code. Finally the code consumer uses a trusted proof checker to check that the accompanying proof is a valid proof of the correct safety theorem.

Virtual machines can be seen as specific instances of PCC. The bytecode sent to a virtual machine encodes both the executable code (via the trusted interpreter or JIT) *and* its proof of safety (to be checked by the trusted bytecode verifier). Of course in general usage "proof-carrying code" tends to imply systems in which the proofs are more directly representations of proofs in formal logic. The key benefit of proof-carrying code, especially in its more general forms, is that more of the burden of certifying safety is shifted from the consumer to the producer, allowing the consumer to perform complicated verifications using a much smaller trusted code base.

The first implementation of PCC was the Touchstone system [11], which verifies that optimized native-machine code produced by a special Java compiler is memory safe. The trusted side uses a *verification-condition generator* (VCGen) to examine the program and produce its safety theorem. On the untrusted side, the certifying compiler automates the generation of the safety proof. The Touchstone system is able to verify quickly even programs of up to one million instructions. This level of scalability has been achieved through careful engineering of the data structures used in the implementation of the verifier along with a number of novel algorithmic "tricks". This does mean that the verifier code, while still much smaller than an alternative trusted compiler, is far from being easy to understand and trust. In fact, there were errors [22] in

that code that escaped the thorough testing of the infrastructure. Moreover, the Touchstone verifier is specifically engineered to the Touchstone certifying compiler for Java, dramatically limiting any ability to extend the system.

PCC was taken in a different direction with *foundational* proof-carrying code (FPCC) [5, 25, 6, 4, 1], which aims to maximize the trustworthiness and flexibility of the system. In Touchstone (and with bytecode verifiers for that matter) there is still a significant amount of analysis performed by the trusted component (the VCGen in Touchstone) which produces the safety theorem about the untrusted code. FPCC on the other hand reduces the trusted code base to a minimum. The semantics of the target machine and the safety policy are directly encoded as definitions in formal logic; then producing the safety theorem is a trivial matter of saying that the machine integers, which constitute the untrusted code, are in fact safe according to those definitions. No program analysis at all is performed by the trusted infrastructure; the trusted code base comprises only the aforementioned logical definitions, and the proof checker.

The cost of the enhanced trustworthiness and flexibility of FPCC is the difficulty of developing the necessary proofs of safety, which must completely describe in the foundational logic whatever mechanisms (for instance, type systems) are used to establish safety. Rather fancy mathematical machinery has been invoked to handle e.g. mutable recursive types [6, 1]. Partly in reaction to the difficulty in producing FPCC proofs as originally envisioned in [5], another group has developed *syntactic* FPCC [19, 39, 43]. Syntactic FPCC does not offer a new architecture but a new and hopefully easier approach to organizing and developing the foundational proofs, which illustrates neatly that purely logical questions may have engineering impact in this field.

Both the original (semantic) and syntactic approaches to FPCC are subjects of ongoing research, and it is clear that the development of FPCC is significantly more labor-intensive than the development of Touchstone. The extra engineering effort required to achieve that level of scalability may be even more costly.

The Open Verifier. My thesis work has been motivated by two main ideas:

1. that it should be possible to work towards the trustworthiness and flexibility promised by FPCC, while still taking more-or-less direct advantage of the substantial engineering which enabled Touchstone to scale to large programs; and

2. that ideas from PCC could provide a better approach to a truly generic virtual machine.

The result is the *Open Verifier* system which I describe in this dissertation. I draw particular attention to its *architecture*, its *logic*, and the *proof technique* we have used with it.

First, the Open Verifier provides a particular *architecture* for an extensible security enforcement mechanism. It slightly generalizes the PCC model by requiring, not that the code producer provide some representation of a formal proof of safety, but in fact that the code producer provide a *verifier*—as executable code. Then the trusted components of the architecture work together with the untrusted verifier (called the *extension*) in such a way that the final verification can be trusted.

Compare this with typical virtual machines, which specify not only the low-level safety policy of interest (i.e., memory safety), but also fix one particular *mechanism* that is sufficient for enforcing it (i.e., a particular strong type system). The Open Verifier allows the producer of the untrusted code to choose the mechanism by which the low-level safety-policy is enforced, with complete control over the type system used or even whether a type system is used at all. After all, who is in a better position to tell what mechanism works best for one particular program? Simply selecting among several built-in mechanisms may not be sufficient or even desirable. By using a simple untyped intermediate language—possibly the machine language for the target machine—fewer constraints are placed on the code producer. This also allows the code producer to perform optimizations that must traditionally be done by JITs.

Sending a verifier rather than a proof offers engineering advantages by reducing concerns about efficiency of the transmission of potentially large proofs. In particular one avoids the complications necessary for proof compression and decompression [33, 35]. Instead, the primary scalability concern, besides the cost of developing the untrusted extensions, is in the efficiency of the verification process; but as execution time is less expensive than transmission time, this is the right trade-off to make. Moreover, none of the generality of PCC is lost; consider the degenerate extension which carries only the explicit particular proofs for the program in question.

The challenge is to ensure the soundness of a verification process which uses untrusted components, while trying to control the additional burden placed on the writer of the extension—ideally, we would like writing the untrusted

verifier to be not many levels more difficult than writing a bytecode verifier for a typical virtual machine. One obvious solution is to verify the correctness of the verifier itself, but this would be impractical for realistic verifiers. At the same time, this approach would indirectly impose restrictions on the design and implementation of the extension. We would like to use an untrusted verifier as a black box, ignoring whatever algorithms, data structures, or heuristics it uses internally, or even the language in which it is written. The technique of runtime result checking [42] makes this possible and has already been used successfully for the problem of verifying the correctness of compilations [37, 31]. Instead of verifying the verifier itself, we structure the verification as a series of queries that the extension has to answer. A small trusted module poses the queries in such a way that it is able to check the correctness of the answers with respect to the low-level safety policy, typically requiring that each answer come with a proof of its correctness written in a suitable logic. For instance, when enforcing a type system, the queries reduce to typing judgments for various subexpressions and their proofs are typing derivations.

Second, the Open Verifier also uses a particular *logical mechanism* for communication between the trusted and untrusted components. The goal has been to structure the logical development to make feasible the development of the untrusted verifiers. The key development here is the Open Verifier's notion of *local invariant*, a logical description of a particular state of execution, which carries also some partial-correctness assertions about possible future states which can be considered safe.

For instance, Touchstone relies exclusively on Horn logic whereas FPCC has used rather sophisticated higher-order logic; our work with the Open Verifier has shown that the logical strength submits to a finer analysis, such that most per-program proofs occur in Horn logic, per-compiler proofs in first-order logic, and most of the use of higher-order features is restricted to the once-only soundness proof of the Open Verifier framework. This is true even for programs that manipulate function pointers directly.

Finally, we have used a particular *proof technique* for structuring the proofs to be given by extensions for the Open Verifier. By analogy with software engineering these might be called proof engineering considerations. The approach we have used are inspired by the structure of Touchstone, and are distinct from both semantic and syntactic FPCC. Like semantic FPCC, we use semantic definitions of the typing predicate; however, circularities are broken by an

intensional approach to recursive types, which eliminates many needless complications. Moreover, in contrast with FPCC, and also with approaches to safety based on TAL [28, 27, 26, 15], our proofs tend to stay more at the level of predicate logic, descending into the type system only locally. This has some apparently superficial ramifications—such as the fact that we prefer direct use of equality to the use of singleton types—but also points to a distinct approach to using type safety to enforce memory safety: Instead of proving a global type-soundness theorem and establishing that the entire program is well-typed, we use types locally to establish local safety conditions, using definitions and lemmas which relate particular local assertions about types to the state of the memory. Even absent the Open Verifier architecture, the proof-development approach it has inspired could lead to a third way in FPCC.

How to read this dissertation. The next three chapters provide the technical developments of my thesis. Chapter 2 presents the logical developments needed to support the architecture, most importantly the soundness theorem and the notion of local invariant. Chapter 3 gives particular instantiations for the necessary logical notions, introduces the algorithm used to implement the trusted components of the Open Verifier architecture, and discusses other implementation-related issues. Chapter 4 concerns itself with the untrusted extensions, and is primarily an effort to demonstrate by example that extensions can be feasibly written which handle interesting programs.

In Chapter 5 I conclude with some preliminary results from our prototype implementation of the Open Verifier, a further discussion of related work, and some suggestions for future progress.

Please note that the body of the dissertation is organized according to logical dependence rather than motivation. Reading it straight through would be appropriate for someone who is already familiar with the basic ideas and wants to solidify an understanding of the details. For the more typical reader, who wants to develop an appreciation of the basic ideas, I suggest the following approach:

1. Read the remainder of this Introduction, a high-level overview of the Open Verifier.

2. Read Section 2.2, which describes a simplified version of the logical structure of the safety verifications produced by the Open Verifier.

3. Read Section 3.4 which describes the algorithm implemented by the trusted code of the Open Verifier. (This will require a willingness to ignore references to the more complicated logical structure introduced later in Chapter 2.)

4. Read Section 4.1 and Section 4.2 which describe a typical implementation of an untrusted extension for the Open Verifier.

Those sections should suffice to provide a good feel for the basics of the system; other sections can then be read as needed to fill in desired details.

1.1 Overview of the Open Verifier

The goal of the Open Verifier is to create a flexible security enforcement mechanism, based on the idea that clients should be able to customize the code verification process to the particular source language and compilation strategy used to produce the untrusted code.

At the same time, the Open Verifier is intended to be a practical tool; clients need to be able to provide with reasonable effort the customization information for realistic languages and compilers. This rules out theoretically attractive possibilities, such as requiring the client to send a code verifier along with a proof of its correctness with respect to the safety policy. In order to increase the usability of the tool, we are willing to incorporate into the trusted infrastructure elements which are likely to be common to many possible customizations. This, however, is balanced by the criterion that the Open Verifier should have a reasonably small trusted code base.

1.1.1 The Basic Idea

The Open Verifier requires code producers to send a code verifier for their untrusted code, to ensure that the code satisfies the safety policy. The code verifier is sent as executable code.

How is it possible to use an untrusted verifier to produce a trustworthy verification?

As mentioned above, proving the correctness of the verifier seems too difficult. Instead we require that the untrusted verifier produce intermediate information which can be used by the Open Verifier to check the validity of

each *individual* verification. In essence, the verifier must be a proof-generating verifier; it must emit a proof, to be checked by the trusted components of the Open Verifier, that the particular code in question is safe. For increased usability, certain basic common tasks (such as parsing the code) are performed by the trusted infrastructure itself, requiring no proof.

In practice, rather than creating a single monolithic proof to be checked by the trusted proof checker, we have found it more convenient to have the trusted framework send particular requests (such as proof obligations) to the untrusted verifier, which responds with the particular proofs and information necessary. We call the untrusted verifier an *extension* to the Open Verifier, and consider that the extension and the trusted components work together to produce the final verification.

In principle an extension could be created for every new piece of untrusted code to be verified. In practice we expect that an extension will be created for a specific source language and compilation strategy, and then will be used to verify all code produced by a given compiler. Note also that we expect to be able to write extensions to verify the output of existing compilers, rather than having to write a new certifying compiler from scratch.

1.1.2 Structure of the Verification

The requests posed to the extension essentially mandate a form of verification based on abstract interpretation or symbolic evaluation. The verifier must maintain an abstract state representing the state of the program at a certain point in execution. Then it simulates the effect of each execution step on the abstract state. To ensure that the process is finite, the abstract state at join points (such as the start of a loop) must be made sufficiently general that a single evaluation of the loop corresponds to all possible evaluations. (In practice, a loop may be evaluated several times, each time weakening the abstract state at the join point until it is sufficiently general.) Finally the verifier must check that for potentially unsafe execution steps (such as memory accesses under a memory-safety policy), the abstract state before the execution step guarantees that taking the step is always safe. This structure is commonly used, notably by bytecode verifiers for virtual machines.

A verifier using this strategy could be broken down into the following parts. For each abstract state to be considered, the verifier must

1. parse the instruction to be executed—this assumes that a given abstract state uniquely specifies the instruction to be executed;

2. check that, in any concrete state represented by the current abstract state, the instruction is safe to execute; and

3. determine the abstract states corresponding to the new state (or states, at branch points) which could result from the execution of the instruction. At join points, this may mean re-using abstract states already considered, to ensure that the verification is finite.

I call this process, of handling a given abstract state, *scanning* the state. Finally the verifier must

4. produce an initial abstract state which corresponds to any possible entry state of the program; and

5. ensure that the iterative process, of scanning new abstract states which result from scanning, is complete. This ensures that all reachable code in the program is considered.

An example of using this verification technique is given in Section 4.1.3.

In order to use an *untrusted* extension, for each of these five requirements we must either

1. meet the requirement with trusted code; or

2. require the extension to give enough information that the way it meets the requirement can be checked. This will be a proof of one form or another.

In the Open Verifier, the first task (parsing) and the last task (iterative completion of the verification) are left entirely to trusted code. The other three tasks (ensuring local safety, determining next states, and producing an initial state) are handled by requiring a proof that *in each particular case* they have been done correctly. Note that this is, of course, very different from requiring a proof that the extension code correctly implements an algorithm which *always* does it correctly; in particular, we don't mind using an unsound extension, as long as it can prove that it is sound for each particular verification step of the program being considered.

In the following sections I discuss the various verification tasks, and the manner in which they are performed by the Open Verifier. First it is necessary to describe the notion of abstract state used.

1.1.2.1 Local Invariants

To facilitate the communication between the untrusted extension and the trusted components of the Open Verifier, we must specify an appropriate notion of the abstract state describing each step of verification. Essentially, we use logical predicates of machine states; the abstract state can be understood as specifying the collection of concrete machine states which satisfies the predicate.

The term that I use for the abstract states is *local invariants*, often abbreviated *locinvs*. Each locinv specifies certain facts which hold of certain machine states at specific points during the execution of the program. In practice we use locinvs which describe some set of machine states at a particular local point in the code, a particular value of the program counter. In principle however, there may not be anything particularly "local" about a locinv. For instance,

$$\lambda\rho.\texttt{True}$$

holds of all states ρ anywhere during the execution of any program, safe or otherwise—though again, the locinvs useful in practice are local in that they specify certain facts about states at a particular point in the program. Meanwhile, a locinv like

$$\lambda\rho.(\texttt{pc}\,\rho) = 5 \wedge (P\ \rho)$$

(where $\texttt{pc}\,\rho$ is the program counter of state ρ) shouldn't be understood as necessarily claiming that all states ρ that reach line 5 will satisfy P. It is a merely a description of certain states, namely those which are at line 5 and satisfy P. One way to understand the job of the extension, is to see it as building a list of locinvs which it then does claim are complete, in that every state during program execution will satisfy one or more of the locinvs; but this list may contain more than one locinv corresponding to a given location in the program.

Other words for locinv might be "abstract state", or "verification state". In this dissertation I will sometimes use *continuation*; particularly when one locinv C claims the safety of another locinv D, D can be seen as an abstract description of one way in which execution might proceed.

As a matter of fact, locinvs need to be somewhat more complicated than merely predicates of machine states. We wish to handle such assertions as the claim that whenever execution reaches the start of a function, the return address register holds a code address which is safe to execute. Since the entire verification process is about establishing the safety of executing the code, some care needs to be taken to avoid circularity in such assertions. These issues are discussed in Section 2.3.

The type of locinvs is described in full detail in Section 2.7.

1.1.2.2 Parsing

Given an locinv which specifies a certain location in the untrusted code, the most basic task is to determine what instruction is going to be executed. In general, the untrusted code could be in its final compiled form—as a sequence of machine integers representing the assembled instructions—or it could be sent in some sort of intermediate language which is then compiled, by trusted components of the framework, into machine code. The second possibility is closer to the behavior of virtual machines, where the intermediate language of bytecodes is designed so that the very form of the bytecodes prevents certain behaviors. In our prototype implementation of the Open Verifier, we use the assembly language of the target machine, thus assuming a trusted assembler. It would not be substantially harder—and would not require substantially more trusted code—to verify machine code instead.

However, we do not verify the target instructions directly. That would require writing both the trusted components and the extensions differently for each target architecture. Instead we translate the assembly instructions of the untrusted code into a generic language called SAL (the Simple Assembly Language). Both the trusted and untrusted verifier components verify SAL programs. Thus, in order to trust the Open Verifier, one must trust the translator to produce a SAL program, the safety of which guarantees the safety of the original program in the language of the target machine. Whereas when working with bytecodes, each bytecode may correspond to a sequence of machine instructions, when working with SAL each machine instruction decomposes into one or more SAL instructions; for example, a stack pop instruction would decompose into separate SAL instructions for reading a word from memory and changing the stack pointer. Since SAL is so simple, and is finer-grained than machine instructions, it is not difficult to trust that the translator is correct.

SAL is described further in Section 3.1.

1.1.2.3 Local Safety Conditions, Next States, and Initial State

Most of the verification effort lies in the iterative exploration of the code. Given a particular locinv, once the parser determines the instruction to be executed, the verifier must establish that it is safe to execute the instruction, and must produce a new locinv (or locinvs) corresponding to the state after the instruction is executed. I call this *scanning* the locinv. In the Open Verifier, a trusted component called the *decoder* determines what is to be proved; the extension then must meet those proof obligations.

The decoder's first job is to provide local safety conditions. For example, suppose we are scanning a locinv C which asserts that $\mathbf{r}_y = 10$, and the current instruction is $\texttt{write}\ \mathbf{r}_x\ \mathbf{r}_y$, that is, to write the value in register $\mathbf{r}_x$ into the contents of memory at address $\mathbf{r}_y$. The decoder will emit the proof obligation $\texttt{addr}\, 10$, where `addr` is a suitably-defined predicate which holds only of accessible memory addresses. The extension must then produce a proof to meet this obligation.

The decoder also establishes the semantics of the instruction being executed, by producing a description—in fact, a locinv or locinvs—of the state after executing the instruction. In this respect the decoder acts as a strongest-postcondition generator. For the memory-write example above, the decoder's output continuation will be

$$\exists M_{\texttt{old}}.\big(M = (\texttt{upd}\, M_{\texttt{old}}\ \mathbf{r}_y\ \mathbf{r}_x)\big) \wedge C[M \mapsto M_{\texttt{old}}].$$

That is, in the new state, we know that the memory is the result of updating the old memory with value $\mathbf{r}_x$ in address $\mathbf{r}_y$; and we also know that all the facts asserted by C still hold as long as all references to the memory are considered to be about the old memory. This is the standard strongest postcondition for an assignment. See Section 3.3 for more examples concerning the decoder.

The extension will also produce continuations describing the next state. The Open Verifier trusts the continuations produced by the decoder, but will actually use the continuations produced by the extension in order to continue the iterative scanning process. The extension must establish that its continuations "cover" the decoder's; that they include more possible concrete states, and so are more general, in fact logically weaker than the decoder's next states.

It may be counterintuitive that the trusted component will produce stronger results than the untrusted component. The decoder performs a very simplistic symbolic evaluation, finding the strongest postcondition of executing the instruction on the input state. The extension on the other hand is willing to forget information; for instance it will not necessarily record the contents of a given address in memory, even directly following a memory write, but instead might only record the fact that the memory is well-typed in the appropriate sense. Finding the appropriate weakening is necessary for verifying loop structures using only a small finite number of loop traversals. Furthermore, to handle indirect jumps the decoder produces an output locinv which doesn't specify the next instruction to be executed, so can't be scanned in the way here described. The extension will have to prove that the address jumped to is safe; for instance, it might be able to prove that it is in fact one of a short list of specific addresses, such as when the indirect jump implements a switch statement. For examples of how the extension might produce locinvs for the next states, and prove that they cover the decoder's locinvs, see Section 4.2.

The initial state of the verifier is handled similarly. A trusted component called the *initializer* produces a locinv which gives a detailed description of the initial state of execution; the extension produces its own locinvs describing the initial state, which are the ones which will be iteratively scanned. The extension must prove that its initial locinvs cover the one provided by the trusted framework.

1.1.2.4 Bringing It All Together

The Open Verifier must ensure that all the individual verifications of single execution steps result in a complete verification of the program. This is handled by a trusted component called the *director*. The director accumulates the extension's locinvs and iteratively scans them until no new locinvs are produced. It checks the proof of each local safety condition, and checks each coverage proof to ensure that the extension's locinvs are sufficient to describe all states possibly resulting after one step of execution. The algorithm for the director is discussed in Section 3.4.

The soundness of the Open Verifier is proved, at a purely logical level, in Section 2.2 (where certain complications are omitted) and Section 2.6 (in full detail).

1.1.3 Building the Extension

It's not enough that the architecture of the Open Verifier guarantees that any verification produced with the extension is trustworthy. It's also necessary that extensions can be produced, with reasonable effort, to verify realistic programs. In this dissertation I concentrate on the trusted framework, but I want to show enough about building extensions to show that this is plausible.

Although in principle an extension could be tailored to handle a single program, I propose that each extension will correspond to a particular source language and compilation strategy—in effect, to a particular compiler. Proof-carrying code is closely linked to the idea of the *certifying compiler*, a compiler which produces not only executable code but also a proof of its safety. With the Open Verifier, ready-made compilers can be used almost unchanged, with a separate extension written to produce the verifications. (Some changes to the compiler will be necessary to provide information—such as the typing declarations of functions—which would otherwise disappear during the compilation process.)

The effort of writing an extension can be divided roughly into four steps.

1. Write a "conventional verifier" along the lines described above in Section 1.1.2. The step should be of comparable difficulty to writing e.g. a Java bytecode verifier.

2. Translate the abstract state of the conventional verifier into the framework of locinvs. This requires designing logical predicates which describe the abstract state, usually including typing predicates. The most difficult part of this step is making explicit certain invariants which may be implicit in the conventional verifier. For instance, if the conventional verifier guarantees the well-typedness of memory, each locinv must contain an explicit assertion that the memory is well typed.

3. Write a logic program which can be used to automatically establish the proofs required by the Open Verifier. We use an untrusted logic interpreter called Kettle which can be used as an automated theorem prover given a set of rules to treat as lemmas. These rules will contain a transcription of the typing rules of the type system being used, as well as rules necessary to prove that explicit invariants (such as the well-typedness of memory) are maintained.

4. Write the definitions of the predicates introduced in step 2, and prove the lemmas introduced in step 3. These definitions and proofs are produced by hand; currently we use the interactive proof assistant Coq.

Observe in particular that the proof effort is divided into per-program proofs, which are produced automatically, and per-compiler proofs, which have to be produced by hand. See Section 4.2 for examples from converting a particular conventional verifier into an extension.

Chapter 2

The Logical Basis of the Open Verifier

In this chapter I will develop the formalism to be implemented by the Open Verifier. I will introduce the appropriate abstract notions corresponding to the safety of a program, and prove Theorem 2.6.11 stating a set of conditions which guarantee safety. The role of the untrusted extension is to provide proofs of these conditions; the role of the trusted infrastructure is, most importantly, to check the given proofs, but also (as I will discuss in Section 3.4) to provide some of the iterative behavior that will be common to all verifications.

I begin with a simplified version of the development; then in later sections I discuss refinements, specializations, and generalizations needed to obtain a logical basis for actual verification work, culminating in Theorem 2.6.11.

2.1 Logical Preliminaries

In general I work within the Calculus of Inductive Constructions [13, 14], which is the logic implemented by the Coq proof assistant [12]. This is a higher-order logic including sorts `Set` and `Prop`, which are used for sets and propositions, respectively. The sorts `Set` and `Prop` themselves belong to the sort `Type`, but the word "type" will be used for values of any of the sorts. The logic allows types and predicates to be defined by induction, as the least type or predicate closed under a given list of constructors.

Though I use this strong logic in discussing the Open Verifier framework,

for implementation reasons it may be worthwhile to restrict extensions to using a small fragment of the logic in presenting proofs to the trusted framework; see Section 2.7.6.

2.2 The Simplified Formal Development

Let us assume the existence of a type

$$\texttt{state} : \texttt{Set}$$

of machine states, as well as a relation

$$\leadsto : \texttt{state} \to \texttt{state} \to \texttt{Prop}$$

describing *safe* machine transitions. These parameters are intended to encode the machine semantics as well as the safety policy; in particular, $\leadsto$ is that subset of all machine transitions which are to be considered safe. See Section 3.2 for a discussion of what kinds of safety policies can be encoded in this way.

Definition 2.2.1. A state ρ is capable of i steps of safe progress, written $\texttt{prog}_i\,\rho$, when any chain of $\leadsto$-transitions starting at ρ can be extended to at least i steps long. That is,

$$\begin{aligned}&\texttt{prog}_0\,\rho \text{ always holds;}\\ &\texttt{prog}_{i+1}\,\rho \iff (\exists \rho'.\,\rho \leadsto \rho') \wedge (\forall \rho'.\,\rho \leadsto \rho' \implies \texttt{prog}_i\,\rho').\end{aligned}$$

By $\texttt{prog}\,\rho$ is meant $\forall i.\,\texttt{prog}_i\,\rho$.

Note that $\leadsto$ only needs to model the safe transitions to be made under the control of the program to be verified. Presumably the program may return control to an operating system after its execution has finished; this would be considered safe, and so to use the above definition we might model the further execution in the OS as a simple infinite $\leadsto$-loop. The actual behavior of the machine at that point is no longer relevant to the safety of the program in question.

Safety of a program is proven by establishing that $\forall i.\,\texttt{prog}_i\,\rho_0$ for every possible initial state ρ_0. I shall not introduce any formal type corresponding to the notion of a "program" to be verified. Instead, I will assume that any

program determines some logical description of all possible initial states, some $C_0 : \texttt{state} \to \texttt{Prop}$ such that $(C_0\ \rho_0)$ holds of every possible initial state ρ_0. The program for which C_0 describes its initial states will be safe if

$$\forall \rho_0 . (C_0\ \rho_0) \implies \forall i .\, \texttt{prog}_i\, \rho_0 .$$

This is the key statement that must be proved to verify the safety of a program.

The extension will work with a type of verification states, corresponding typically to facts known to hold at some particular point during program execution. I call these *local invariants* and assume the existence of some type

$$\texttt{locinv} : \texttt{Type}.$$

I will not specify the type `locinv`, but assume that there is a satisfaction relation by which any `locinv` can be regarded as a state predicate:[1]

$$\models : \texttt{state} \to \texttt{locinv} \to \texttt{Prop}.$$

For a state ρ and a locinv C, I write $\rho \models C$ for the statement that this relation holds. It is only this satisfaction relation which matters about locinvs for the theorem I am establishing; and in examples I will typically write locinvs in the form of state predicates.

Definition 2.2.2. A locinv C is *safe for i steps*, written $\texttt{safe}_i\, C$, when

$$\forall \rho . (\rho \models C) \implies \texttt{prog}_i\, \rho .$$

By $\texttt{safe}\, C$ is meant $\forall i .\, \texttt{safe}_i\, C$.

With this definition, we can verify a program by proving $\forall i .\, \texttt{safe}_i\, C_0$, for an initial locinv C_0 such that all possible initial states satisfy C_0.

Definition 2.2.3. A list $\mathcal{E}$ of locinvs is said to *cover* a list $\mathcal{D}$ of locinvs, written $\mathcal{E}\, \texttt{covers}\, \mathcal{D}$, when

$$\forall i . \Big(\bigwedge_{E \in \mathcal{E}} \texttt{safe}_i\, E \Big) \implies \Big(\bigwedge_{D \in \mathcal{D}} \texttt{safe}_i\, D \Big).$$

[1] I do this rather than simply defining `locinv` as a state predicate, because I will eventually want to use a particular notion of `locinv` which is more restricted, and automatically enforces certain invariants. See Section 2.7.

The Open Verifier framework involves using a *decoder* to determine whether and which safe transitions are possible from the states satisfying a locinv. The decoder reflects the definition of the state transition relation $\leadsto$, in terms of locinvs; in the implementation it is used to *replace* reasoning about $\leadsto$, so that the extensions only need prove specific claims required by the decoder, rather than reasoning about $\leadsto$ directly.

Definition 2.2.4. A *decoder* is a function which takes a locinv and returns a pair, consisting of a state predicate (the local safety condition) and a list of locinvs (the possible next states); thus a decoder is of type

$$\texttt{locinv} \rightarrow \big((\texttt{state} \rightarrow \texttt{Prop}) \times \texttt{locinv list}\big)$$

I will sometimes use the term *continuation* as a synonym for locinv, particularly in the context of the decoder output, which indicates ways for the execution to continue.

Definition 2.2.5. A decoder `decode` satisfies the *decoder correctness property* iff, for any locinv C, where $(P, \mathcal{D}) = \texttt{decode}\, C$,

$$\forall \rho.(\rho \models C) \wedge (P\ \rho) \implies (\exists \rho'.\rho \leadsto \rho') \wedge \Big(\forall \rho'.\rho \leadsto \rho' \implies \bigvee_{D \in \mathcal{D}} \rho' \models D\Big).$$

That is, for any state ρ satisfying the input locinv C, if the local safety condition P holds, then progress is possible for at least one step, and the resulting state will satisfy some $D \in \mathcal{D}$.

I now introduce the key property to be established by the untrusted extension. The word "scanning" comes from the algorithm used in the implementation, where locinvs from a list are iteratively "scanned" to provide the necessary verification about each.

Definition 2.2.6. A set $\mathcal{E}$ of locinvs is *closed under scanning with respect to* `decode` if the following two conditions hold for each $E \in \mathcal{E}$. Let $(P, \mathcal{D}) = \texttt{decode}\, E$. Then

1. $\forall \rho.(\rho \models E) \implies (P\ \rho)$;
2. $\mathcal{E}$ `covers` $\mathcal{D}$.

Lemma 2.2.7. *If a set $\mathcal{E}$ of locinvs is closed under scanning with respect to* `decode`*, and* `decode` *satisfies the decoder correctness property, then for each $E \in \mathcal{E}$, we have that $\forall i.\, \mathtt{safe}_i\, E$.*

Proof. The proof proceeds by induction on the index i. The base case $i = 0$ holds trivially. So suppose as induction hypothesis that for each $E \in \mathcal{E}$, $\mathtt{safe}_i\, E$. We will establish that for each $E \in \mathcal{E}$, $\mathtt{safe}_{i+1}\, E$, which will complete the proof by induction.

Fix some $E \in \mathcal{E}$. To show $\mathtt{safe}_{i+1}\, E$, by definition we must show that for every ρ such that $\rho \models E$, $\mathtt{prog}_{i+1}\, \rho$. Fix some ρ such that $\rho \models E$.

Let $(P, \mathcal{D}) = \mathtt{decode}\, E$. By the decoder correctness property, instantiated at our chosen ρ,

$$(\rho \models E) \wedge (P\ \rho) \implies (\exists \rho'.\rho \rightsquigarrow \rho') \wedge \big(\forall \rho'.\rho \rightsquigarrow \rho' \implies \bigvee_{D \in \mathcal{D}} \rho' \models D\big).$$

By condition (1) of closure under scanning, we have that $(P\ \rho)$. Therefore it is possible to make one step of progress from ρ; and any resulting state ρ' will satisfy some $D \in \mathcal{D}$.

We now use condition (2) of closure under scanning, which has that $\mathcal{E}\,\mathtt{covers}\,\mathcal{D}$. By the induction hypothesis $\mathtt{safe}_i\, E$ for each $E \in \mathcal{E}$. Thus by the definition of `covers`, $\mathtt{safe}_i\, D$ for each $D \in \mathcal{D}$.

By the definition of `safe` we can conclude that $\mathtt{prog}_i\, \rho'$ for any ρ' satisfying some $D \in \mathcal{D}$ at index i. But we have already established that this is the case for each ρ' resulting from our chosen ρ after one step of execution. Thus we have established $\mathtt{prog}_{i+1}\, \rho$, completing the proof. □

Theorem 2.2.8 (Soundness of the Open Verifier). *Suppose a set $\mathcal{E}$ of locinvs is closed under scanning with respect to* `decode`*, and* `decode` *satisfies the decoder correctness property. Suppose also that $\mathcal{E}\,\mathtt{covers}\{C_0\}$, where C_0 is an initial locinv for a program in the sense that for any possible initial state ρ_0, $\rho_0 \models C_0$. Then the program is safe in the sense that each such ρ_0 can make indefinite safe progress.*

Proof. By the theorem $\forall i.\, \mathtt{safe}_i\, E$ for each $E \in \mathcal{E}$. Thus by the definition of `covers`, $\forall i.\, \mathtt{safe}_i\, C_0$. The rest follows by the definitions of safety and progress. □

Later in Section 2.7 I will instatiate the type `locinv` and its satisfaction relation $\models$. Once that is done, the role of the untrusted extension can be understood as producing a list of locinvs, which covers an initial locinv for the the program to be verified; and proving that it is closed under scanning with respect to a correct decoder.

In the next few sections, however, I introduce certain complications to the simple logical formalism presented above. Then in Section 2.6 I will re-prove the soundness theorem for the actual formalism.

2.3 Indexing

2.3.1 The Motivation for Indexing

Commonly needed in verifications is some means of expressing the notion that it is safe to continue execution from a certain point. For instance, when the execution is at the beginning of a function F, it can be assumed that it is safe to return to the address stored in the return-address register. (Typically of course it is only safe if some function post-condition also holds, but I will omit this consideration for now.) This can be expressed as a locinv:

$$\lambda\rho.\ (\texttt{pc}\,\rho) = F \wedge \texttt{safe}(\lambda\rho'.(\texttt{pc}\,\rho') = (\texttt{ra}\,\rho)),$$

where `ra` is the return-address register.

Later during the execution of the function, the original return address might end up stored on the stack in order that the return-address register can be used for another function call. Then a locinv might hold such as:

$$\lambda\rho.\ \exists\texttt{ra}_{\text{orig}}.\ (\texttt{sel}(\mathbf{r}_M\ \rho)\ n) = \texttt{ra}_{\text{orig}} \wedge \texttt{safe}(\lambda\rho'.(\texttt{pc}\,\rho') = \texttt{ra}_{\text{orig}}).$$

Here $(\texttt{sel}(\mathbf{r}_M\ \rho)\ n)$ means the contents of memory $(\mathbf{r}_M\ \rho)$ at address n (wherever the address was where the return address was stored).

Consider now how this might be put to use at function call and return points. The return works nicely; the decoder will emit a continuation such as

$$\lambda\rho.\ \exists\texttt{ra}_{\text{orig}}.\ (\texttt{pc}\,\rho) = \texttt{ra}_{\text{orig}} \wedge \texttt{safe}(\lambda\rho'.(\texttt{pc}\,\rho') = \texttt{ra}_{\text{orig}}).$$

Call this locinv C. The extension will now face a coverage proof obligation, to prove that C is itself `safe`. In fact,

Lemma 2.3.1. $\texttt{safe}\, C$.

Proof. Pick some ρ such that $\rho \models C$. We must show that $\texttt{prog}\, \rho$.

Since $\rho \models C$, then in particular

$$\exists \texttt{ra}_{\text{orig}}.\ (\texttt{pc}\, \rho) = \texttt{ra}_{\text{orig}} \wedge \texttt{safe}(\lambda \rho'.(\texttt{pc}\, \rho') = \texttt{ra}_{\text{orig}}).$$

Pick some $\texttt{ra}_{\text{orig}}$ witnessing this existential. Since $(\texttt{pc}\, \rho) = \texttt{ra}_{\text{orig}}$ then $\rho \models (\lambda \rho'.(\texttt{pc}\, \rho') = \texttt{ra}_{\text{orig}})$. But we have that $\texttt{safe}(\lambda \rho'.(\texttt{pc}\, \rho') = \texttt{ra}_{\text{orig}})$, and so it follows that $\texttt{prog}\, \rho$. This completes the proof. □

Corollary 2.3.2. $\{\}\, \texttt{covers} \{C\}$.

Essentially, C claims its own safety.

When we consider the function call itself, we begin to encounter problems. Suppose the machine is about to execute a jump instruction to F, which we want to interpret as a function call, where the return will go to line n. Let

$$\begin{aligned} D &= \lambda \rho.\ (\texttt{pc}\, \rho) = F \wedge (\texttt{ra}\, \rho) = n \\ E &= \lambda \rho.\ (\texttt{pc}\, \rho) = F \wedge \texttt{safe}(\lambda \rho'.(\texttt{pc}\, \rho') = (\texttt{ra}\, \rho)) \\ A &= \lambda \rho.\ (\texttt{pc}\, \rho) = n. \end{aligned}$$

D represents the decoder continuation following the jump. The extension would like to establish that

$$\{E, A\}\, \texttt{covers} \{D\}.$$

In fact, we *can* establish

Lemma 2.3.3. $\texttt{safe}\, E \wedge \texttt{safe}\, A \implies \texttt{safe}\, D$.

Proof. Pick some ρ such that $\rho \models D$. Then $(\texttt{pc}\, \rho) = F$ and $(\texttt{ra}\, \rho) = n$. Since $\texttt{safe}\, A$ we have that

$$\texttt{safe}(\lambda \rho'.(\texttt{pc}\, \rho') = (\texttt{ra}\, \rho)).$$

Thus $\rho \models E$. Since $\texttt{safe}\, E$, we have that $\texttt{prog}\, \rho$. It follows that $\texttt{safe}\, D$, completing the proof. □

However, the coverage claim is stronger, that

$$\forall i.\, \texttt{safe}_i\, E \wedge \texttt{safe}_i\, A \implies \texttt{safe}_i\, D.$$

This stronger claim does not hold. For a counterexample, consider a safety policy in which one step of safe progress can be made when the program counter is n, but no safe progress can be made when the program counter is F, and moreover there is no state from which two steps of safe progress are possible. Then E is contradictory (since its internal claim of indefinite safety cannot hold), and so is $\texttt{safe}_i$ for all i. In particular $\texttt{safe}_1\, E$, $\texttt{safe}_1\, A$, and yet not $\texttt{safe}_1\, D$.

It is necessary to re-write the formal definitions such that

1. the soundness theorem is provable;

2. it is possible to *use* internal safety claims of locinvs (as for the function return example above);

3. it is possible to *establish* internal safety claims of locinvs (as for the function call example above).

I have not discovered a way to do this without using an *indexed* notion of locinv satisfaction, where a locinv can be treated as a *natural-number indexed predicate of states*.

A similar notion of indexing was independently but previously introduced by Appel and McAllester in [6].

2.3.2 Indexed Local Invariants

So let us assume that there is some type

$$\texttt{locinv} : \texttt{Type},$$

together with an *indexed* satisfaction relation

$$\models : \texttt{nat} \to \texttt{state} \to \texttt{locinv} \to \texttt{Prop},$$

where $\rho \models_i C$ is read "ρ satisfies C at index i". Note that it would be misleading to say "ρ satisfies C for i steps", as if it were being claimed that certain properties will hold for the states obtained from ρ after several steps of execution. Appel and McAllester ([6]) have used indexing in this way, but I will always work with locinvs where the index is used to claim that certain *other* states can independently make i steps of safe progress. For instance, in the

function call example, I will use a locinv C for the start of a function F such that

$$\rho \models_i C \iff (\texttt{pc}\,\rho) = F \land \forall j \leq i.\, \texttt{safe}_j(R\,\rho),$$

where $(R\,\rho)$ is a locinv about the return state, such that

$$\rho' \models_i (R\,\rho) \iff (\texttt{pc}\,\rho') = (\texttt{ra}\,\rho).$$

In this example, the index i does not refer to steps of progress from the state ρ, but rather from the other state at function return. Thus to repeat, the satisfaction relation should not be understood as "for i steps" but simply as "at index i".

In future examples I will write indexed locinvs directly as the indexed state predicates induced by the satisfaction relation $\models$. For instance the locinv C above is

$$\lambda i.\ \lambda\rho.\ (\texttt{pc}\,\rho) = F \land \forall j \leq i.\, \texttt{safe}_j\big(\lambda i'.\lambda\rho'.(\texttt{pc}\,\rho') = (\texttt{ra}\,\rho)\big).$$

In many examples the index is unused, insofar as the locinv defines the same state predicate at every index. In such cases I will freely omit reference to the index. In other examples it may be understood that the index is used in the same particular way as in C, to index an internal safety claim; when the index is not explicitly important, I will continue to write, for example, C as

$$\lambda\rho.\ (\texttt{pc}\,\rho) = F \land \texttt{safe}\big(\lambda i'.\lambda\rho'.(\texttt{pc}\,\rho') = (\texttt{ra}\,\rho)\big).$$

(In all examples in this thesis, use of `safe` inside of a locinv implies an indexed rather than an unindexed claim.)

It is sensible to assume that the satisfaction relation satisfies the following property of *monotonicity*:

Definition 2.3.4. An indexed satisfaction relation $\models$ is *monotonic* if, for every state ρ and locinv C,

$$\forall i.\ \rho \models_{i+1} C \implies \rho \models_i C.$$

This property is not necessary for soundness, but it is necessary to allow for a correct decoder when using the type of locinvs used in implementation; see Section 2.7.4.

One appropriate instantiation of type `locinv` is simply as indexed state predicates, `nat` $\rightarrow$ `state` $\rightarrow$ `Prop`, where the satisfaction relation is defined so that $\rho \models_i C$ iff $\forall j \leq i.(C\ j\ \rho)$. It is trivial to show that this is monotonic.

To complete the indexed definitions, we must re-define safety:

Definition 2.3.5. A locinv C is *safe for i steps*, written $\texttt{safe}_i\, C$, when

$$\forall \rho.\ \rho \models_i C \implies \texttt{prog}_i\, \rho.$$

and decoder correctness:

Definition 2.3.6. A decoder `decode` satisfies the *decoder correctness property* iff, for each locinv C, where $(P, \mathcal{D}) = \texttt{decode}\, C$,

$$\forall i.\forall \rho.(\rho \models_{i+1} C) \wedge (P\ \rho) \implies (\exists \rho'.\rho \rightsquigarrow \rho') \wedge \Big(\forall \rho'.\rho \rightsquigarrow \rho' \implies \bigvee_{D \in \mathcal{D}} \rho' \models_i D\Big).$$

That is, for any ρ satisfying the input locinv C, if the local safety condition P holds, then progress is possible for at least one step, and the resulting state will satisfy some $D \in \mathcal{D}$, at an index one less than that by which the original state satisfied C.[2] Note especially the change in the index; but see Section 2.7.4, where the definition is reconsidered using the same index in both occurences.

The soundness theorem, including indexing and certain other changes, will be established in Section 2.6. The handling of function calls and returns is worked out fully in Section 4.4.

Indexing should be considered a purely technical device, and in that sense it causes a certain amount of distraction, as all of the proofs get polluted with indices. In Section 2.7 I will show a way to minimize this problem. Conceptually, one's intuitions are usually well-enough served by forgetting about the indexing, and, in particular, treating the internal safety claims of locinvs (such as the claim that the return continuation is safe) as if they were unindexed.

2.4 Augmented Decoder Input

Recall that one of the goals of the decoder is to have sufficiently detailed output that the extension does not have to refer directly to the notion of the state

[2] The property can be generalized in at least two ways. First, we might allow decoders to describe the next states after possibly more than one step—by taking ρ' to be some k steps from ρ and to satisfy D at index $i+1-k$. Second, we might allow the local safety condition P also to be parametrized by the index i. We have not found either generalization useful.

transition relation $\rightsquigarrow$; instead, the decoder can be considered to encode all possible reasoning about $\rightsquigarrow$. In this section I consider a change to the notion of the decoder which will allow the construction of such decoders.

Consider a typical machine where the state transitions can be defined simply in terms of the instruction to be executed in a given state. Usually of course the instruction is encoded in the value stored in memory at the location given by a specified program-counter register. I will define a decoder for a specific such machine in Section 3.3, but for now I put forth this very general example.

Assume that there is a type `inst` of instructions. Any state ρ specifies an instruction $\texttt{instat}\,\rho$. Any $i : \texttt{inst}$ specifies a state transition $\rightsquigarrow_i : \texttt{state} \rightarrow \texttt{state}$. Observe that $\rightsquigarrow_i$ is defined even on input states ρ for which $\texttt{instat}\,\rho \neq i$; this corresponds to the common-sense view of instructions whereby they specify changes to a state independent of the contents of the state. The full state transition $\rightsquigarrow$ is simply given by

$$\rho \rightsquigarrow \rho' \iff \rho \rightsquigarrow_{\texttt{instat}\,\rho} \rho'.$$

Observe that $\rightsquigarrow$ is functional, so progress can always be made; for the sake of the example any execution will be considered safe.

How can we specify a decoder for this machine? The idea is to write the decoder as a strongest-postcondition generator; thus we could have $\texttt{decode}\,C = (\texttt{True}, \{D\})$ where $\rho \models_{i+1} D$ iff

$$\exists \rho_0 . (\rho_0 \rightsquigarrow_{\texttt{instat}\,\rho_0} \rho) \wedge (\rho_0 \models_i C).$$

It is trivial to establish that this decoder satisfies the decoder correctness property. However, it fails to satisfy a meta-logical criterion for the decoder, namely that the decoder be used to replace all reasoning about $\rightsquigarrow$ in the extension's proofs.

The solution I propose is that the extension be required to give an extra piece of information to the decoder, namely that it specify the instruction to be executed. Then the decoder's output can explicitly describe the effects of that instruction in such a way that the extension need not refer directly to $\rightsquigarrow$ to reason about it. Such a decoder is defined for a specific machine in Section 3.3. Here I specify the logical formalism needed to support this.

Definition 2.4.1. For $\tau : \texttt{locinv} \rightarrow \texttt{Type}$, a *$\tau$-augmented locinv* is a pair $(C, \boldsymbol{t})$ where $C : \texttt{locinv}$, and $\boldsymbol{t} : \tau\, C$.

For the instruction example, we want $\tau\ C$ to be the dependent type $\Sigma \boldsymbol{t}$: $\texttt{inst}.(Q\ \boldsymbol{t}\ C)$, where $Q\ \boldsymbol{t}\ C \iff (\forall i.\forall\rho.\ \rho \models_i C \implies \texttt{instat}\,\rho = \boldsymbol{t})$. Then an augmented locinv is an ordinary locinv which specifies the instruction at any state satisfying it (and provides the proof of that fact).

Definition 2.4.2. For τ as before, a τ*-augmented decoder* is a function which takes a τ-augmented locinv and returns a pair, consisting of a state predicate (the local safety condition) and a list of locinvs (the possible next states); thus a decoder is of type

$$(\Sigma C : \texttt{locinv}.(\tau\ C)) \to \big((\texttt{state} \to \texttt{Prop}) \times \texttt{locinv list}\big)$$

The decoder correctness property is the same as before, using the locinv component of the augmented input. Thus the augmentation does not change the property satisfied by the decoder output; it instead restricts the input, and provides extra information to produce the output. Finally the property of *closure under scanning* must be redefined in the obvious way in order to apply to collections of augmented locinvs; see Section 2.6 for the complete definitions.

In the actual implementation, some trusted code automatically generates the augmentations from a syntactic inspection of the extension's results, so no extra work is created for the extension. See Section 3.3.

2.5 Global Invariants

Now consider a program which does not modify its own code, or dynamically create code. In this case the instruction to be executed depends only on the program counter. I now consider how to use the trusted infrastructure to enforce such a restriction. This provides benefits in terms of simplicity of extensions, which no longer have to maintain invariants about the code in memory, which are instead enforced by the trusted infrastructure.

Essentially we would like to have the extension give just the program counter, rather than the instruction to be executed, in the augmented information it gives to the decoder. Then the decoder has to maintain a *global invariant* to the effect that none of the values in the program's code have been altered—that the addresses in the code block have the literal values corresponding to the actual code. This example is worked out in Section 3.3, using the formalism I will now describe.

Assume that the type `locinv` comes with a satisfaction relation indexed not only by natural numbers but by a global invariant of type $\texttt{state} \to \texttt{Prop}$:

$$\models : (\texttt{state} \to \texttt{Prop}) \to \texttt{nat} \to \texttt{state} \to \texttt{locinv} \to \texttt{Prop},$$

where $\rho \models_i^I C$ is read "ρ satisfies C at index i with global invariant I". The following property is needed to motivate the term "global invariant":

Definition 2.5.1. An indexed satisfation relation $\models$ *is correct with the invariant* if for every state ρ, index i and locinv C, and for every state predicate I,

$$(\rho \models_i^I C) \implies (I\ \rho).$$

Observe that this property would hold if we defined some notion of $\rho \models_i C$ independently of the global invariant, and then let

$$\rho \models_i^I C \iff (\rho \models_i C) \wedge (I\ \rho).$$

This is an appropriate intuition, but we may want $\rho \models_i^I C$ to be *weaker* than this; for instance, we may want C to be able to claim the safety (with respect to global invariant I) of other locinvs, and as I gets stronger such safety claims generally get weaker. See Section 2.7.3.

Definition 2.5.2. Let $I : \texttt{state} \to \texttt{Prop}$. A decoder `decode` satisfies the *(generalized) correctness property with invariant* I iff, for any locinv C, where $(P, \mathcal{D}) = \texttt{decode}\, C$,

$$\forall i. \forall \rho. (\rho \models_{i+1}^I C) \wedge (P\ \rho) \implies (\exists \rho'. \rho \rightsquigarrow \rho') \wedge \left(\forall \rho'. \rho \rightsquigarrow \rho' \implies \bigvee_{D \in \mathcal{D}} (\rho' \models_i^I D)\right).$$

That is, for any state ρ satisfying the input locinv C (and the global invariant I), if the local safety condition P holds, then progress is possible for at least one step, and the resulting state will satisfy some $D \in \mathcal{D}$ (and I), at an index one less than that by which the original state satisfied C.

It is easy to extend this definition to augmented decoders; and the notions of `safe` and `covers` also need to be relativized to the global invariant (see Section 2.6).

The decoder can be expected to enforce the global invariant by means of the local safety conditions, for instance allowing that a memory write is safe only if it doesn't alter the program's code. It may be useful to allow the extension to add facts to the global invariant, as long as those facts are automatically preserved (i.e. without needing the decoder to enforce them); this is discussed in Section 3.1.5. In earlier work [36], Necula and I explored the possibility of using extension-specific global invariants, where each extension actually provides its own decoder and proves it correct with respect to its own global invariant. In this thesis I instead propose that the extension directly does the work of showing that its specific invariants are preserved, by incorporating them into each local invariant. The decoder-enforced global-invariant framework given here is then only used for invariants, namely preservation of the program's code block, which are intended to be used by all extensions.

2.6 The Soundness Theorem

Now I will present the logical formalism implemented by the Open Verifier in its complete form. A Coq implementation of the results of this section is available.

We must begin with a logical correspondent to the notion of safe execution of a program. So let us assume the existence of a type

$$\texttt{state} : \texttt{Set}$$

of machine states, as well as a relation

$$\rightsquigarrow : \texttt{state} \rightarrow \texttt{state} \rightarrow \texttt{Prop}$$

describing *safe* machine transitions. These parameters are intended to encode the machine semantics as well as the safety policy; in particular, $\rightsquigarrow$ is that subset of all machine transitions which are to be considered safe. See Section 3.2 for a discussion of what kinds of safety policies can be encoded in this way.

Definition 2.6.1. A state ρ is capable of i steps of safe progress, written $\texttt{prog}_i\,\rho$, when any chain of $\rightsquigarrow$-transitions starting at ρ can be extended to at least i steps long. That is,

$$\begin{array}{l} \texttt{prog}_0\,\rho \text{ always holds;} \\ \texttt{prog}_{i+1}\,\rho \iff (\exists \rho'.\, \rho \rightsquigarrow \rho') \wedge (\forall \rho'.\, \rho \rightsquigarrow \rho' \implies \texttt{prog}_i\,\rho'). \end{array}$$

By $\texttt{prog}\,\rho$ is meant $\forall i.\, \texttt{prog}_i\, \rho$.

I observe here again that $\leadsto$ only needs to model the safe transitions to be made under the control of the program to be verified; for example it need not reflect actual machine behavior after control is returned to the operating system, simply whether returning control to the operating system is to be considered safe in any given instance.

The extension will work with a type of verification states, corresponding typically to facts known to hold at some particular point during program execution. I call these *local invariants* and assume the existence of some type

$$\texttt{locinv} : \texttt{Type}.$$

I will not specify the type `locinv`, but assume that there is a satisfaction relation by which any `locinv` can be regarded as a state predicate indexed by a natural number and a state predicate (the global invariant):

$$\models: \texttt{nat} \to (\texttt{state} \to \texttt{Prop}) \to \texttt{state} \to \texttt{locinv} \to \texttt{Prop}.$$

For a state ρ, an index i, a global invariant I, and a locinv C, I write $\rho \models_i^I C$ for the statement that this relation holds, where $\rho \models_i^I C$ is read "ρ satisfies C at index i with global invariant I".

The global invariant is a predicate of states which is intended to hold at all points during the execution of the program. In the implementation we will use a global invariant stating that the code block is stored in memory at a particular location, and has not been modified. The following assumption about $\models$ is necessary for the global invariant to live up to its name:

Assumption 2.6.2. *The relation $\models$ is correct with the invariant in the sense that for every state ρ, index i and locinv C, and for every state predicate I,*

$$(\rho \models_i^I C) \implies (I\ \rho).$$

Although not necessary for the results of this section, I will state here the assumption that the satisfaction relation is monotonic in the index:

Assumption 2.6.3. *The relation $\models$ is monotonic, i.e. for every state ρ and locinv C,*

$$\forall i.\ \rho \models_{i+1}^I C \implies \rho \models_i^I C.$$

This is necessary in order that we are able to use the type of locinvs defined in Section 2.7, as discussed in Section 2.7.4.

It is only the satisfaction relation which matters about locinvs for the theorem I am establishing. The reason for using an abstract type `locinv`, instead of using natural-number indexed state predicates directly, is to emphasize that the soundness theorem will still hold if we restrict the predicates which can be expressed, as is actually done in the implementation (see Section 2.7).

Definition 2.6.4. A locinv C is *safe for i steps with invariant I*, written $\texttt{safe}_i^I\, C$, when

$$\forall \rho.\ (\rho \models_i^I C) \implies \texttt{prog}_i\, \rho.$$

By $\texttt{safe}^I\, C$ is meant $\forall i.\, \texttt{safe}_i^I\, C$.

Safety of a program is proven by establishing that $\forall i.\, \texttt{prog}_i\, \rho_0$ for every possible initial state ρ_0. I shall not introduce any formal type corresponding to the notion of a "program" to be verified. Instead, I will assume that any program determines some logical description of all possible initial states; in fact I will assume that it determines a global invariant I and an initial locinv C_0, such that $\forall i.\ \rho_0 \models_i^I C_0$ holds of every possible initial state ρ_0. (In the implementation, it will be the responsibility of the trusted framework to set up C_0 and I such that this holds.) Thus, safety of the program with initial locinv C_0 can be established by proving $\forall i.\, \texttt{safe}_i^I\, C_0$.

Definition 2.6.5. A list $\mathcal{E}$ of locinvs is said to *cover* a list $\mathcal{D}$ of locinvs with respect to global invariant I, written $\mathcal{E}\ \texttt{covers}^I\, \mathcal{D}$, when

$$\forall i. \Big(\bigwedge_{E \in \mathcal{E}} \texttt{safe}_i^I\, E \Big) \implies \Big(\bigwedge_{D \in \mathcal{D}} \texttt{safe}_i^I\, D \Big).$$

We will require the extension to provide extra information about its locinvs; in particular, a proof that the locinv specifies a particular value for the program counter. This uses the following notion.

Definition 2.6.6. For $\tau : \texttt{locinv} \to \texttt{Type}$, a *$\tau$-augmented locinv* is a pair $(C, \boldsymbol{t})$ where $C : \texttt{locinv}$, and $\boldsymbol{t} : \tau\ C$.

Any augmented locinv can be treated as a locinv by ignoring the augmentation. In the following I will do this without comment, applying the predicates $\models$, `safe`, and `covers` to augmented locinvs and lists of augmented locinvs.

The decoder is that part of the trusted framework which specifies the transition relation $\rightsquigarrow$, in terms of locinvs.

Definition 2.6.7. For τ as before, a *τ-augmented decoder* is a function which takes a τ-augmented locinv and returns a pair, consisting of a state predicate (the local safety condition) and a list of locinvs (the possible next states); thus a decoder is of type

$$(\Sigma C : \texttt{locinv}.(\tau\ C)) \to \big((\texttt{state} \to \texttt{Prop}) \times \texttt{locinv list}\big)$$

Definition 2.6.8. A τ-augmented decoder `decode` satisfies the *decoder correctness property with invariant I* iff, for any τ-augmented locinv $(C, \boldsymbol{t})$, where $(P, \mathcal{D}) = \texttt{decode}(C, \boldsymbol{t})$,

$$\forall i.\forall \rho.(\rho \models^{I}_{i+1} C) \wedge (P\ \rho) \implies (\exists \rho'.\rho \rightsquigarrow \rho') \wedge \Big(\forall \rho'.\rho \rightsquigarrow \rho' \implies \texttt{prog}\,\rho' \vee \big(\bigvee_{D \in \mathcal{D}} \rho' \models^{I}_{i} D\big)\Big).$$

That is, for any state ρ satisfying the input locinv C (and the global invariant I), if the local safety condition P holds, then progress is possible for at least one step, and the resulting state is either safe (can make indefinite safe progress), or will satisfy some $D \in \mathcal{D}$ (and I), at an index one less than that by which the original state satisfied C.

Observe that we will allow the decoder to omit safe continuations from its output $\mathcal{D}$, insofar as $\mathcal{D}$ can exclude ρ' for which $\texttt{prog}\,\rho'$. The remaining continuations will be those the safety of which requires a proof obligation for the untrusted extension. This can be used by the implementation to eliminate cases which are known to the trusted framework to result in a safe abort, see Section 4.3.4.2.

The following is the key property to be established by the untrusted extension.

Definition 2.6.9. A set $\mathcal{E}$ of τ-augmented locinvs is *closed under scanning with respect to τ-augmented decoder* `decode` *and global invariant I*, if the following two conditions hold for each $E \in \mathcal{E}$. Let $(P, \mathcal{D}) = \texttt{decode}\,E$. Then

1. $\forall i.\forall \rho.(\rho \models^{I}_{i} E) \implies (P\ \rho)$;

2. $\mathcal{E}\,\mathtt{covers}^I\,\mathcal{D}$.

Lemma 2.6.10. *Let* $I : \mathtt{state} \to \mathtt{Prop}$*, and* $\mathtt{decode}$ *be a* τ*-augmented decoder which satisfies the decoder correctness property with invariant* I*. If a set* $\mathcal{E}$ *of* τ*-augmented locinvs is closed under scanning with respect to* $\mathtt{decode}$ *and* I*, then for each* $E \in \mathcal{E}$*, we have that* $\forall i.\, \mathtt{safe}_i^I\, E$*.*

Proof. The proof proceeds by induction on the index i. The base case $i = 0$ holds trivially. So suppose as induction hypothesis that for each $E \in \mathcal{E}$, $\mathtt{safe}_i^I\, E$. We will establish that for each $E \in \mathcal{E}$, $\mathtt{safe}_{i+1}^I\, E$, which will complete the proof by induction.

Fix some $E \in \mathcal{E}$. To show $\mathtt{safe}_{i+1}^I\, E$, by definition we must show that for every ρ such that $\rho \models_{i+1}^I E$, $\mathtt{prog}_{i+1}\,\rho$. Fix some ρ such that $\rho \models_{i+1}^I E$.

Let $(P, \mathcal{D}) = \mathtt{decode}\, E$. By the decoder correctness property, instantiated at our chosen ρ and i,

$$(\rho \models_{i+1}^I E) \land (P\ \rho) \implies \\ (\exists \rho'. \rho \rightsquigarrow \rho') \land \Big(\forall \rho'. \rho \rightsquigarrow \rho' \implies \mathtt{prog}\,\rho' \lor \big(\bigvee_{D \in \mathcal{D}} . \rho' \models_i^I D\big)\Big).$$

By condition (1) of closure under scanning, we have that $(P\ \rho)$. Therefore it is possible to make one step of progress from ρ; and any resulting state ρ' either satisfies $\mathtt{prog}\,\rho'$, or else will satisfy some $D \in \mathcal{D}$ (and the global invariant I).

Suppose the second case. We now use condition (2) of closure under scanning, which has that $\mathcal{E}\,\mathtt{covers}^I\,\mathcal{D}$. By the induction hypothesis, $\mathtt{safe}_i^I\, E$ for each $E \in \mathcal{E}$. Thus by the definition of $\mathtt{covers}$, $\mathtt{safe}_i^I\, D$ for each $D \in \mathcal{D}$. By the definition of $\mathtt{safe}^I$ we can conclude that $\mathtt{prog}_i\,\rho'$ for any ρ' satisfying some $D \in \mathcal{D}$ at index i and invariant I.

Now for each ρ' resulting from our chosen ρ after one step of execution, either we have $\mathtt{prog}\,\rho'$ directly from the decoder correctness property, or else the above condition holds; either way we have $\mathtt{prog}_i\,\rho'$. Thus we have established $\mathtt{prog}_{i+1}\,\rho$, completing the proof. ☐

Theorem 2.6.11 (Soundness of the Open Verifier). *Let* $I : \mathtt{state} \to \mathtt{Prop}$*, and* $\mathtt{decode}$ *be a* τ*-augmented decoder which satisfies the decoder correctness property with invariant* I*. Suppose a set* $\mathcal{E}$ *of* τ*-augmented locinvs is closed under scanning with respect to* $\mathtt{decode}$ *and* I*.*

Suppose also that $\mathcal{E}\ \texttt{covers}^I\{C_0\}$*, where* C_0 *is an initial locinv for a program in the sense that for any possible initial state* ρ_0*,* $\rho_0 \models_i^I C_0$ *for all* i*. Then the program is safe in the sense that each such* ρ_0 *can make indefinite safe progress.*

Proof. By the lemma $\forall i.\, \texttt{safe}_i^I\, E$ for each $E \in \mathcal{E}$. Thus by the definition of $\texttt{covers}^I$, $\forall i.\, \texttt{safe}_i^I\, C_0$. The rest follows by the definitions of safety and progress. □

In the following Section 2.7 I will instatiate the type `locinv` and its satisfaction relation $\models$. Once that is done, the role of the untrusted extension can be understood as producing a list of locinvs, which covers an initial locinv for the program to be satisfied, and proving that it is closed under scanning with respect to some correct decoder.

2.7 The Type Locinv

Recall that for the purposes of soundness, the use of locinvs is just that they induce a state predicate indexed by natural numbers and by the global invariant, via the relation

$$\models : \texttt{nat} \to (\texttt{state} \to \texttt{Prop}) \to \texttt{state} \to \texttt{locinv} \to \texttt{Prop},$$

with the only requirement being that the global invariant in fact holds, i.e.

$$(\rho \models_i^I C) \implies (I\ \rho).$$

To use the soundness theorem, nothing prevents us from simply using indexed state predicates. For various implementation reasons, however, we have found it useful to *restrict* the expressiveness of locinvs. This is explored in the following.

2.7.1 Existential Variables, Registers, and Assumptions

The first optimization is to note that we can remove the need to reason explicitly about predicates of *states*, by requiring locinvs to have the form (omitting for now the indexing):

$$\lambda\rho.\ \exists \boldsymbol{x} : \tau.\ \big(\rho = f(\boldsymbol{x})\big) \wedge (A\ \boldsymbol{x}).$$

Using this scheme, when the extension has to prove a local safety condition

$$\forall \rho.(\rho \models C) \implies (P\ \rho),$$

it suffices to prove

$$\forall \boldsymbol{x} : \tau.\, (A\ \boldsymbol{x}) \implies \big(P\ (f\ \boldsymbol{x})\big).$$

On actual systems machine states (type `state`) will generally consist of a register file and a memory; `state` can be imagined as a type of tuples, with many components (the registers) of type `val` and one (the memory) of type `mem`. Often the memory can be treated analogously with the registers and so I often use the term "registers" to refer to all of the components of `state`. The function f in the locinv above can then be said to specify the "registers".

The type τ is often referred to in the plural as the "existential variables" of the locinv; τ is generally a large tuple, with various components for particular values we wish to existentially quantify, which can in fact be thought of as many individual variables. The predicate A is referred to in the plural as the "assumptions" of the locinv.

For the implementation we have the decoder output a local safety condition which is already a function of τ, using the registers f specified by the input locinv to translate $(P\ \rho)$ directly into $(P\ (f\ \boldsymbol{x}))$.

2.7.2 Progress Continuations

The other refinement I propose has to do with the notion of *indexing*. The original motivation for indexing is for the handling of function calls and returns; the locinv for the start of a function makes the claim that it is safe to jump to the return address, which is implemented as a locinv C such that

$$\rho \models_i C \iff (\texttt{pc}\,\rho) = F \wedge \forall j \leq i.\, \texttt{safe}_j(\mathit{Ret}\,\rho)$$

where for any ρ, $(\mathit{Ret}\,\rho)$ is a locinv such that

$$\rho' \models_i (\mathit{Ret}\,\rho) \iff (\texttt{pc}\,\rho') = (\texttt{ra}\,\rho).$$

The extension can use C to cover a decoder continuation, only if the extension can establish that the return address has been correctly set up to somewhere that we know is safe to jump to.

In actual fact, during the execution of any program which we can verify, in which C is the locinv for line F, any ρ that comes to line F will satisfy the stronger, unindexed property that

$$(\texttt{pc}\,\rho) = F \wedge \texttt{safe}_\infty(\mathit{Ret}\,\rho).$$

Of course, this matches precisely the suggested intuition of the requirement for a function call. As discussed in Section 2.3, it is only that it turns out to be difficult to establish all the required facts using unindexed claims. To make the proof go through, we need to generalize the locinvs which contain safety claims about other locinvs, such that they contain indexed safety claims instead.

Thus the indexing is a *purely technical device*; and having the indices everywhere is distracting as well. So I will now propose a refinement of the system such that the extensions never have to reason about indices. Essentially, locinvs are restricted so that the only allowable use of indexing is in safety claims about other locinvs.

Note that this does restrict the use of the system. Appel and McAllester in [6] create a type system using a similar notion of indexing, in which the indexing is used to define recursive types, by expressing at index i that a state can make i steps of progress, at each step satisfying some predicate. The refined version of locinv cannot express this kind of indexed predicate. We have found a more fruitful approach to recursive types is to break the recursion by means of intensional typing (see Section 4.2.1.1).

2.7.3 Locinvs, Again

I can now define a type of locinvs using the above ideas.

Definition 2.7.1. A *locinv* C is inductively defined as the type of tuples

$$(C.\texttt{type}, C.\texttt{regs}, C.\texttt{assume}, C.\texttt{progress}),$$

where

$$\begin{aligned} C.\texttt{type} &: \texttt{Set}; \\ C.\texttt{regs} &: C.\texttt{type} \to \texttt{state}; \\ C.\texttt{assume} &: C.\texttt{type} \to \texttt{Prop}; \\ C.\texttt{progress} &: (C.\texttt{type} \to \texttt{locinv})\ \texttt{list}. \end{aligned}$$

The notions of satisfaction and safety are defined recursively on the structure of locinvs, as follows. Let $I : \texttt{state} \to \texttt{Prop}$. A state ρ *satisfies* a locinv C *at*

index i with global invariant I, written $\rho \models_i^I C$, when

$$(I\ \rho) \wedge \exists \boldsymbol{x} : C.\mathtt{type}.\ \rho = (C.\mathtt{regs}\ \boldsymbol{x}) \wedge \\ (C.\mathtt{assume}\ \boldsymbol{x}) \wedge \bigwedge_{P \in C.\mathtt{progress}} \forall j \leq i.\ \mathtt{safe}_j^I (P\ \boldsymbol{x});$$

a locinv C is *safe for i steps with invariant I*, written $\mathtt{safe}_i^I\ C$, when

$$\forall \rho. (\rho \models_i^I C) \implies \mathtt{prog}_i\ \rho.$$

By $\mathtt{safe}^I\ C$ is meant $\forall i.\ \mathtt{safe}_i^I\ C$.[3]

Observe that it is logically important that locinvs be an inductive type, and that the definition of $\models$ be recursive over that inductive structure. Otherwise there would be a circularity in the definition. Since we only allow well-founded locinvs, it can be imagined that $\models$ is first defined (non-recursively) for locinvs without a `progress` field; and then for locinvs whose `progress` locinvs come from that class; and so on.

In the implementation we have found it efficient to store the `assume` field as a *list* of predicates, which are joined by conjunction to create a single predicate. Because of this I will refer to C.`assume` in the plural as the *assumptions* of C. Similarly, C.`type` is implemented as a tuple type, and so I refer to it as the *existential variables* of C. As mentioned above, the notion of `state` is implemented as a tuple of individual registers, motivating the name of the field C.`regs` as the *registers* of C.[4]

The elements of C.`progress` are called C's *progress continuations*. They embody claims that safe progress can be made by continuing execution in a state satisfying certain conditions. The example to keep in mind is the continuation, stating that it is safe to continue if the `pc` is set to the value of the return address register at the start of execution of the function.

[3]Technically, the notion of safety used in the soundness theorem is defined *after* the satisfaction relation $\models$; but of course that definition will coincide with the notion of safety defined *concurrently* with $\models$ here.

[4]In our prototype implementation, the program counter is stored separately from the other registers, giving a fifth component to the type of locinvs. This is mostly due to the fact that the value of the program counter is the augmentation sent to the decoder, to enable the decoder to find the instruction to be executed. The separation is not necessary for the formal development, but see Section 3.1.7 for related implementation issues.

Observe that the only dependence on the index i occurs in the claims that the progress continuations are safe for i steps.

We will need the following:

Lemma 2.7.2. $\models$ *is monotonic, i.e. for any* I, i, ρ, *and* C,

$$\rho \models^I_{i+1} C \implies \rho \models^I_i C.$$

Proof. Using that the only dependence on the index in the definition of $\models$ is with the progress continuation, this follows from the fact that

$$\bigwedge_{P \in C.\texttt{progress}} \forall j \leq i+1.\, \texttt{safe}^I_j(P\ \boldsymbol{x}) \implies \bigwedge_{P \in C.\texttt{progress}} \forall j \leq i.\, \texttt{safe}^I_j(P\ \boldsymbol{x}).$$

□

Finally, two notes on notation. First, it is often convenient to use a mapping notation with the progress continuations. Thus given a locinv D, and $\boldsymbol{x}$: $D.\texttt{type}$, I will write

$$(D.\texttt{progress}\ \boldsymbol{x})$$

for

$$\{(R\ \boldsymbol{x}) \mid R \in D.\texttt{progress}\}.$$

Second, in examples it is often more convenient to write locinvs as state predicates, rather than specifying the four fields directly. I will suppress the global invariant and indexing and write, for instance,

$$\lambda\rho.\ \exists\tau.\ (A\ \tau\ \rho) \wedge \texttt{safe}(R\ \tau\ \rho).$$

This should be understood as specifying a locinv C where

$$\begin{aligned} C.\texttt{type} &= \tau \times \texttt{state}; \\ C.\texttt{regs} &= \lambda(\boldsymbol{t}, \rho).\ \rho; \\ C.\texttt{assume} &= \lambda(\boldsymbol{t}, \rho).\ (A\ \boldsymbol{t}\ \rho); \\ C.\texttt{progress} &= \{\lambda(\boldsymbol{t}, \rho).\ (R\ \tau\ \rho)\}. \end{aligned}$$

Note that the `regs` field is always boring under than translation. In order to improve the efficiency of automated theorem proving, we have found it better to have equalities among registers reflected directly in the `regs` field rather than in the `assume` field.

2.7.3.1 A Final Restriction

For completeness I note here one more restriction on the type of locinvs, used in the implementation. Logically it is used in this thesis only in Lemma 2.7.11, though the lemma can be weakened only slightly to avoid it. As defined above, the progress continuations of a locinv can depend on the host locinv's existential variables in any conceivable way; but in practice the only use of this is to include the host's existential variables among the progress continuation's, as I now define.

Definition 2.7.3. For $\tau : \mathtt{Set}$ let a τ-dependent locinv, type ($\mathtt{locinv'}\ \tau$), be the type of tuples

$$(C.\mathtt{type'}, C.\mathtt{regs}, C.\mathtt{assume}, C.\mathtt{progress}),$$

where

$$\begin{aligned} C.\mathtt{type'} &: \mathtt{Set}; \\ C.\mathtt{regs} &: \tau \times C.\mathtt{type'} \to \mathtt{state}; \\ C.\mathtt{assume} &: \tau \times C.\mathtt{type'} \to \mathtt{Prop}; \\ C.\mathtt{progress} &: (\mathtt{locinv'}\ (\tau \times C.\mathtt{type'}))\ \ \mathtt{list}. \end{aligned}$$

The type $\mathtt{locinv}$ of top-level locinvs can be identified with $\mathtt{locinv'}\ \mathtt{unit}$. For $R : \mathtt{locinv'}\ (\tau \times \tau')$ and $x : \mathtt{tau}$, let $(R\ x) : \mathtt{locinv'}\ \tau'$ be defined recursively as

$$\begin{aligned} (R\ x).\mathtt{type'} &= R.\mathtt{type'}; \\ (R\ x).\mathtt{regs} &= \lambda(y, z).\ (R.\mathtt{regs}\ (x, y, z)); \\ (R\ x).\mathtt{assume} &= \lambda(y, z).\ (R.\mathtt{assume}\ (x, y, z)); \\ (R\ x).\mathtt{progress} &= \{(S\ x) \mid S \in R.\mathtt{progress}\} \end{aligned}$$

In particular, any progress continuation of a locinv C can be considered as a function $C.\mathtt{type} \to \mathtt{locinv}$, as before.

Using that last fact, we can pretend that locinvs are as defined in Definition 2.7.1, only that the progress continuations are restricted so that they depend on the existential variables in a particularly simple way. In Lemma 2.7.11 I will make use of the fact that there is a sensible notion of $R.\mathtt{progress}$ for for each $R \in D.\mathtt{progress}$, not just $(R\ \boldsymbol{x}).\mathtt{progress}$ for some $\boldsymbol{x} : D.\mathtt{type}$.

Finally note that given $f : \tau \to \tau'$, any C : `locinv'` τ' can be lifted to a `locinv'` τ. I will write this as $\lambda x.\ (C\ (f\ x))$, defining $D = \lambda x.\ (C\ (f\ x))$ recursively by

$$\begin{aligned} D.\texttt{type'} &= C.\texttt{type'}; \\ D.\texttt{regs} &= \lambda(x,y).\ (C.\texttt{regs}\ (f\ x, y)); \\ D.\texttt{assume} &= \lambda(x,y).\ (C.\texttt{assume}\ (f\ x, y)); \\ D.\texttt{progress} &= \{\lambda(x,y).(S\ (f\ x, y)) \mid S \in R.\texttt{progress}\}. \end{aligned}$$

2.7.4 The Decoder

There is a bit of a surprise related to the decoder. One might imagine that the decoder should be not only *correct* but *complete*, in that for instance the output continuations $\mathcal{D}$ should describe exactly those states which can result from executing one step on some state described by the input locinv.

Recall the definition of decoder correctness (here omitting the global invariant):

Definition 2.7.4. A decoder `decode` satisfies the *decoder correctness property* iff, for each locinv C, where $(P, \mathcal{D}) = \texttt{decode}\, C$,

$$\forall i.\forall \rho.(\rho \models_{i+1} C) \wedge (P\ \rho) \implies (\exists \rho'.\rho \rightsquigarrow \rho') \wedge \big(\forall \rho'.\rho \rightsquigarrow \rho' \implies \bigvee_{D \in \mathcal{D}} \rho' \models_i D\big).$$

That is, for any state ρ satisfying the input locinv C, if the local safety condition P holds, then progress is possible for at least one step, and the resulting state will satisfy some $D \in \mathcal{D}$, at an index one less than that by which the original state satisfied C.

But if the input locinv states that, at index $i+1$, some locinv R is $\texttt{safe}_{i+1}$, then a complete decoder would, in its output continuations, need to say *at index* i that R is $\texttt{safe}_{i+1}$. This is impossible according to the definition of locinv I have now given.

The decoder we will use (defined in Section 3.3) satisfies the following stronger property.

Definition 2.7.5. A decoder `decode` satisfies the *strong decoder correctness property* iff, for each locinv C, where $(P, \mathcal{D}) = \texttt{decode}\, C$,

$$\forall i.\forall \rho.(\rho \models_{i+1} C) \wedge (P\ \rho) \implies (\exists \rho'.\rho \rightsquigarrow \rho') \wedge \big(\forall \rho'.\rho \rightsquigarrow \rho' \implies \bigvee_{D \in \mathcal{D}} \rho' \models_{i+1} D\big).$$

That is, for any state ρ satisfying the input locinv C, if the local safety condition P holds, then progress is possible for at least one step, and the resulting state will satisfy some $D \in \mathcal{D}$, *at the same index* by which the original state satisfied C.

The strong correctness property implies the original correctness property, by the monotonicity of $\models$. Thus, we can work with our strong decoder and still rely on the soundness theorem (Theorem 2.6.11). This is the only technical reason for requiring that $\models$ be monotonic.

This "slackness", where the decoder satisfies a property stronger than that required by the soundness theorem, is an indication that I have given up some expressive power by this restriction on locinvs. I will show in Chapter 4 that locinvs are nonetheless powerful enough to express the predicates needed for serious verification projects. I propose that the loss of expressive power will be worth it for the simplicity, gained by not having to reason about indexing, to the writers of extensions.

2.7.5 Coverage Proof Rules

To actually remove the need for reasoning about indices requires providing means for the extension to establish the coverage proof obligations, such that the proofs do not involve indexing. Recall the definition

Definition 2.7.6. A list $\mathcal{E}$ of locinvs is said to *cover* a list $\mathcal{D}$ of locinvs with respect to global invariant I, written $\mathcal{E}\ \texttt{covers}^I\ \mathcal{D}$, when

$$\forall i.\big(\bigwedge_{E \in \mathcal{E}} \texttt{safe}_i^I\ E\big) \implies \big(\bigwedge_{D \in \mathcal{D}} \texttt{safe}_i^I\ D\big).$$

Note the easy lemma

Lemma 2.7.7. *Let* $I : \texttt{state} \rightarrow \texttt{Prop}$*. Let* $\mathcal{E}$ *and* $\mathcal{D}$ *sets of locinvs. Then* $\mathcal{E}\ \texttt{covers}^I\ \mathcal{D}$ *if and only if for each* D *in* $\mathcal{D}$*,* $\mathcal{E}\ \texttt{covers}^I \{D\}$*.*

Proof. Immediate. □

By means of this lemma, we can restrict our attention to the notion of covering a single locinv D.

Another useful easy lemma is

Lemma 2.7.8. *Let $\mathcal{D}$, $\mathcal{E}$, and $\mathcal{E}'$ be sets of locinvs such that $\mathcal{E} \subseteq \mathcal{E}'$. Then*

$$\mathcal{E}\,\texttt{covers}^I\,\mathcal{D} \implies \mathcal{E}'\,\texttt{covers}^I\,\mathcal{D}.$$

Thus, having established a proof of coverage with one set $\mathcal{E}$ of locinvs, coverage will also hold with any larger set $\mathcal{E}'$.

Proof. Immediate. □

I now re-express coverage using the refined notion of locinv.

Lemma 2.7.9. *Let $I : \texttt{state} \to \texttt{Prop}$. Let D be a locinv and $\mathcal{E}$ a set of locinvs. Then $\mathcal{E}\,\texttt{covers}^I\{D\}$ if*

$$\begin{aligned}&\forall \boldsymbol{x} : D.\texttt{type}\;.\;(D.\texttt{assume}\;\boldsymbol{x}) \wedge (I\;(D.\texttt{regs}\;\boldsymbol{x})) \implies \\ &\quad \bigvee_{E \in (\mathcal{E} \cup (D.\texttt{progress}\;\boldsymbol{x}))} \exists \boldsymbol{y} : E.\texttt{type}\;.\;(D.\texttt{regs}\;\boldsymbol{x}) = (E.\texttt{regs}\;\boldsymbol{y}) \wedge \\ &\qquad (E.\texttt{assume}\;\boldsymbol{y}) \wedge (\mathcal{E} \cup (D.\texttt{progress}\;\boldsymbol{x})\;\texttt{covers}^I(E.\texttt{progress}\;\boldsymbol{y})).\end{aligned}$$

That is, the coverage claim is implied by the following: Suppose we have an instantiation of D's variables such that D's assumptions hold, and the global invariant I holds at the state specified by D's registers. Then we can choose an E, either from $\mathcal{E}$ or from D's progress continuations, and an instantiation of E's variables, such that the states described by D and E coincide, and E's assumptions hold, and E's progress continuations are safe under the assumption that D's progress continuations are safe.

Proof. Choose an index i and suppose $\bigwedge_{E \in \mathcal{E}} \texttt{safe}_i^I\,E$. I will show $\texttt{safe}_i^I\,D$. So choose ρ such that $\rho \models_i^I D$. I will show $\texttt{prog}_i\,\rho$.

By the definition of $\models$, we have that

$$\begin{aligned}&(I\;\rho) \wedge \exists \boldsymbol{x} : D.\texttt{type}.\;\rho = (D.\texttt{regs}\;\boldsymbol{x}) \wedge \\ &\qquad\qquad (D.\texttt{assume}\;\boldsymbol{x}) \wedge \textstyle\bigwedge_{P \in D.\texttt{progress}} \texttt{safe}_i^I(P\;\boldsymbol{x}).\end{aligned}$$

Fix a $\boldsymbol{x}$ witnessing the existential; then we can deduce from the hypothesis that

$$\bigvee_{E \in (\mathcal{E} \cup (D.\mathrm{progress}\ \boldsymbol{x}))} \exists \boldsymbol{y} : E.\mathtt{type}\ .\ (D.\mathtt{regs}\ \boldsymbol{x}) = (E.\mathtt{regs}\ \boldsymbol{y}) \wedge (E.\mathtt{assume}\ \boldsymbol{y}) \wedge (\mathcal{E} \cup (D.\mathtt{progress}\ \boldsymbol{x})\ \mathtt{covers}^I (E.\mathtt{progress}\ \boldsymbol{y})).$$

Choose E and $\boldsymbol{y}$ for which this is so. Since we have assumed that $\bigwedge_{E \in \mathcal{E}} \mathtt{safe}_i^I\, E$ and we have also that $\bigwedge_{P \in D.\mathrm{progress}} \mathtt{safe}_i^I(P\ \ \boldsymbol{x})$, then we can derive that $\mathtt{safe}_i^I\, E$, and also that $\bigwedge_{R \in E.\mathrm{progress}} \mathtt{safe}_i^I(R\ \boldsymbol{y})$.

But this gives us

$$(I\ \rho) \wedge \exists \boldsymbol{y} : E.\mathtt{type}.\ \rho = (E.\mathtt{regs}\ \boldsymbol{y}) \wedge (E.\mathtt{assume}\ \boldsymbol{y}) \wedge \bigwedge_{R \in E.\mathrm{progress}} \mathtt{safe}_i^I(R\ \boldsymbol{y}),$$

in other words that $\rho \models_i^I E$. Together with $\mathtt{safe}_i^I\, E$, this yields $\mathtt{prog}_i\, \rho$, which completes the proof. □

Observe that to prove a coverage obligation by means of this lemma, one will typically be required to prove other coverages, of each progress continuation R of the covering locinv E. However, these coverage proofs (1) happen in a context in which D's assumptions hold, and (2) have D's progress continuations additionally available as candidates for the covering. We can take advantage of the fact that a locinv is a higher-order entity so that all coverage subproofs happen without additional context and with the same covering set; intuitively all coverage proofs will happen at "top level". This offers significant advantages in efficiency and simplicity for the implementation.

The rough idea is to expand R so that it contains D's assumptions and progress continuations. I will now make this precise.

Definition 2.7.10. The function

$$\mathtt{Acc} : (D : \mathtt{locinv})(\tau_E : \mathtt{Set})(f : D.\mathtt{type} \to \tau_E)(R : \tau_E \to \mathtt{locinv})\mathtt{locinv}$$

is defined as follows: $\mathtt{Acc}\, D\ f\ R$ is the locinv C where

$$\begin{aligned} C.\mathtt{type} &= \Sigma x : D.\mathtt{type}.\ (R\ (f\ x)).\mathtt{type} \\ C.\mathtt{regs} &= \lambda(x,y).\ (R\ (f\ x)).\mathtt{regs}\ y \\ C.\mathtt{assume} &= \lambda(x,y).\ (D.\mathtt{assume}\ x) \wedge ((R\ (f\ x)).\mathtt{assume}\ y) \\ C.\mathtt{progress} &= \lambda(x,y).\ (D.\mathtt{progress}\ x) \cup ((R\ (f\ x)).\mathtt{progress}\ y). \end{aligned}$$

This definition can satisfy the restriction on progress continuations introduced in Section 2.7.3.1, as long as R has the restricted form (i.e. R : $\texttt{locinv'}\ \tau_E$). In that case we should properly write

$$\begin{aligned} C.\texttt{type} &= D.\texttt{type} \times R.\texttt{type'} \\ C.\texttt{regs} &= \lambda(x,y).\, R.\texttt{regs}\ (f\ x, y) \\ C.\texttt{assume} &= \lambda(x,y).\, (D.\texttt{assume}\ x) \wedge (R.\texttt{assume}\ (f\ x, y)) \\ C.\texttt{progress} &= \{\lambda(x,y).\ (P\ x) \mid P \in D.\texttt{progress}\} \cup \\ &\quad\ \{\lambda(x,y).\ (P\ (f\ x, y)) \mid P \in R.\texttt{progress}\} \end{aligned}$$

The locinv $\texttt{Acc}\, D\ f\ R$ is formed by "accumulating" the assumptions and progress continuations of D, into the progress continuation R, lifting it to a bona fide locinv by the instantiating function f.

In the following the notation $\texttt{Acc}\, D\ f\ \mathcal{E}$ is used for $\{\texttt{Acc}\, D\ f\ E \,|\, E \in \mathcal{E}\}$.

Lemma 2.7.11. *Let* I : $\texttt{state} \to \texttt{Prop}$*. Let* D *be a locinv and* $\mathcal{E}$ *a set of locinvs. Then* $\mathcal{E}\ \texttt{covers}^I \{D\}$ *if there is a covering continuation*

$$E' \in \{\lambda_.E \,|\, E \in \mathcal{E}\} \cup D.\texttt{progress}$$

and some instantiating function

$$f : \Pi\boldsymbol{x} : D.\texttt{type}.(E'\ \boldsymbol{x}).\texttt{type}$$

such that the following two facts hold: for all $\boldsymbol{x}$: $D.\texttt{type}$*, letting* $E = (E'\ \boldsymbol{x})$*,*

$$\begin{aligned} &(D.\texttt{assume}\ \boldsymbol{x}) \wedge (I\ (D.\texttt{regs}\ \boldsymbol{x})) \implies \\ &\qquad (D.\texttt{regs}\ \boldsymbol{x}) = (E.\texttt{regs}\ (f\ \boldsymbol{x})) \wedge (E.\texttt{assume}\ (f\ \boldsymbol{x})) \end{aligned}$$

and, letting $f'\ \boldsymbol{x} = (\boldsymbol{x}, f\ \boldsymbol{x})$*,*

$$\mathcal{E}\ \texttt{covers}^I\ \texttt{Acc}\, D\ f'\ (E'.\texttt{progress}).$$ [5]

---

[5] *To write* $E'.\texttt{progress}$ *I am using the restriction of Section 2.7.3.1. Otherwise I would need to use* $\texttt{Acc}\, D\ f\ (E.\texttt{progress})$*; then this coverage subproof would still not quite be at "top level" in that it would be parametric in* $\boldsymbol{x} : D.\texttt{type}$*.*

Proof. I proceed by Lemma 2.7.9, where for any $\boldsymbol{x}$ the choice of E is given by the hypothesis, and $\boldsymbol{y} = (f\ \boldsymbol{x})$. The only difficult part is to establish that

$$\begin{aligned}&\mathcal{E}\ \texttt{covers}^I\ \texttt{Acc}\, D\ f'\ (E'.\texttt{progress}) \implies \\ &\qquad\qquad \forall \boldsymbol{x} : D.\texttt{type}.\ \mathcal{E} \cup (D.\texttt{progress}\ \boldsymbol{x})\ \texttt{covers}^I (E.\texttt{progress}\ (f\ \boldsymbol{x})).\end{aligned}$$

So fix $\boldsymbol{x}$ and let $\boldsymbol{y} = (f\ \boldsymbol{x})$. Suppose, for some index i, that $\bigwedge_{E\in\mathcal{E}} \texttt{safe}^I_i\, E$ and that $\bigwedge_{P\in D.\texttt{progress}} \texttt{safe}^I_i(P\ \boldsymbol{x})$. Choose any R in $E.\texttt{progress}$; note that $R = (R'\ \boldsymbol{x})$ for $R' \in E'.\texttt{progress}$. We must show that $\texttt{safe}^I_i(R\ \boldsymbol{y})$.

Choose some ρ such that $\rho \models^I_i (R\ \boldsymbol{y})$; we must show that $\texttt{prog}_i\, \rho$. Let $\boldsymbol{z}$ be the instantiation of $(R\ \boldsymbol{y}).\texttt{type}$ which witnesses that $\rho \models^I_i (R\ \boldsymbol{y})$. Letting $C = \texttt{Acc}\, D\ f'\ R'$, and using that $(R\ (f\ x)) = (R'\ (x, f\ x))$, we have that

$$\begin{aligned} C.\texttt{type} &= \Sigma x : D.\texttt{type}.\ (R\ (f\ x)).\texttt{type} \\ C.\texttt{regs} &= \lambda(x,y).\ (R\ (f\ x)).\texttt{regs}\ y \\ C.\texttt{assume} &= \lambda(x,y).\ (D.\texttt{assume}\ x) \wedge ((R\ (f\ x)).\texttt{assume}\ y) \\ C.\texttt{progress} &= \lambda(x,y).\ (D.\texttt{progress}\ x) \cup ((R\ (f\ x)).\texttt{progress}\ y). \end{aligned}$$

In particular, for the instantiation $\boldsymbol{t} = (\boldsymbol{x}, \boldsymbol{z})$, we have that $\rho = C.\texttt{regs}\ \boldsymbol{t}$, and that $C.\texttt{assume}\ \boldsymbol{t}$. To establish that $\rho \models^I_i C$ we must just establish the safety of its progress continuations. But we already have that $(D.\texttt{progress}\ \boldsymbol{x})$ are all safe; and $(R\ \boldsymbol{y}).\texttt{progress}\ \boldsymbol{z}$ are all safe by the fact that $\rho \models^I_i (R\ \boldsymbol{y})$.

Thus $\rho \models^I_i C$. But, under that assumption that $\bigwedge_{E\in\mathcal{E}} \texttt{safe}^I_i\, E$, $\texttt{safe}^I_i\, C$ by hypothesis. Thus $\texttt{prog}_i\, \rho$. This completes the proof. ☐

The statement of the following lemma is somewhat technical, but it simply embodies the claim that if E has a progress continuation R which is essentially the same as a progress continuation of D, then the coverage of $\texttt{Acc}\, D\ f\ R$ is automatic. Intuitively, this is because D already claims the safety of R, and the accumulation into R of any extra assumptions from D won't affect that.

Lemma 2.7.12. *Let D be a locinv such that $R \in D.\texttt{progress}$. Suppose that $f : D.\texttt{type} \to \tau$ and $g : \tau \to D.\texttt{type}$ have the property that $(g\ (f\ x)) = x$ for all x.*

Then for any I

$$\{\}\ \texttt{covers}^I \big\{\texttt{Acc}\, D\ f\ \big(\lambda z.(R\ (g\ z))\big)\big\}.$$

Proof. It is possible to prove this using Lemma 2.7.11 by induction on the structure of locinvs, but easier to proceed by returning to the original definition of `covers`; so let $C = \left\{\texttt{Acc}\, D\; f\; \left(\lambda z.(R(g\; z))\right)\right\}$. We must prove $\texttt{safe}_i^I\, C$.

So pick some ρ such that $\rho \models_i^I C$. By the definition of $\models$ there is some $\boldsymbol{x} : C.\texttt{type}$ witnessing this; let $\boldsymbol{x} = (w, y)$ where $w : D.\texttt{type}$, $y : (R\; (g\; (f\; w))).\texttt{type} = (R\; w).\texttt{type}$.

By inspection of the definition of $\models$ and `Acc`, observe that $\rho \models_i^I C$ implies $\texttt{safe}_i^I (E\; w)$ for any $E \in D.\texttt{progress}$; and so $\texttt{safe}_i^I (R\; w)$. But by the definition of `Acc`, it is clear that $\rho \models_i^I C$ (with $\boldsymbol{x} = (w, y)$ witnessing the existential) implies $\rho \models_i^I (R\; w)$ (with y witnessing the existential). Thus we have $\texttt{prog}_i\, \rho$, completing the proof of $\texttt{safe}_i^I\, C$. □

Observe that proving coverage by Lemma 2.7.11 requires choosing a single particular covering continuation. The statement of Lemma 2.7.9 is more general, in that the continuation which covers D is chosen in the context of some particular instantiantion of D's existential variables, and under the assumptions of D and the global invariant; in particular this allows a case analysis, that different covering continuations be chosen based on which of several disjunctive possibilities hold. The following lemma allows us to regain this; first a notation.

Definition 2.7.13. Let C be a locinv and $A : C.\texttt{type} \to \texttt{Prop}$. Then the notation $C \wedge A$ refers to the locinv where A is added to the assumptions, that is:

$$\begin{aligned}
(C \wedge A).\texttt{type} &= C.\texttt{type} \\
(C \wedge A).\texttt{regs} &= C.\texttt{regs} \\
(C \wedge A).\texttt{assume} &= \lambda \boldsymbol{x}.\, (C.\texttt{assume}\; \boldsymbol{x}) \wedge (A\; \boldsymbol{x}) \\
(C \wedge A).\texttt{progress} &= C.\texttt{progress}
\end{aligned}$$

Lemma 2.7.14. *Let* $I : \texttt{state} \to \texttt{Prop}$*. Let* D *be a locinv and* $\mathcal{E}$ *a set of locinvs; let* $A, B : D.\texttt{type} \to \texttt{Prop}$ *such that*

$$\forall \boldsymbol{x} : D.\texttt{type}.\, (D.\texttt{assume}\; \boldsymbol{x}) \wedge (I\; (D.\texttt{regs}\; \boldsymbol{x})) \implies ((A\; \boldsymbol{x}) \vee (B\; \boldsymbol{x})).$$

Then $\mathcal{E}\; \texttt{covers}^I \{D\}$ *if* $\mathcal{E}\; \texttt{covers}^I \{D \wedge A, D \wedge B\}$*.*

Proof. Suppose that $\mathcal{E}\,\texttt{covers}^I\{D \wedge A, D \wedge B\}$. To show that $\mathcal{E}\,\texttt{covers}^I\{D\}$, assume that $\texttt{safe}_i^I\,E$ for all $E \in \mathcal{E}$. Then $\texttt{safe}_i^I(D \wedge A)$ and $\texttt{safe}_i^I(D \wedge B)$, and it must be shown that $\texttt{safe}_i^I\,D$.

So choose some ρ such that $\rho \models_i^I D$. Then $(I\ \rho)$, and there is some $\boldsymbol{x} : D.\texttt{type}$ such that $(D.\texttt{assume}\ \boldsymbol{x})$, and $\rho = (D.\texttt{regs}\ \boldsymbol{x})$. Thus $((A\ \boldsymbol{x}) \vee (B\ \boldsymbol{x}))$ holds. But in the first case, $\rho \models_i^I (D \wedge A)$; in the second case $\rho \models_i^I (D \wedge B)$. Either way $\texttt{prog}_i\,\rho$, completing the proof. ☐

One more means of establishing coverage will be needed:

Lemma 2.7.15. *Let* $I : \texttt{state} \rightarrow \texttt{Prop}$*. Let* D *be a locinv such that*

$$\forall \boldsymbol{x} : D.\texttt{type}.\,(D.\texttt{assume}\ \boldsymbol{x}) \wedge (I\ (D.\texttt{regs}\ \boldsymbol{x})) \implies \texttt{False}.$$

Then for any set $\mathcal{E}$ *of locinvs,* $\mathcal{E}\,\texttt{covers}^I\{D\}$*.*

Proof. If there were any ρ such that $\rho \models_i^I D$, then $\texttt{False}$ would hold. Thus there is no such ρ and $\texttt{safe}_i^I\,D$ holds vacuously, and thus $\mathcal{E}\,\texttt{covers}^I\{D\}$ holds. ☐

Together, the means of proving `covers` given in Lemmas 2.7.11, 2.7.14 and 2.7.15 seem sufficient for the needs of extensions; in the implementation these are the only means of proving coverage that the extension can use. See Figure 2.1 for an expression of these as proof rules. This constitutes an alternate, weaker and yet sufficient definition of the notion of `covers`.

One last refinement of the notion of `covers` has proved useful. Looking ahead to the implementation, the extension has the job of producing locinvs which cover the decoder's output continuations. The common case is for the extension to use a locinv very close to that given by the decoder. The extension will make incremental changes: perhaps to introduce explicitly a new assumption which follows from the others, perhaps to forget that a register has a certain specific value and replace it with a new existential variables (which we call "freshening the register").

For example, type-based proofs of memory safety typically require that the memory satisfy a certain invariant, say $(\texttt{memOk}\ M)$; the actual contents of the memory are not so important. So the decoder might have an input locinv along the lines of

$$\lambda\rho.\ \exists M : \texttt{mem}, A : \texttt{val}, B : \texttt{val}, T : \texttt{type}.((\mathbf{r}_M\ \rho) = M) \wedge (\texttt{memOk}\ M) \wedge (\texttt{hasType}\ A\ (\texttt{ptr}\ T)) \wedge (\texttt{hasType}\ B\ T).$$

For choice of $E' \in \{\lambda_.E \mid E \in \mathcal{E}\} \cup D.\texttt{progress}$
and $f : \Pi \boldsymbol{x} : D.\texttt{type}.(E'\ \boldsymbol{x}).\texttt{type}$;
letting $E = (E'\ \boldsymbol{x})$ and $f'\ x = (x, f\ x)$:

$$\frac{\begin{array}{c}[\ (D.\texttt{assume}\ \boldsymbol{x}),\ (I\ (D.\texttt{regs}\ \boldsymbol{x}))\ ]\\ \vdots \\ (D.\texttt{regs}\ \boldsymbol{x}) = (E.\texttt{regs}\ (f\ \boldsymbol{x})) \wedge \\ (E.\texttt{assume}\ (f\ \boldsymbol{x}))\end{array} \qquad \mathcal{E}\ \texttt{covers}^I\ \texttt{Acc}\, D\ f'\ (E'.\texttt{progress})}{\mathcal{E}\ \texttt{covers}^I \{D\}}$$

$$\frac{\begin{array}{c}[\ (D.\texttt{assume}\ \boldsymbol{x}),\ (I\ (D.\texttt{regs}\ \boldsymbol{x}))\ ]\\ \vdots \\ (A\ \boldsymbol{x}) \vee (B\ \boldsymbol{x})\end{array} \qquad \mathcal{E}\ \texttt{covers}^I \{D \wedge A, D \wedge B\}}{\mathcal{E}\ \texttt{covers}^I \{D\}}$$

$$\frac{\begin{array}{c}[\ (D.\texttt{assume}\ \boldsymbol{x}),\ (I\ (D.\texttt{regs}\ \boldsymbol{x}))\ ]\\ \vdots \\ \texttt{False}\end{array}}{\mathcal{E}\ \texttt{covers}^I \{D\}}$$

$$\frac{R \in D.\texttt{progress} \qquad \forall x.(g\ (f\ x)) = x}{\mathcal{E}\ \texttt{covers}^I \{\texttt{Acc}\, D\ f\ (\lambda z.(R\ (g\ z)))\}}$$

$$\frac{\mathcal{E}\ \texttt{covers}^I \{D\} \qquad \mathcal{E}\ \texttt{covers}^I\ \mathcal{D}}{\mathcal{E}\ \texttt{covers}^I (\{D\} \cup \mathcal{D})} \qquad\qquad \frac{\mathcal{E}\ \texttt{covers}^I\ \mathcal{D} \qquad \mathcal{E} \subseteq \mathcal{E}'}{\mathcal{E}'\ \texttt{covers}^I\ \mathcal{D}}$$

Figure 2.1: Proof rules for `covers`.

If the next instruction is the (type-safe) memory write of B into address A, then the decoder's output continuation might be

$$\lambda\rho.\ \exists M : \texttt{mem}, A : \texttt{val}, B : \texttt{val}, T : \texttt{type}.((\mathbf{r}_M\ \rho) = (\texttt{upd}\, M\ A\ B)) \wedge (\texttt{memOk}\ M) \wedge (\texttt{hasType}\ A\ (\texttt{ptr}\ T)) \wedge (\texttt{hasType}\ B\ T).$$

But what the extension wants is to forget the contents of memory and establish that the new memory is still `memOk`, by means of a lemma such as

$$\forall M : \texttt{mem}, A : \texttt{val}, B : \texttt{val}, T : \texttt{type}.(\texttt{memOk}\ M) \wedge (\texttt{hasType}\ A\ (\texttt{ptr}\ T)) \wedge (\texttt{hasType}\ B\ T) \implies (\texttt{memOk}\ (\texttt{upd}\, M\ A\ B));$$

so in fact the extension would like to cover the decoder's locinv with one like

$$\lambda\rho.\ \exists M' : \texttt{mem}, A : \texttt{val}, B : \texttt{val}, T : \texttt{type}.((\mathbf{r}_M\ \rho) = M') \wedge (\texttt{memOk}\ M') \wedge (\texttt{hasType}\ A\ (\texttt{ptr}\ T)) \wedge (\texttt{hasType}\ B\ T).$$

It is easy enough to establish this coverage, but going directly by Lemma 2.7.11 requires proving more than should be necessary: for it requires proving that all the assumptions of E follow from those of D, while really we only need to prove the new assumption $(\texttt{memOk}\ M')$, where $M' = (\texttt{upd}\, M\ A\ B)$). This has important efficiency ramifications with regard to the implementation of the Open Verifier. The following shows one way to avoid inefficiency.

Definition 2.7.16. For a locinv D, let a *delta* Δ be a tuple

$$(\Delta.\texttt{type}, \Delta.\texttt{regs}, \Delta.\texttt{assume}),$$

where

$$\begin{aligned} \Delta.\texttt{type} &: \texttt{Set}; \\ \Delta.\texttt{regs} &: D.\texttt{type} \times \Delta.\texttt{type} \to \texttt{state}; \\ \Delta.\texttt{assume} &: D.\texttt{type} \times \Delta.\texttt{type} \to \texttt{Prop}. \end{aligned}$$

Let ΔD be the locinv

$$\begin{aligned} \Delta D.\texttt{type} &= D.\texttt{type} \times \Delta.\texttt{type} \\ \Delta D.\texttt{regs} &= \lambda(x,y).(\Delta.\texttt{regs}\ (x,y)) \\ \Delta D.\texttt{assume} &= \lambda(x,y).(\Delta.\texttt{assume}\ (x,y)) \wedge (D.\texttt{assume}\ x) \\ \Delta D.\texttt{progress} &= \{\lambda(x,y).(E\ x) \mid E \in D.\texttt{progress}\}. \end{aligned}$$

Such a delta only allows adding assumptions and existential variables; in practice we also need to be able to drop assumptions and existential variables on which nothing depends, but this is not difficult and I will not formalize it here.

Lemma 2.7.17. *Let* $I : \texttt{state} \to \texttt{Prop}$*. Let* D *be a locinv and* Δ *a delta of* D*. Suppose that there is a function*

$$g : D.\texttt{type} \to \Delta.\texttt{type}$$

such that for any $x : D.\texttt{type}$*, under the assumptions* $(D.\texttt{assume}\ x)$ *and* $(I\ (D.\texttt{regs}\ x))$ *it is proven that*

$$(\Delta.\texttt{regs}\ (x, g\ x)) = (D.\texttt{regs}\ x)$$

and

$$(\Delta.\texttt{assume}\ (x, g\ x)).$$

Then

$$\{\Delta D\}\ \texttt{covers}^I \{D\}.$$

Proof. I proceed by Lemma 2.7.11, using the instantiation function

$$f\ x = (x, g\ x).$$

The proof is straightforward. Worth noting is how to establish that for each E in $D.\texttt{progress}$, letting $E' = \lambda(x, y).(E\ x)$, $\{\Delta D\}\ \texttt{covers}^I \{\texttt{Acc}\ D\ f\ E'\}$; this follows by Lemma 2.7.12. ☐

2.7.6 First-Order Logic

Above I have introduced a restriction limiting the expressive power available to the extension, for the benefit of simplicity and clearness. A further such restriction is possible, as follows. I have made use of higher-order features in the notion of locinv and the proof of the soundness theorem Theorem 2.6.11. Most importantly, I have made use of a higher-order predicate `safe`. But the current definition of locinv specially encodes certain claims involving `safe`, via the progress continuations. It is possible, then, to make this the only use of higher-order features, by restricting the logic used for the variables, registers, and assumptions of locinvs to *first-order*.

The activity of the extension would then be restricted to (1) producing proofs of first-order formulae, and (2) producing coverage proofs. The production of coverage proofs, in turn, essentially amounts to producing lists of locinvs, and further first-order proofs. The structure of the coverage proofs can be seen as a proxy for higher-order logic.

This is a meta-logical consideration, in the sense that it concerns the syntactic representations of logical formulae to be passed around by the modules of the system, and the design of the proof checker.

The advantage to the trusted framework is that the proof checker can be simpler. The advantage to the extension is that proof generation can be easier, e.g. using a resolution theorem prover. In practice we have managed to restrict the logic of the assumptions of locinvs to Horn logic, which makes proof generation especially easy.

Essentially, I have encapsulated all the necessary higher-order reasoning into the soundness theorem and the soundness of the `covers` rules. Extensions need only perform first-order reasoning. Surprisingly, this has sufficed even to verify programs which manipulate function pointers directly.

I believe that these constraints will also make the job of writing extensions easier. As described in Chapter 4, most of the work of writing an extension is recognizing the kinds of first-order facts that should hold at a point in execution, and proving lemmas relating such facts. Although the machinery of coverage proofs discussed above may seem somewhat technical, in an implementation it tends to be intuitive. The common case is the "coverage by delta" where one simply specifies incremental changes to be made to the state, together with proofs that no essentially new facts are being asserted. Even the most complicated cases should seem intuitive upon reflection:

- locinvs for method calls to the `foo` method of particular classes C, D, and E, should cover a dynamic dispatch to method `foo` of an object whose dynamic type could be any of C, D, or E;
- one locinv for the start of function F, which claims that jumping to the return address is safe, together with a second locinv for the actual return site at line $n+1$, should cover the result of a function call at line n.

2.7.6.1 Types

Inductive types are useful in writing extensions, where the set of program-level types from the programs to be verified can be naturally expressed as an inductive datatype. Similarly, typing predicates may be defined inductively. Based on our experience with a prototype implementation of the Open Verifier, it seems possible that such uses of types are a convenience, which could be replaced by e.g. coding program-level types into natural numbers. This would allow a much weaker logic and a correspondingly smaller trusted code base. However, it runs up against another desideratum, the ease of extension writing. The proper balance here remains unclear, but in this thesis I will assume that extension writers have access to the full power of inductive types.

Thus, I will consider that the logic to be used by the extension is a typed first-order logic, where there are certain types provided (such as `val` for machine integers and `mem` for states of memory), and further inductive types can be created by the extension as needed.

Chapter 3

The Implementation of the Open Verifier

In this chapter I discuss the implementation of a verifier based on the logical formalism developed in the preceding chapter. That formalism is abstracted over notions of machine state and safe machine transitions, so I begin by describing a particular logical instantiation of those notions, and discussing how this relates to intended notions of safety on actual machines. Next I produce an instantiation of the *decoder* required by the formalism. Finally in Section 3.4 I describe the algorithm of the Open Verifier. After discussing various issues related to the implementation, I close the chapter with a discussion of what is required to trust a verification produced by the Open Verifier.

3.1 A Simple Machine and Safety Policy

In this section I describe a simple, generic machine in order to produce a specific instantiation of the Open Verifier framework described in Section 2.6. The programs to be verified are considered to be machine-encoded programs in SAL, the "Simple Assembly Language", the instructions of which determine the state transition relation of the machine. SAL was originally described in [30] but the version I describe here is more general. In Section 3.1.7 I outline how to ascend to the verification of programs compiled for actual machines.

3.1.1 The Machine State

The machine state consists of a certain number of *registers* and a *memory*. I introduce the type `val` for the values stored in registers and memory locations, as well as for the memory locations themselves. (In a concrete machine, `val` would typically be the type of 32-bit integers.) I will assume the existence of addition and subtraction operators on `val`; I will use zero as an object of `val` and assume that equality with zero is decidable.

I introduce the type `mem` for memory states. The type `mem` comes equipped with the operators

$$\texttt{upd} : \texttt{mem} \to \texttt{val} \to \texttt{val} \to \texttt{mem}$$
$$\texttt{sel} : \texttt{mem} \to \texttt{val} \to \texttt{val}$$

where $(\texttt{upd}\, M\ A\ V)$ is the new memory resulting from a write in memory M to address A of value V, and $(\texttt{sel}\, M\ A)$ is the value stored in address A in memory M. Essentially `mem` consists of partial mappings from values to values.

For convenience I collect the registers into a type `reg`. I assume that the machine has some specified number N of registers; then the elements of `reg` are $\mathbf{r}_1$, $\mathbf{r}_2$, ..., $\mathbf{r}_N$. It is often convenient to treat the memory as a register (with values of an unusual type); in such circumstances I will refer to it as $\mathbf{r}_M$.

An element of type `state` is determined by N values of type `val`, one for each register, plus a value of type `mem` for the memory; thus

$$\texttt{state} \cong (\texttt{reg} \to \texttt{val}) \times \texttt{mem}.$$

I will use a register r as a projection function, writing $(\mathbf{r}_i\ \rho)$ for the contents of register i in state ρ; similarly $(\mathbf{r}_M\ \rho)$ is the memory of state ρ. I will use the notation

$$\rho[r_i \mapsto v_i, r_j \mapsto v_j, \ldots]$$

to indicate the state obtained by starting with ρ and replacing the values in registers (or memory) r_i, r_j, ... with v_i, v_j, ...

3.1.2 Machine Transitions

The transition relation $\rightsquigarrow$ is determined by what *instruction* is to be executed in a given state. I introduce the type `inst` of instruction, and a function `instat` such that $\texttt{instat}\,\rho$ is the instruction that will be executed from state ρ.

I will define `instat` in terms of a specified program counter register, `pc`, by decoding the value in memory at the address stored in `pc`:

$$\texttt{instat}\,\rho = \texttt{to_inst}\,\big(\texttt{sel}\;(\mathbf{r}_M\;\rho)\;(\texttt{pc}\;\;\rho)\big),$$

where `to_inst` is the function determining the instruction encoded by a given value of type `val` (that is, a machine integer). I include an instruction `invalid` to allow for the possibility that a given `val` may not properly encode an instruction.

To describe the type of instructions, I need a type `exp` of *expressions*. These are expressions which encode values relative to the contents of the registers; we can take

$$\texttt{exp} = (\texttt{reg} \to \texttt{val}) \to \texttt{val},$$

although in practice the kinds of expressions one can make will be more limited. I will coerce elements of type `exp` to functions of type `state` $\to$ `val`, writing $(e\;\rho)$ for the value of expression e in state ρ; for example, we might have an expression "$\mathbf{r}_1 + \mathbf{r}_2$" which encodes $\lambda\rho.(\mathbf{r}_1\;\rho) + (\mathbf{r}_2\;\rho)$. Observe however that expressions have no way to refer to the memory contents; specific instructions will have to be used to access the memory, making it easier to describe memory safety.

In Figure 3.1 I give the instructions as a set of constructors for the type `inst`. The semantics of instructions are given by defining a function $n : \texttt{inst} \to \texttt{state} \to \texttt{state}$ which gives the next state after the given instruction is executed on the given state. I will comment here that the name `ijump` is mnemonic for "indirect jump", and `bnez` for "branch if not equal to zero". The (direct) jump and branch instruction are given offsets from the current `pc` to determine the jump target; for genericity I am leaving the sizes of instuctions unspecified, and use an assumed ++ operator to determine the location of the next instruction in order. Observe also that the semantics of the `invalid` instruction is given as an infinite loop, but this is not particularly important; safe programs must never execute `invalid` instructions.

In our current implementation, SAL uses integer arithmetic rather than a genuine machine arithmetic. Thus we do not correctly handle arithmetic overflow. There is nothing about our system which makes this impossible, but we have not yet done the work to incorporate it.

I can now define the (not necessarily safe) machine transitions:

$$\rho \mathrel{\tilde{\leadsto}} \rho' \iff \rho' = n\;(\texttt{instat}\,\rho)\;\rho.$$

$$
\begin{aligned}
\texttt{set} &: \texttt{reg} \to \texttt{exp} \to \texttt{inst} \\
& n\ (\texttt{set}\ r\ e)\ \rho = \rho[\mathbf{r}_{PC} \mapsto (\mathbf{r}_{PC}\ \rho) +\!+; r \mapsto e\ \rho] \\
\texttt{read} &: \texttt{reg} \to \texttt{exp} \to \texttt{inst} \\
& n\ (\texttt{read}\ r\ a)\ \rho = \rho[\mathbf{r}_{PC} \mapsto (\mathbf{r}_{PC}\ \rho) +\!+; r \mapsto \big(\texttt{sel}(\mathbf{r}_M\ \rho)\ (a\ \rho)\big)] \\
\texttt{write} &: \texttt{exp} \to \texttt{exp} \to \texttt{inst} \\
& n\ (\texttt{write}\ a\ e)\ \rho = \rho[\ \mathbf{r}_{PC} \mapsto (\mathbf{r}_{PC}\ \rho) +\!+; \\
& \qquad\qquad\qquad\qquad \mathbf{r}_M \mapsto \big(\texttt{upd}(\mathbf{r}_M\ \rho)\ (a\ \rho)\ (e\ \rho)\big)] \\
\texttt{jump} &: \texttt{val} \to \texttt{inst} \\
& n\ (\texttt{jump}\ i)\ \rho = \rho[\mathbf{r}_{PC} \mapsto (\mathbf{r}_{PC}\ \rho) + i] \\
\texttt{ijump} &: \texttt{exp} \to \texttt{inst} \\
& n\ (\texttt{ijump}\ e)\ \rho = \rho[\mathbf{r}_{PC} \mapsto (e\ \rho)] \\
\texttt{bnez} &: \texttt{exp} \to \texttt{val} \to \texttt{inst} \\
& n\ (\texttt{bnez}\ e\ i)\ \rho = \begin{cases} \rho[\mathbf{r}_{PC} \mapsto (\mathbf{r}_{PC}\ \rho) +\!+] & \text{if } (e\ \rho) = 0; \\ \rho[\mathbf{r}_{PC} \mapsto (\mathbf{r}_{PC}\ \rho) + i] & \text{if } (e\ \rho) \neq 0 \end{cases} \\
\texttt{invalid} &: \texttt{inst} \\
& n\ \texttt{invalid}\ \rho = \rho
\end{aligned}
$$

Figure 3.1: SAL instructions

3.1.3 Memory Safety

Now I will introduce a safety policy. I will use memory-safety, which is our constant example. I assume the existence of a predicate $\texttt{addr} : \texttt{val} \rightarrow \texttt{Prop}$ which holds exactly of valid memory addresses. Typically we assume that there are particular blocks of memory that are allocated for use by the program. The exact location of these blocks (and thus the definition of `addr`) will change for each execution of the program, so really we will need to use a notion of safety which is not fixed but is parametric in such execution-specific values. This is discussed further below; now, for simplicity, I assume some fixed predicate `addr`.

The safety policy is then that

- it is only safe to execute $\texttt{instat}\,\rho$ from ρ when $\texttt{addr}(\texttt{pc}\,\rho)$;
- it is never safe to execute an `invalid` instruction;
- an instruction $\texttt{read}\ r\ a$ or $\texttt{write}\ a\ e$ is safe from a state ρ only when $\texttt{addr}(a\ \rho)$.

We can finally define $\rho \rightsquigarrow \rho'$ to hold whenever both

- $\rho \tilde{\rightsquigarrow} \rho'$, and
- the instruction $\texttt{instat}\,\rho$ is safe from ρ according to the safety policy above.

3.1.4 Execution Parameters

Now consider a simple operating system for the SAL machine, which takes a SAL program and executes it. This involves loading the code (and accompanying data) of the program into memory, and setting up blocks of memory for the program to use. In principle which blocks are available, and thus which memory is to be considered `addr`, might vary from execution to execution of the program. We would like to verify the *program*, in such a way that every such execution is proven safe, rather than verifying a particular execution. I will call the various values which vary from execution to execution, but are fixed over the course of any particular execution, *execution parameters*.

It is worth noting first how the variant information about accessible memory can be made available to the program itself. Consider the example of Cool

(Section 4.1) programs running on the MIPS simulator `spim`. In this setting the accessible memory of the program, outside of its own data block, consists of a stack and a heap. The extent of the heap is given to the program in the initial values of the **$gp** and **$s7** registers, marking the first and last available address in the heap, respectively. The top of the stack is passed in the **$sp** register, and the stack is guaranteed to be "large enough" in some appropriate sense (see Section 4.3). The location of the code and data is not needed directly, as the program could refer to code and data by relative offsets from the current program counter (as indicated in Figure 3.1); in practice, *labels* are used (see Section 3.1.6).

I introduce the execution parameters as a component of the type `state`, additional to the registers and the memory:

$$\texttt{state} = (\texttt{regs} \to \texttt{val}) \times \texttt{mem} \times \texttt{paramsType}.$$

The type `paramsType` is supposed to contain all of the execution-specific information about how the operating system has set up the program for execution. For definiteness we can consider it to comprise the following eight addresses of type `val`:

`code_start`	`data_start`	`heap_start`	`stack_start`
`code_end`	`data_end`	`heap_end`	`stack_end`

I will use the notation (`params` ρ) to refer to the component of ρ of type `paramsType`.

I will take the definition of $\tilde{\leadsto}$ as before, with the observation that all transitions preserve (`params` ρ). (Recall that $\leadsto$ only models transitions during the execution of a single program; $\leadsto$ does not and need not accurately describe the return of control to the operating system or another program.)

I can now define a notion of safety relative to the execution parameters.

Definition 3.1.1. For ψ : `paramsType`, let

$$\texttt{codeaddr}_\psi\, A \iff \texttt{code_start}_\psi \le A \le \texttt{code_end}_\psi;$$

and let

$$\begin{aligned}\texttt{addr}_\psi\, A \iff & \neg(\texttt{codeaddr}_\psi\, A) \wedge \\ & (\texttt{data_start}_\psi \le A \le \texttt{data_end}_\psi \vee \\ & \texttt{heap_start}_\psi \le A \le \texttt{heap_end}_\psi \vee \\ & \texttt{stack_start}_\psi \le A \le \texttt{stack_end}_\psi).\end{aligned}$$

This division will be useful for the definition of a decoder with respect to a global invariant that the code of the program is preserved, see Section 3.3.

The safety policy is then that

- it is only safe to execute $\texttt{instat}\,\rho$ from ρ when $\texttt{addr}_{(\texttt{params}\ \rho)}(\texttt{pc}\,\rho)$ or $\texttt{codeaddr}_{(\texttt{params}\ \rho)}(\texttt{pc}\,\rho)$; and
- it is never safe to execute an `invalid` instruction;
- an instruction $\texttt{read}\ r\ a$ or $\texttt{write}\ a\ e$ is safe from a state ρ only when $\texttt{addr}_{(\texttt{params}\ \rho)}(a\ \rho)$ or $\texttt{codeaddr}_{(\texttt{params}\ \rho)}(a\ \rho)$.

We can finally define $\rho \rightsquigarrow \rho'$ to hold whenever both

- $\rho \tilde{\rightsquigarrow} \rho'$, and
- the instruction $\texttt{instat}\,\rho$ is safe from ρ according to the safety policy above.

Really, the execution parameters are only needed for the formalism. For any single particular verification, the execution parameters can be considered as fixed logical constants, and predicates like `addr` can be considered fixed particular predicates.

In our prototype implementation, the dependence of predicates like `addr` on execution parameters of the state is suppressed, so the initial locinv will contain apparently global assertions like (`addr data_start`), and the decoder will output local safety conditions of the form ($\texttt{addr}\,A$). Intuitively, then, `addr` is treated like some specific predicate, which, in the course of any single execution, it is.

To connect to the formalism, in order for the locinvs to contain assumptions about $\texttt{params}\ \rho$, we can enforce that every locinv's existential variables contain an extra parameter p of type `paramsType`, and the locinv's register state asserts that $(\texttt{params}\ \rho) = p$, and every occurence of predicates like `addr` is relativized to p. Again, in the implementation we have left all of this implicit.

3.1.5 Execution Parameters and Extension Lemmas

Since $\texttt{params}\ \rho$ is preserved by $\rightsquigarrow$, any facts that hold of the initial state of execution will hold at any state. In particular, any facts, dependent only on the

parameters, that the extension can prove from the assumptions of the initial locinv, can be considered to be part of the assumptions of any locinv. With respect to the formalism, there are two ways to go about this. One way is to have the extension explicitly carry any such facts in each locinv it produces; they will automatically be preserved by the decoder. This can be refined by having the extension define some predicate which encodes all the necessary information about the initial state, prove various lemmas about that predicate, and then carry just that one predicate in each locinv. This is done by the Cool extension (Section 4.2) using the `hierarchyOk` predicate.

It is possible, however, to use the mechanism of the global invariant to allow the extension to use execution-specific definitions and lemmas. Suppose the extension introduces certain facts which depend only on the execution parameters:

$$\texttt{Lemmas} : \texttt{paramsType} \to \texttt{Prop}.$$

Suppose that the extension is able to prove, given the initial locinv C_0, that

$$\forall i. \forall \rho_0. (\rho_0 \models_i^{I_{\mathrm{orig}}} C_0) \implies (\texttt{Lemmas}\ (\texttt{params}\ \rho_0)).$$

Then we could perform the verification using the global invariant

$$(I\ \rho) \iff (I_{\mathrm{orig}}\ \rho) \wedge (\texttt{Lemmas}\ (\texttt{params}\ \rho)).$$

This will work soundly, but there are certain complications involved in establishing it. In order to apply the soundness theorem (Theorem 2.6.11) it is necessary that the decoder be correct with respect to the new invariant, and that the initial locinv still hold with respect to the new invariant. In fact, if $\rho \models_i^{I \wedge J} C$ were equivalent to $(\rho \models_i^I C) \wedge (J\ \rho)$, these facts would follow from the fact that the new part of the invariant holds in the initial state, and that it is automatically preserved by $\rightsquigarrow$ (since the `params` component of the state is).

Unfortunately, the invariant is also used in the safety claims given by C's progress continuations, so this is not quite so easy. A closer analysis is needed of how progress continuations are used by the decoder and in the initial locinv is needed. I will return to this in Section 3.3 and Section 3.4.4.

3.1.6 Labels

Although the SAL instructions described in Figure 3.1 use direct jumps and branches to offsets from the current program counter, it is usually more con-

$$\texttt{jump} : \texttt{labelType} \rightarrow \texttt{inst}$$
$$n\ (\texttt{jump}\ L)\ \rho = \rho[\mathbf{r}_{PC} \mapsto \&L]$$
$$\texttt{bnez} : \texttt{exp} \rightarrow \texttt{labelType} \rightarrow \texttt{inst}$$
$$n\ (\texttt{bnez}\ e\ L)\ \rho = \begin{cases} \rho[\mathbf{r}_{PC} \mapsto (\mathbf{r}_{PC}\ \rho) +\!+] & \text{if } (e\ \rho) = 0; \\ \rho[\mathbf{r}_{PC} \mapsto \&L] & \text{if } (e\ \rho) \neq 0 \end{cases}$$

Figure 3.2: SAL instruction, modified to use labels for direct jumps.

venient (particularly when hand-coding assembly) to use *labels*. This can be done using the framework of execution parameters, as follows.

Assume a type `labelType` of labels (this is typically some type of strings). In the assembly program, certain lines of code are given certain labels. Then for any particular execution of the program there is an "address-of" operator & which maps labels to particular values of the program counter. In particular we can re-express the direct jump and branch instructions to use labels; see Figure 3.2. Formally the execution parameters (type `paramsType`) have to be extended to incorporate the labels, so that & is a function of `params` ρ. In the figure this is suppressed, so $\&_{(\texttt{params}\ \rho)}L$ is written simply $\&L$.

We also assume that expressions (type `exp`) can refer to the & operator, which is useful for setting the return address before a function call.

In a real system, of course, all references to the labels are replaced with offsets, so this mechanism is simply a convenience for the extension writer. Currently the implementation does verify assembly code rather than machine code, which means that the assembler would have to be considered a trusted component.

3.1.7 Verification on Actual Machines

To verify programs compiled on an actual machine, it would be possible to implement the notion of `state` and $\rightsquigarrow$ for the language of that machine, and design a decoder for that language; but for our prototype implementation, we instead translate such programs into SAL and verify the SAL translations. This is an assembly-to-assembly translation. I will not discuss the details of

translation here; SAL is simple and generic enough that it is usually not difficult to see how the translation works. Some discussion of how to translate MIPS and x86 code into a slightly different version of SAL can be found in [30].[1]

To actually believe in the safety of the original program given a verification of the SAL translation, we have to trust the translator. This means to trust that it is "safety-reflecting" in the sense that if a translation is safe (in the SAL sense), then the original program is safe (in the original sense). I believe that the translation process is easy to trust in general, though I leave it for future work to spell out precisely what is required.

The only point which seems to provide any trouble is the handling of the program counter. A single MIPS or x86 instruction will in general translate into several SAL instructions. This makes it difficult to deal with program-counter arithmetic in the SAL translation. The solution we have used is to restrict the possible values of the program counter in locinvs to expressions of the form

$$\texttt{code_start} +_{\mathrm{SAL}} n$$

for natural numbers n, where $+_{\mathrm{SAL}}$ is a special operator which is not otherwise used in the code. The SAL instructions are laid out in order at addresses `code_start` $+_{\mathrm{SAL}} 0$, `code_start` $+_{\mathrm{SAL}} 1, \ldots$. If the extension tries to use a locinv whose `pc` is not of this special form, the decoder will fail (or, at best, produce output which requires a proof of `False`).

In particular, if a program tries to do any arithmetic on the program counter, it will translate to a SAL program which will not verify. The only way to produce a new program counter is by going to the next instruction, for which the decoder uses $+_{\mathrm{SAL}}1$, or by using the & operator to refer to an instruction by its label.

This is a conservative approach; it does mean that genuinely safe original programs may translate into unverifiable SAL programs. I do not think that many sensible programs (especially those produced by sensible compilers) will be affected.

Finally observe that to handle indirect jumps, the decoder may produce locinvs with `pc` not of the special form; but the extension will have to provide

[1] It may be worth noting that the SAL register set can be larger than that of the actual machine. For instance, to handle x86 condition flags the SAL translation treats those flags as registers; each arithmetic operation translates into several SAL instructions, one to perform the arithmetic and the others to set the flags.

a coverage proof from locinvs with valid `pc` (by using progress continuations, for instance). Example are in Section 4.2.2.2 and Section 4.4.1.

3.2 Safety Policies

In this section I will make a short digression on safety policies. First I give some examples of slightly more complicated safety policies (which are still centered on memory safety) that could still be easily handled by our implementation. Then I discuss more generally what sorts of safety policies could be handled in the formalism.

3.2.1 More about the SAL Safety Policy

Above in Section 3.1.3 I have given a straightforward safety policy for a hypothetical machine using the language SAL. In Section 3.1.4, this safety policy is modified to allow for a simple operating system which might set up different memory configurations for different executions of the program.

A useful addition is to allow the operating system to provide routines which the program can call. It needs to be incorporated into the safety policy that calling the operating system routines is considered to be safe, without having to verify the code for the routines. In terms of the formalism, the transition relation $\rightsquigarrow$ can be augmented with special transitions for calls to trusted functions; functions which do not return (such as safe aborts and exits) can be modelled as automatically safe infinite loops. In terms of the verification effort, trusted functions can be modelled by progress continuations which are claimed to be safe in the initial locinv for the program. In particular, when calling a trusted function is only to be considered safe under certain conditions, this can be made part of the locinv for the trusted function, as a precondition (see Section 4.4).

One interesting example is allocation. Above, I have modelled an operating system which supplies a certain amount of accessible memory to the program at the beginning of execution, but provides no way to obtain more memory. Instead we might like to use an operating system which allows calls to a routine which may provide a new block of accessible memory. In such a scenario, `addr` can no longer considered to be static over the execution of the program, but must be explicitly made dependent on the state. A new component of the state corresponding to the state of the operating system allocator can be set

aside, much like a new register. The progress continuation for the allocator can specify that the new allocation state will maintain the accessibility of all addresses which were `addr` before the allocator was called.

3.2.2 Generalizing the Safety Policy

In the formal development, I have specified the safety policy via a transition relation $\leadsto$ on machine states. A program is considered to be safe if every possible initial state of the program can make indefinite progress according to $\leadsto$. Generally, $\leadsto$ is some subset of all the transitions which the machine can make; some of the transitions are considered unsafe, and $\leadsto$ is restricted so that unsafe transitions are forbidden.[2] For example, to handle memory-safety, $\leadsto$ only allows transitions involving a memory access when the memory concerned is part of some region considered accessible.

Although in this thesis I only consider the application of the Open Verifier to memory safety, here I briefly consider the question of what kinds of safety policies could be implemented by this formal framework. The following definitions are adapted from [38]. A *security policy* is a predicate on sets of executions. A program is said to satisfy a security policy (is safe) if the set of *all* possible executions of the program satisfies the predicate. A *security property* is a security policy that can be specified by a predicate on individual executions, where a program is safe if *each* execution individually satisfies the predicate. Finally a *safety property* is a security property which holds of an execution, only if it holds for every finite prefix of an execution. Safety properties correspond to the notion of preventing "bad things" from happening during an execution; once it happens, the execution cannot later become safe.

The class of security policies which can be handled by the Open Verifier formalism, defining safety via a $\leadsto$ consisting of the safe subset of all machine transitions $\tilde{\leadsto}$, essentially coincides with the class of safety properties over $\tilde{\leadsto}$-

[2] If the set of possibly-unsafe machine transitions $\tilde{\leadsto}$ is "functional" or "deterministic", in the sense that there is at most one transition from any state, then $\leadsto$ is simply that subset of $\tilde{\leadsto}$ consisting of safe transitions. When, however, a single state has transitions both safe and unsafe, then all those transitions must be forbidden by $\leadsto$; since the notion of safety is "never getting stuck", we have to ensure that there are no $\leadsto$-transitions at all from a state which *may* produce an unsafe transition. In such a scenario it may be more fruitful to think primarily of safe and unsafe *states*, rather than safe and unsafe *transitions*—if any unsafe transition is possible from a state, it can never be safe to be in that state.

executions. Any transition relation $\leadsto$ determines a safety property, which holds of an execution when each transition is within $\leadsto$. It is only slightly harder to translate a safety property into an appropriate transition relation. We can augment the notion of `state` to include a *history* component, a pseudo-register which contains the execution so far. Then given a safety property S, we can define $\rho \leadsto \rho'$ if and only if

$$(\rho \tilde{\leadsto} \rho') \quad \wedge \quad \forall \rho''.(\rho \tilde{\leadsto} \rho'') \implies \big(S\ (\texttt{history}\ \rho'')\big).$$

3.3 The Decoder

As established in Theorem 2.6.11, the decoder correctness property is sufficient for soundness. Observe however that a decoder which always returns

$$\lambda i.\lambda \rho.\texttt{False},\ \{\}$$

trivially satisfies the correctness property. With such a decoder we have no hope of proving the safety of a program. We need to design a decoder which better reflects the actual semantics of $\leadsto$. In fact, the intended use of the decoder within the OpenVerifier is *to replace all reasoning about* $\leadsto$, that is, *to encode all facts about machine transitions and the safety policy.* We intend that reference to $\leadsto$ need only occur in the proof of decoder correctness, the proof of special proof rules to be used by the extension to prove the `covers` proof obligations, and finally the overall proof of the soundness of the framework (Theorem 2.6.11). The untrusted extension should never have to reason about $\leadsto$; this requires that the decoder express its local safety condition and continuations without reference to $\leadsto$. (This is not a *logical* requirement but a *meta-logical* one about how we would like the decoder output to be represented.)

I will illustrate the intended decoder by producing a decoder for the language of SAL and the $\leadsto$ described in Section 3.1.4. The decoder will work with locinvs with restricted program counters as described in Section 3.1.7.

The decoder will formally be considered specific to the program being verified, though it should be clear how the implementation can provide this decoder parametrically for any SAL program. Suppose that the instructions of the program are numbered 0, 1, ..., N; let the instructions be i_0, i_1, ..., i_N. Then

the decoder will be correct with respect to the global invariant

$$\begin{aligned} \lambda\rho.&\texttt{to_inst}\ (\texttt{sel}\ (\mathbf{r}_M\ \rho)\ (\texttt{code_start} + 0)) = i_0\ \wedge \\ &\texttt{to_inst}\ (\texttt{sel}\ (\mathbf{r}_M\ \rho)\ (\texttt{code_start} + 1)) = i_1\ \wedge \\ &\cdots \wedge \\ &\texttt{to_inst}\ (\texttt{sel}\ (\mathbf{r}_M\ \rho)\ (\texttt{code_start} + N)) = i_N. \end{aligned}$$

Call this global invariant I. Assume additionally that $\texttt{code_end} = \texttt{code_start} + N$.

The decoder will work with τ-augmented locinvs where for a locinv C, $\tau\ C$ is the type

$$\Sigma n : \texttt{nat}.(0 \leq n \leq N) \wedge \forall \boldsymbol{x} : C.\texttt{type}.\big((\texttt{pc}\ (C.\texttt{regs}\ \boldsymbol{x})) = \texttt{code_start} + n\big).$$

That is, a τ-augmented locinv is a locinv which specifies a particular program counter which is (provably) inside the code block. Taking such a locinv as input, the decoder can determine, via the global invariant, what instruction is going to be executed; and then it can tailor its output based on the instruction.

In the implementation, the extension doesn't produce the augmentation explicitly. Instead, it produces simple locinvs which are syntactically checked by the trusted framework. Locinvs for which the program counter of the locinv is syntactically identical to $\texttt{code_start} + n$ for some n are called *direct locinvs*; other locinvs are called *indirect locinvs*. Only direct locinvs are accepted by the trusted framework.

The terminology is motivated by direct and indirect jumps; by inspection of the decoder definition given below, it should be clear that the decoder only produces direct locinvs,[3] except in the case of an indirect jump. When the extension covers the decoder output after an indirect jump, it can only use direct locinvs. This can be done either with progress continuations, or by a case analysis over the possible explicit program counters which might be the target of the indirect jump. For further discussion see Section 4.4.

In Figure 3.3 the decoder is defined. Recall that a (τ-augmented) decoder takes as input a τ-augmented locinv, and returns a pair, consisting of an state predicate (the local safety condition) and a list of locinvs (the possible next states). The figure shows, given the instruction to be executed, the local safety

[3]Program counters of the form &L for a label L can be automatically translated to the direct form.

Instruction	P	$\mathcal{D}$
`set` $r\ e$		$\{C$ with $\texttt{regs} = \lambda \boldsymbol{x}.\rho[\texttt{pc} \mapsto (\texttt{pc}\,\rho +_{\text{SAL}} 1);$ $r \mapsto e\ \rho]\}$
`read` $r\ a$	$\lambda\rho.\,\texttt{addr}(a\ \rho)$	$\{C$ with $\texttt{regs} = \lambda \boldsymbol{x}.\rho[\texttt{pc} \mapsto (\texttt{pc}\,\rho +_{\text{SAL}} 1);$ $r \mapsto \big(\texttt{sel}(\mathbf{r}_M\ \rho)\ (a\ \rho)\big)]\}$
`write` $a\ e$	$\lambda\rho.\,\texttt{addr}(a\ \rho)$	$\{C$ with $\texttt{regs} = \lambda \boldsymbol{x}.\rho[\texttt{pc} \mapsto (\texttt{pc}\,\rho +_{\text{SAL}} 1);$ $\mathbf{r}_M \mapsto \big(\texttt{upd}(\mathbf{r}_M\ \rho)\ (a\ \rho)\ (e\ \rho)\big)]\}$
`jump` L		$\{C$ with $\texttt{regs} = \lambda \boldsymbol{x}.\rho[\texttt{pc} \mapsto \&L]\}$
`ijump` e		$\{C$ with $\texttt{regs} = \lambda \boldsymbol{x}.\rho[\texttt{pc} \mapsto (e\ \rho)]\}$
`bnez` $e\ L$		$\Big\{C$ with $\texttt{regs} = \lambda \boldsymbol{x}.\rho[\texttt{pc} \mapsto (\texttt{pc}\,\rho +_{\text{SAL}} 1)],$ $\texttt{assume} = \lambda \boldsymbol{x}.(e\ \rho) = 0 \wedge (C.\texttt{assume}\ \boldsymbol{x}),$ C with $\texttt{regs} = \lambda \boldsymbol{x}.\rho[\texttt{pc} \mapsto \&L],$ $\texttt{assume} = \lambda \boldsymbol{x}.(e\ \rho) \neq 0 \wedge (C.\texttt{assume}\ \boldsymbol{x})\Big\}$
`invalid`	$\lambda\rho.\texttt{False}$	$\{\ \}$

Figure 3.3: The decoder. When the P column is left blank the local safety condition is `True`. In the $\mathcal{D}$ column, "ρ" is intended as shorthand for $(C.\texttt{regs}\ \boldsymbol{x})$.

condition P and the output continuations $\mathcal{D}$ in terms of the input locinv C. This decoder uses the SAL instructions modified to use labels for direct jumps, as in Figure 3.2.

When the P column is left blank the local safety condition is `True`. In the $\mathcal{D}$ column, "ρ" is intended as shorthand for $(C.\texttt{regs}\ \boldsymbol{x})$. Finally, the notation "C with `regs`=..." is intended to represent the locinv which is obtained by taking C and replacing its `regs` field with the given value. Observe that in no case does the decoder change the `type` or `progress` fields of the input locinv.

Recall the definition

Definition 3.3.1. A τ-augmented decoder `decode` satisfies the *decoder correctness property with invariant I* iff, for any τ-augmented locinv $(C, \boldsymbol{t})$, where $(P, \mathcal{D}) = \texttt{decode}(C, \boldsymbol{t})$,

$$\forall i.\forall \rho.(\rho \models^I_{i+1} C) \wedge (P\ \rho) \implies (\exists \rho'.\rho \rightsquigarrow \rho') \wedge \Big(\forall \rho'.\rho \rightsquigarrow \rho' \implies \texttt{prog}\,\rho' \vee \big(\bigvee_{D\in\mathcal{D}} \rho' \models^I_i D\big)\Big).$$

That is, for any state ρ satisfying the input locinv C (and the global invariant I), if the local safety condition P holds, then progress is possible for at least one step, and the resulting state is either safe (can make indefinite safe progress), or will satisfy some $D \in \mathcal{D}$ (and I), at an index one less than that by which the original state satisfied C.

Then we have

Theorem 3.3.2. *The decoder* `decode` *defined in Figure 3.3 is correct with respect to the global invariant* I*.*

Proof. The proof is a straightforward case analysis over the instruction to be executed, referring to the notion of $\rightsquigarrow$ derived from Figures 3.1 and 3.2. The witness of the existential variables for the satisfaction of the output continuations in $\mathcal{D}$ is the same as the witness for the input continuation C.

One point worth noting is that the output continuations will actually be satisfied at index $i+1$ rather than i (the "strong decoder correctness" described in Section 2.7.4). As mentioned there, the monotonicity of $\models$ ensures satisfaction at i as well.

The preservation of the global invariant I is handled by the fact that `addr` specifically excludes all the addresses in the code block, so the code is preserved by all memory updates allowed by the decoder. $\square$

Finally, recall from Section 3.1.5 that we might like to expand the global invariant with facts, provided by the extension, which are guaranteed to be preserved by $\rightsquigarrow$ because they depend only the the execution-specific parameters. It is not hard to see that `decode` is correct with respect to any invariant $I \wedge J$ where J is automatically preserved by $\rightsquigarrow$. This depends on the fact that the decoder's output locinvs have the same progress continuations as the input locinvs; because of that, whenever

$$\rho \models_i^I C,\ \rho \rightsquigarrow \rho',\ \text{and}\ \rho' \models_i^I D,$$

it also holds that

$$\rho \models_i^{I \wedge J} C \implies \rho' \models_i^{I \wedge J} D.$$

3.4 The Algorithm

Having defined the logical notions of `state`, $\rightsquigarrow$, and a correct decoder, it is now possible to put together an algorithm for program verification. Recall the soundness theorem

Theorem 3.4.1 (Soundness of the Open Verifier). *Let I : `state` $\rightarrow$ `Prop`, and `decode` be a τ-augmented decoder which satisfies the decoder correctness property with invariant I. Suppose a set $\mathcal{E}$ of τ-augmented locinvs is closed under scanning with respect to `decode` and I.*

Suppose also that $\mathcal{E}$ $\texttt{covers}^I\{C_0\}$, where C_0 is an initial locinv for a program in the sense that for any possible initial state ρ_0, $\rho_0 \models_i^I C_0$ for all i. Then the program is safe in the sense that each such ρ_0 can make indefinite safe progress.

The role of the untrusted extension is to produce the set $\mathcal{E}$ of locinvs, and the necessary proofs:

- that $\mathcal{E}$ $\texttt{covers}^I\{C_0\}$;
- that $\mathcal{E}$ is *closed under scanning*; that is, for each $E \in \mathcal{E}$, letting $(P, \mathcal{D}) =$ `decode` E, it is the case that
  - $\forall i.\forall \rho.(\rho \models_i^I E) \implies (P\ \rho)$;
  - and $\mathcal{E}$ $\texttt{covers}^I\,\mathcal{D}$.

As will be seen, part of the production of $\mathcal{E}$ is handled by the trusted framework.

3.4.1 Trusted Components

The trusted Open Verifier framework can be divided into four components:

1. the initializer;
2. the decoder;
3. the proof checker; and
4. the director.

The *initializer* is responsible for producing the initial locinv C_0 for the program to be verified. The *decoder* is simply a wrapper for the logical decoder defined in Section 3.3. The *proof checker* checks the extension's proofs. Finally, the *director* is responsible for working with the extension and the decoder to produce the set $\mathcal{E}$ and ensure that it is closed under scanning.

3.4.2 The Director

The heart of the Open Verifier algorithm is a dialogue between the trusted decoder and an untrusted extension, as shown in Figure 3.4. The decoder, given a direct locinv, describes the conditions under which that locinv is safe (the local safety condition P) and will create the locinvs describing the state after further execution (the locinvs $\mathcal{D}$). The extension is responsible for knowing why the program is safe, presumably based on its domain-specific knowledge, e.g., the source language and the compilation strategy. For every locinv that is considered, the extension must, first, prove the local safety condition, and second, prove that each of the decoder's local invariants are safe. This latter obligation is met by providing new locinvs $\mathcal{E}$ over which the whole process is repeated and by proving that the safety of its locinvs implies the safety of the decoder's, as $\mathcal{E}\ \texttt{covers}^I\ \mathcal{D}$.

The director is responsible for coordinating the dialogue between the decoder and the extension, ensuring that all facts are checked and that all program paths are explored, as shown below:

$$
\begin{array}{l}
\textbf{director}\ \textit{Scanned}\ [\,] = \textbf{success} \\
\textbf{director}\ \textit{Scanned}\ (C{::}\textit{ToScan}) = \\
\quad \texttt{let}\ (P, \mathcal{D}) = \textbf{decoder}\ C\ \ \texttt{in} \\
\quad \texttt{let}\ (pf_P, \mathcal{E}, pf_{\texttt{covers}}) = \textbf{extension.scan}\ C\ \ \texttt{in} \\
\quad \texttt{if}\ \textbf{proofok}\ pf_P\ \big(\forall \boldsymbol{x} : C.\texttt{type}.\,(C.\texttt{assume}\ \boldsymbol{x}) \wedge \texttt{Lemmas} \implies \\
\qquad\qquad\qquad\qquad\qquad\qquad (P\ (C.\texttt{regs}\ \boldsymbol{x}))\big)\ \texttt{and} \\
\qquad \textbf{proofok}\ pf_{\texttt{covers}}\ (\mathcal{E}\ \texttt{covers}^{I \wedge \texttt{Lemmas}}\ \mathcal{D}) \\
\quad \texttt{then}\ \ \textbf{director}\ (C{::}\textit{Scanned})\ \big((\mathcal{E} \setminus (C{::}\textit{Scanned})) \cup \textit{ToScan}\big) \\
\quad \texttt{else}\ \ \textbf{failure}
\end{array}
$$

The director maintains two lists of *direct* locinvs, called *Scanned* and *ToScan*. Initially, *Scanned* is empty and *ToScan* contains a list of locinvs which the extension can prove cover the initial locinv (see below). The verification succeeds

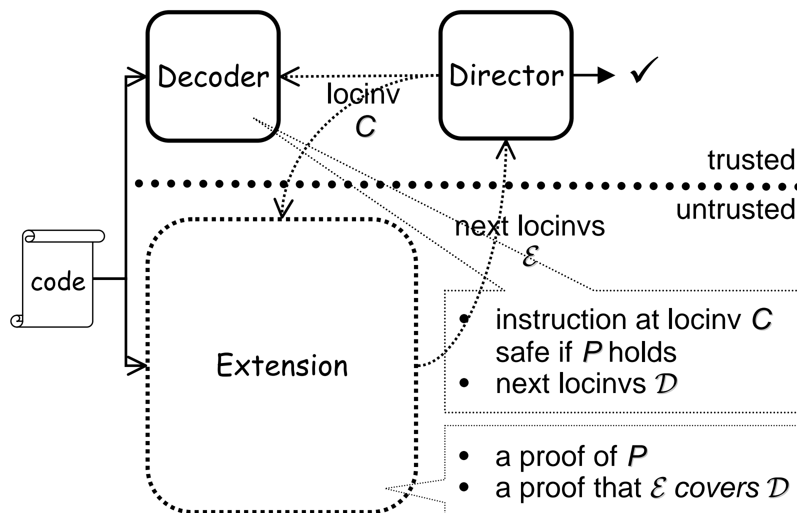

Figure 3.4: The Open Verifier architecture.

if the director achieves a state where *ToScan* is empty. The process of "scanning" a locinv involves querying the decoder and the extension, checking the relevant proofs of the local safety condition and coverage, and adding any new locinvs given by the extension to the *ToScan* list. Observe that the extension may wish to prove coverage using a previously scanned locinv (for instance, when scanning a loop where the locinv encodes the loop invariant); in this case, the locinv will not be scanned again (as indicated by the removal of all previously scanned locinvs at the recursive call). In the implementation, rather than actually comparing new locinvs with all previously scanned ones, this is handled by allowing the extension to name locinvs, and refer to previously scanned locinvs by name.

3.4.2.1 The Open Verifier Algorithm

I need to explain the use of `Lemmas` in the above algorithm for the director. As discussed in Section 3.1.5, we will allow the extension to provide proofs of lemmas which can be used to establish the necessary proofs. Formally these `Lemmas` are dependent on the execution parameters but for the implementation this dependence is suppressed and they are treated simply as logical facts.

For the purposes of the algorithm, the director can be understood as taking an extra parameter `Lemmas`, formally of type `paramsType` $\rightarrow$ `Prop`; and the global invariant for the proofs is "$I \wedge \texttt{Lemmas}$", or, more precisely,

$$\lambda\rho.(I\ \rho) \wedge \texttt{Lemmas}_{(\texttt{params}\ \rho)}.$$

In the proofs of local safety conditions, `Lemmas` should be understood as $(\texttt{Lemmas}\ (\texttt{params}\ (C.\texttt{regs}\ \boldsymbol{x})))$. This has the correct formal effect of instantiating the lemmas with the identical value of `paramsType` as is used to instantiate any use of `addr` in the local safety condition.

This is all formal detail; intuitively the lemmas can pretty safely be understood as global logical facts.

The complete Open Verifier algorithm is

$$\begin{array}{l}
\texttt{let}\ C_0 = \textbf{initializer}()\ \texttt{in} \\
\texttt{let}\ (\texttt{Lemmas}, pf_{\texttt{Lemmas}}, \mathcal{E}, pf_{\texttt{covers}}) = \textbf{extension.init}\ C_0\ \texttt{in} \\
\texttt{if}\ \ \textbf{proofok}\ pf_{\texttt{Lemmas}}\ \big(\forall \boldsymbol{x} : C_0.\texttt{type}.\ (C_0.\texttt{assume}\ \boldsymbol{x}) \implies \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{Lemmas}_{(\texttt{params}\ (C.\texttt{regs}\ \boldsymbol{x}))}\big) \\
\qquad \textbf{proofok}\ pf_{\texttt{covers}}\ (\mathcal{E}\ \texttt{covers}^{(I \wedge \texttt{Lemmas})}\{C_0\}) \\
\texttt{then}\ \ \textbf{director}\ \{\ \}\ \mathcal{E} \\
\texttt{else}\ \ \textbf{failure}
\end{array}$$

The requirements on the extension are that it provide the two methods:

$$\begin{array}{l}
\textbf{init} : \texttt{locinv} \rightarrow \\
\qquad\qquad (\texttt{paramsType} \rightarrow \texttt{Prop}) \times \texttt{proof} \times \texttt{locinv list} \times \texttt{coverproof}
\end{array}$$

and

$$\textbf{scan} : \texttt{locinv} \rightarrow \texttt{proof} \times \texttt{locinc list} \times \texttt{coverproof}.$$

For the verification to succeed, the output locinvs must always be *direct* locinvs, and the proofs must be correct. For `proof` and `coverproof` see Section 3.4.3.

The soundness of this algorithm—that a program successfully verified by it is, in fact, safe in the necessary sense—follows from Theorem 2.6.11. The correctness of the decoder has already been discussed (Section 3.3). The initializer must produce an initial locinv C_0 which is satisfied by every initial state with global invariant $I \wedge \texttt{Lemmas}$—see Section 3.4.4.

The contents of *Scanned* upon the successful termination of the **director** algorithm form the set of locinvs which is closed under scanning. To see this, take any $E \in \mathit{Scanned}$. At some point in the operation of the director algorithm, E must have been at the top of the *ToScan* list (the C in the written algorithm). Let $(P, \mathcal{D}) = \mathtt{decode}\, E$. When E was scanned, the extension had to provide a proof (pf_P) of

$$\forall \boldsymbol{x} : C.\mathtt{type}.(C.\mathtt{assume}\ \boldsymbol{x}) \wedge \mathtt{Lemmas} \implies (P\ (C.\mathtt{regs}\ \boldsymbol{x}));$$

this implies

$$\forall i.\forall \rho.(\rho \models_i^{(I \wedge \mathtt{Lemmas})} E) \implies (P\ \rho).$$

Similarly the decoder must provide a set $\mathcal{E}$ of locinvs and a coverage proof $pf_{\mathtt{covers}}$ of $\mathcal{E}\ \mathtt{covers}^{I \wedge \mathtt{Lemmas}}\ \mathcal{D}$. But clearly $\mathcal{E} \subseteq \mathit{Scanned}$, so $\mathit{Scanned}\, \mathtt{covers}^{I \wedge \mathtt{Lemmas}}\ \mathcal{D}$ by Lemma 2.7.8. This establishes the *Scanned* is closed under scanning in the sense needed to use Theorem 2.6.11.

For the termination of the algorithm, see Section 3.5.1.

3.4.3 The Proof Checker

The *proof checker* is the component of the trusted infrastructure which checks the proofs provided by the extension.

In the implementation of the algorithm, the various components of the system are not embedded *within* the logic, but rather act upon *syntactic representations* of logical entities (terms and types). In particular, the untrusted extension takes as input (a syntactic representation of) a proof obligation, and produces (a syntactic representation of) a proof. The proof checker takes the proof obligation and its purported proof (that is, their syntactic representations) and checks that the proof indeed proves what it is supposed to. Implicitly then, part of what we must trust about the *trusted* components of the system, is that their encoding and decoding of these syntactic representations is faithful to the actual logical manipulations the trusted components are supposed to perform.

I use the types `proof` and `coverproof` to refer to the proofs to be produced by the extension. The type `proof` refers to proofs of ordinary logical facts, encoded perhaps as natural deductions or as terms in Coq. The type `coverproof` is used for proofs of locinv coverage; the notion of correct

`coverproof` can be understood as defined by the proof rules in Figure 2.1. Observe that `coverproof` depends on `proof`. Observe also that no `proof` provided by the extension needs to include explicit reasoning about `state`, $\rightsquigarrow$, `locinv`, $\models$, `safe`, or `covers`; any such reasoning is broken down, by the structure of the implementation and the mechanism of `coverproof`, into proofs of simple logical facts.

Although in this thesis and in the prototype implementation I have made free use of the full logic of the Coq system, as I discuss in Section 2.7.6, the proof checker need actually only check a tiny fragment of Coq logic, corresponding to some particular typed first-order logic.

3.4.4 The Initializer

The fact that the verification actually verifies the program in question depends formally on two things: first, that the global invariant I refers to the instructions of the program (as discussed in Section 3.3); and second, that the initial locinv C_0 correctly reflects the possible initial states of execution of the program. The *initializer* is the component responsible for setting up the initial locinv.

The initial locinv reflects any guarantees that can be made about the initial values of the registers and the memory. Consider an example system in which a program is guaranteed to receive the lowest and highest available heap addresses in registers **$gp** and **$s7**, with the initial stack point at the highest stack address, where furthermore a whole megabyte of stack is guaranteed to be available. The initial locinv must include these claims, and also the fact that the data segment of the program has been loaded into memory. Formally the

initial locinv might then have the form

$$
\begin{aligned}
C_0.\texttt{type} &= \texttt{state};\\
C_0.\texttt{regs} &= \lambda \boldsymbol{x}.\, \boldsymbol{x}[\mathbf{\$pc} \mapsto \&\texttt{main};\\
&\qquad\quad \mathbf{\$gp} \mapsto \texttt{heap_start};\\
&\qquad\quad \mathbf{\$s7} \mapsto \texttt{heap_end};\\
&\qquad\quad \mathbf{\$sp} \mapsto \texttt{stack_end}]\\
C_0.\texttt{assume} &= \lambda \boldsymbol{x}.(\texttt{stack_end} - \texttt{stack_start}) \geq (\texttt{srl}\ 1\ 20) \wedge\\
&\qquad (\texttt{sel}\ (\mathbf{r}_M\ \boldsymbol{x})\ \texttt{data_start}) = \cdots \wedge\\
&\qquad (\texttt{sel}\ (\mathbf{r}_M\ \boldsymbol{x})\ (\texttt{data_start} + 1)) = \cdots \wedge \ldots\\
C_0.\texttt{progress} &= \{\lambda \boldsymbol{x}.\texttt{abort}\}
\end{aligned}
$$

where `abort` is the locinv specifying any jump to &`_abort`. The extension can establish where `addr` holds by means of its definition (Definition 3.1.1). The extension also needs some way to relate labels in the assembly program to actual locations in the data in memory; for example it needs access to such facts as &`first_data_label` = `data_start`. These are also included in the assumptions of the initial locinv.

It should be clear that such an initial locinv C_0 has the desired property that $\forall i.\ \rho_0 \models_i^I C_0$, for any initial state ρ_0. Recall from Section 3.1.5, that in order to allow the extension to add its `Lemmas` to the global invariant, the initial states must satisfy the initial locinv with the global invariant $I \wedge \texttt{Lemmas}$. We must ensure that this will hold for any `Lemmas` the extension might supply, as long as

$$\forall i. \forall \rho_0. (\rho_0 \models_i^I C_0) \implies (\texttt{Lemmas}\ (\texttt{params}\ \rho_0)).$$

As mentioned before the difficulty lies in the progress continuations of the initial locinv. As described above, where the only progress continuation reflects a call to a safe `abort`, it is clear that the call is safe for *any* global invariant. This is true for any progress continuation that doesn't specify some return to the program.

A more complicated situation might occur in the initial locinv, if we used it to contain safety claims about trusted run-time functions (as discussed in Section 3.2.1). For instance, C_0 could claim that it is safe to jump to &*OSFunction* as long as it is safe to continue execution from the current return address

(perhaps also with certain extra facts corresponding to the postcondition of *OSFunction*). Since the `Lemmas`, being dependent only on the execution parameters, are automatically preserved by $\rightsquigarrow$, it should be easy to argue that the safety claim is still valid with the stronger global invariant; but I will not discuss this more complicated case further in this thesis.

3.5 Further Implementation Issues

In this section I will more directly consider certain issues arising from the implementation of the logical formalism, rather than that formalism itself.

3.5.1 Termination

We want the verification algorithm of Section 3.4.2.1 to always terminate with either success or failure. This requires that the director and the extension always terminate.

To ensure the termination of the director (relative to that of the extension), we can have the director keep a list of the program counters of all the direct locinvs that it scans. Then it can terminate with **failure** after seeing a program counter some fixed number of times. Since there are only finitely many program counters in the program, this guarantees that the director will only scan a finite number of locinvs.

The reason to allow a program counter to be scanned more than once, is that the extension might traverse a loop several times before finding the correct loop invariant (see Section 4.2). If necessary, the system could be expanded so that director could even query the extension, for some finite limit on how many times it expects to scan a particular program counter.

Ensuring the termination of calls to the extension is trickier. The extension is provided as executable code. It would be impractical to require the extension to prove its own termination. We could require that the extension be written in some language (or logic) which guarantees termination. In practice, we think that it will suffice to introduce a simple timeout mechanism. This will allow the extension writer to use any language to produce the extension, perhaps including techniques or heuristics for which it would be difficult to guarantee termination otherwise.

3.5.2 Memory Safety of the Extension

Since the extension is provided as executable code, we need to ensure that the extension itself is safe. In particular, a guarantee of its memory safety and lack of side-effects will be needed before verifications using it can be trusted; otherwise, it could "cheat" by interfering with the trusted framework's memory or with the operating system.

To prevent this, we could run the extension in a sandbox enforced either by hardware mechanism or by software-fault isolation [41], and it should not be allowed to access system calls. Alternatively, we could require that the extension be written in some type-safe language or using some mechanism like CCured [34], and compiled with a PCC-enabled compiler so that a trustworthy proof of the extension's safety is available.

Interestingly, we can use the extension to verify itself with the Open Verifier. Only while this is done we must run the extension in a sandbox. This latter condition ensures that the extension is actually memory-safe in the run in which it is used to "prove" its own memory safety for *all* runs. After this step we can safely run the extension in the same address space as the rest of the Open Verifier. This provides a way to bootstrap the process so that we incur the cost of the sandbox only once during a configuration phase.

3.5.3 Annotations

We want to allow the extension to require certain extra information about the programs it verifies. Perhaps the extension is geared to work with a certifying compiler which can supply such information. For example, it may be that the extension cannot, or can only with difficulty, determine the intended types of function arguments from the compiled code; but that this information is immediately available from the source code. In such a situation the extension might only work with a compiler which supplies information about the types of functions.

We call such additional information *annotations*. It is not terribly important the mechanism by which the annotations are made available to the extension; it could be sent in a separate file, for instance. In our prototype implementation, the annotations are sent as comments in the assembly code of the program.

In the extreme case, one could write an extension which does no more than follow a recipe for verification provided as annotations, with each proof to be

given being supplied. This extension makes the Open Verifier an instance of proof-carrying code.

3.6 What Do We Trust?

Certain aspects of this question are common to any verifier or PCC implementation. In particular, it is always necessary to trust that one's abstraction is sufficient to correctly represent the semantics of the machine and safety policy; that the program to be verified is correctly translated into the abstraction; and that the proof checker is correct. Of course, how difficult it is to trust these things will depend on the particulars of the abstraction and the logic.

In terms of the Open Verifier, it is necessary to trust:

- that the notions of `state` and $\rightsquigarrow$ correctly represent the intended machine semantics and safety policy;
- that the initializer produces an initial locinv which correctly represents the possible initial states of the program to be verified; and
- that the proof checker is correect.

Since the Open Verifier does not produce a single verification proof and then check it, but rather breaks down the verification into small steps provided by the extension, we also need to trust those trusted components:

- that the decoder, as an implemented piece of executable code, correctly performs the functions of a correct decoder in the sense of the logical formalism; and
- that the director correctly manages all of the proofs provided by the decoder in such a way that they could all be put together into a complete verification.

In particular, we need to trust the way in which these components work with syntactic representations of complex logical notions such as locinvs; that they correctly reflect the actual logical facts. Also,

- in order not to trust the extension, we need to trust the mechanism which establishes the memory safety of the extension, or prevents a lack of safety from affecting the verification.

Finally, there are some issues which arise as a result of the particular way in which we have implemented the Open Verifier:

- Since we verify SAL programs rather than programs in actual machine languages, we need to trust the SAL translator; in particular, that any real program which translates into a safe SAL program is, in fact, safe.
- Since we verify assembly code rather than machine code, we need to trust the assembler/disassembler.

These issues are discussed further in Chapter 5.

Chapter 4

Extensions

In this chapter I discuss extensions for the Open Verifier. The main focus of this thesis is on the structure of the Open Verifier, not the building of any particular extension. However, it is important to demonstrate that extensions can be written with reasonable effort which apply to interesting programs. I do this by example, by developing an extension for the object-oriented language Cool. I begin by introducing the language and its type system, in particular the low-level type system which can be used to describe the assembly code produced by the `coolc` compiler. Then I describe the process by which we can produce an extension which verifies such code, starting with a "conventional verifier" which works much like a bytecode verifier, and building up in layers something which can be checked by the Open Verifier framework.

The discussion of Cool, the type system of compiled Cool code, and the Cool extension is not intended to be complete in detail. Readers may be reassured to remember that, from the standpoint of soundness, it does not matter whether the Cool extension reflects an accurate and complete understanding of the Cool type system and its compiler. Because of the Open Verifier framework, any programs we are able to verify with the Cool extension are memory-safe. It is provided as executable code and can use whatever heuristics it likes; we do not need to verify anything about the extension itself.

Finally I include a discussion of handling of certain software conventions, the stack and function calls, which can be used in a modular way by other extensions as well.

4.1 Cool

In this section I discuss the structure of compiled programs in Cool, the Classroom Object-Oriented Language [3]. Cool is generally a subset of Java (with one interesting exception being the `SELF_TYPE` construct). Cool is used to teach one-semester compiler classes at the University of California, Berkeley. We chose Cool as the initial demonstration language for our experiments because it is relatively simple, and we already have compilers and test programs available. Furthermore, we eventually intend to encourage the students who develop compilers for Cool to use the code verifier for quickly discovering bugs in their compilers, a strategy that has proved very effective in larger compiler projects [11].

4.1.1 Programs in Cool

A Cool program consists of a list of class definitions, each specifying:

- the name of the class;
- what class it inherits from (defaulting to the built-in class `Object`);
- the *attributes* of the class, specified by declaring their types, and optionally giving an expression with which the attribute is initialized in every new object of the class; and
- the *methods* of the class, specified by declaring the names and types of the formal parameters, the return type, and the code to be executed on method dispatch.

The Cool program begins execution by creating a new object of class `Main` and dispatching to its method `main`; these definitions must be supplied by the programmer.

A class specifies a list of *attributes* and a list of *methods*. Attributes, the dynamic data associated with an object of a class, always have local scope; methods always have global scope. Method dispatch has a pass-by-value semantics.

The Cool language is type-safe, with classes and types coinciding. All values are members of some class. Cool supports single inheritance, with the class

hierarchy rooted at the built-in class `Object`; subtyping is given by the inheritance hierarchy. A `case` expression allows branching on the dynamic type of an object.

In addition to `Object` Cool supplies four other built-in classes: `IO` (which supplies methods useful for input and output), and the three "basic classes" `Int`, `Bool`, and `String`. Except for objects of the basic classes, any uninitialized object has the special value `void`; an `isvoid` operator is provided to test for this value. The three basic classes have special default values which correspond to 0, false, and the empty string, respectively; these values are not primitive but objects like any other. An equality operator can test whether objects of basic classes have the same value; on all other classes equality is pointer-equality. User-defined classes may not inherit from the basic classes. The `new` operator takes the name of a class and produces a new initialized object of that class.

A special identifier `self` allows the programmer to reference the object on which the current method is operating. A special type `SELF_TYPE` allows the programmer to reference the dynamic type of the `self` object. The use of `SELF_TYPE` is restricted to the return types of methods, the declared types of attributes, and the expression `new SELF_TYPE`.[1]

The Cool language is very similar to Java, and I will not go into further detail here, except as needed to provide examples of verifications. For a detailed description of Cool's syntax, type-checking and operational semantics, see [3].

4.1.2 Compiling Cool

In order to design an extension to verify compiled Cool, it is necessary to study the structure of the compiled code. The standard Cool compiler, `coolc`, produces MIPS code, to be run on the MIPS emulator `spim` [21]. Run-time functions and methods are in a hand-coded MIPS assembly file `trap.handler`. These include functions to be used in compiled code, such as the equality tests on basic classes, and run-time errors such as dispatching or case analysis on `void`; and also the methods of the built-in classes, notably `Object.copy` which is also used to allocate new objects. The implementation of `Object.copy` includes calls to a generational garbage collector.

The layout of a Cool object in memory is as follows; the offsets are in bytes.

[1] `SELF_TYPE` may also be used in `let...in` expressions, which do not add any particular challenge to the verification process, and will not be discussed further here.

offset -4	Garbage collector eyecatcher
offset 0	Class tag
offset 4	Object size (in 32-bit words)
offset 8	Dispatch pointer
offset 12...	Attributes

The garbage collector "eyecatcher" always has the value -1. The class tag is an integer shared by all objects of the same dynamic type, used most importantly by the `case` statements to perform branching on the dynamic type of an object. The object size, which does not include the eyecatcher, is used only by the allocator `Object.copy`. The dispatch pointer holds the location of a table with pointers to each of the object's methods. This dispatch table is simply a list of the code locations for each method, the first at offset 0, the second at offset 4, etc. Dynamic dispatch is handled by ensuring that all ancestors of a class `C` have the methods originally defined by `C` at the same offsets. So for example, if `C` has method `foo` at offset 12, then any subclass `D` will have its method `foo` at offset 12—either the original `C.foo` if the method was not overridden, or the new `D.foo` otherwise.

For each class `C` including the built-in classes, the compiled program will include in its static data an object `C_protObj` and some code `C_init`. The object `C_protObj` is a skeleton object of class `C` with the correct class tag, object size, and dispatch pointer (to the dispatch table stored at the label `C_dispTab`). The code `C_init` is a function which takes an object of class `C`, whose location is given by the contents of the **$a0** register on function entry, and performs the attribute initializations defined by `C` and all of `C`'s parent classes.

The `new C` operation is compiled by calling the run-time `Object.copy` method on `C_protObj`, and then calling `C_init`. Finally, `new SELF_TYPE` is handled by looking up the correct `protObj` and `init` in a table indexed by the class tag of the current `self`. This table is stored at label `class_objTab`, and stores a pointer to `C_protObj` at offset 8 times the class tag of class `C`, and a pointer to `C_init` at offset 4 more than that.

More detail on Cool's run-time functions and methods, and how they interact with the compiled code, can be found in [2].

4.1.3 Verifying Cool

Now I will look at how to verify the memory-safety of compiled Cool programs. To start I will discuss an approach to writing a verifier independently of the Open Verifier framework; this "conventional verifier" will then be adapted to the framework by adding the appropriate elements.

We present the behavior of our conventional Cool verifier by example, using the following fragment of a Cool program[2]:

```
class S {                                  // A "sequence" class
  S next() { ... }                         // Iterator method
}
class R extends S {
  S next() { ... }

  void scan() {                            // Scan the sequence
    S x = self;
    do { x = x.next(); } while (x != null)
  }
}
```

We assume that our compiler and SAL translator transform the body of method `R.scan` into the SAL code as shown in Figure 4.1. We elide the function prologue and epilogue. Note that $\mathtt{L}_{\mathtt{null}}$ labels a fragment of code that issues a null pointer exception and $\mathtt{L}_{\mathtt{done}}$ labels the top of the epilogue (neither shown here). The notation $\mathbf{r}_x$ denotes a register name. For clarity, we have used subscripts on the register names according to the source variable to which they correspond (e.g., $\mathbf{r}_x$ corresponds to `x`) or to which role they play (e.g., `ra` holds the return address, $\mathbf{r}_{arg_0}$ holds the first argument of a function, and $\mathbf{r}_{ret}$ holds the return value of a just-returned function).

The `self` argument is passed in the $\mathbf{r}_{arg_0}$ register. The instructions in lines 3–9 implement the method dispatch `x.next`, consisting of a null check (line 3), fetching of the pointer to the dispatch table (line 4), fetching of the pointer to the method (line 5), setting the `self` argument and the return address (lines 6–7), and finally the indirect jump in line 8. This particular

[2]This example actually uses Java syntax, modified to use Cool's `self` instead of Java's `this`.

```
1  R.scan:  set r_x r_arg0
2  Loop:
3           bnez (r_x = 0) L_null
4           read r_t (r_x + 8)
5           read r_t (r_t + 4)
6           set r_arg0 r_x
7           set ra &L_ret
8           ijump r_t
9  L_ret:
10          bnez (r_ret = 0) L_done
11          set r_x r_ret
12          jump Loop
```

Figure 4.1: Assembly language version of the example.

compilation assumes that the pointer to method `next` is at offset 4 in the tables for classes S and R.

Our approach to writing a Cool verifier is following the model of the Java or MSIL bytecode verifiers: perform abstract interpretation of the code to find, for each program point, an approximation of the dynamic type of the contents of each variable [23]. The abstract state is a mapping of local variables to types in the class hierarchy. Method signatures give the initial abstraction for the method entry point. At each step through the code, the abstract state is updated to reflect the effect of the given instruction on the types of the variables. Bodies of loops might have to be scanned several times if the interpreter discovers that it needs to relax the abstract state on loop entry.

Consider, for example, the code in Figure 4.1, assuming for the moment that the code fragment in lines 3–9 is treated atomically as a virtual method dispatch to method `next` in class S and with argument x—this assumption reflects how verification would be done in the JVML or MSIL where there are specialized bytecodes for various kinds of calls.

In Figure 4.2, we show the abstract state at each program point right-justified and boxed, using the notation $\mathbf{r}_{arg_0} : R$ to say that the dynamic type of $\mathbf{r}_{arg_0}$ is a subtype of R. The abstract interpreter starts with the assumption that $\mathbf{r}_{arg_0}$ has type R, given by the signature of method `R.scan`. After it sees the assignment in line 1, the abstract state reflects that $\mathbf{r}_x$ also has type R.

Since we are treating lines 3–9 atomically, ignore the abstract state annotations marked with $*$ for now. Because R is a subtype of S, the verifier type checks the virtual call and adds the assumption that after line 8 the return value (assumed to be in $\mathbf{r}_{ret}$) has type S. The interesting point is that after the assignment in line 11, the abstract state of $\mathbf{r}_x$ changes to S. This in turn means that when the abstract interpreter reaches the start of the loop, it will have to join the abstract states after lines 1 and 11 to conclude that $\mathbf{r}_x : S$ after line 2. The abstract interpreter continues to scan the loop body until it reaches a fixpoint. Figure 4.2 shows the abstract state after the first pass through the loop, not at the fixpoint.

We now turn our attention to how our verifier can check the implementation of the virtual method dispatch. The verifier recognizes the null check, and it remembers in its abstract state that $\mathbf{r}_x$ is not null after the check. It then recognizes that reading from offset 8 into an object $\mathbf{r}_x$ yields the dispatch table of that object. We use the dependent type `dispatchPtr` $\mathbf{r}_x$ to encode this property. (Note that it is not sufficient to remember that the dispatch table belongs to an object of static type R because that would allow the fetching of the method pointer from an object whose dynamic type is different from that of the `self` parameter.) Then the verifier recognizes that we are fetching in $\mathbf{r}_t$ the pointer to the method at offset 4 in the table of $\mathbf{r}_x$, as encoded by `method` $\mathbf{r}_x$ 4. In the next line, the verifier remembers in its abstract state not only the type of the argument, but its value as well.

All of these steps collect as part of the abstract state enough information that the indirect jump instruction on line 8 can be verified, as follows. Since $\mathbf{r}_t$: `method` $\mathbf{r}_x$ 4 and $\mathbf{r}_x : R$, the verifier can consult the metadata accompanying the compiled code to find that a method `next` is being called. Since $\mathbf{r}_{arg_0} = \mathbf{r}_x$, we can check that the `self` argument is equal to the object that was used to resolve the method. Additionally, the verifier must check that the return address is correctly set and then continues the abstract interpretation of the code.

One important observation about verifying compiled code is that it is helpful to expand the notion of types beyond the programmer-level types. Thus, in addition to the types of classes, we now have the type of a dispatch table of an object, and the type of a method of an object at a particular offset. Other compiled-code-level types useful in verifying Cool include: the class tag of an object, the init method of the dynamic type of an object, etc. It is also useful to include pointer types, so that for instance we can say that if $\mathbf{r}_x$ is an object

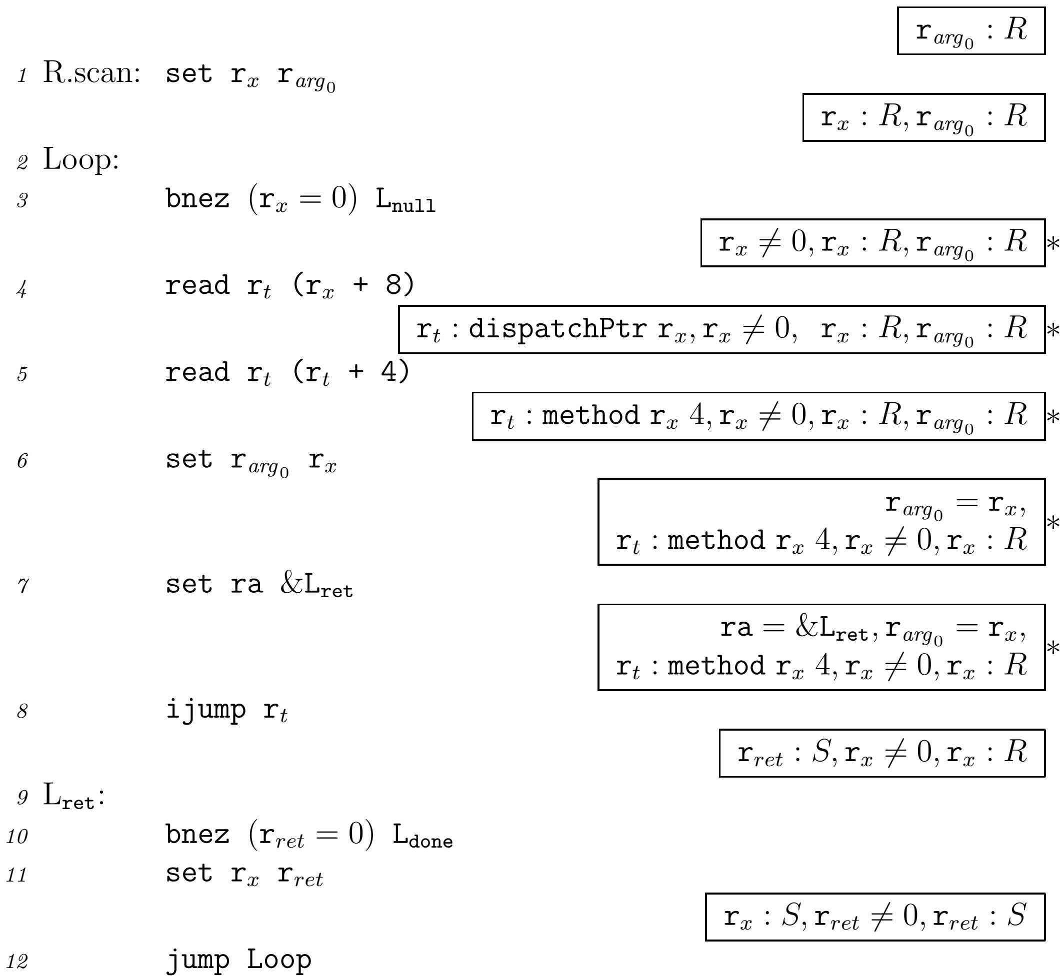

Figure 4.2: Verification of the example by abstract interpretation. (The states marked $*$ are intermediate stages of verifying the method dispatch, which would be treated atomically in most bytecode languages.)

of a class with a field of type T at offset 12, then $\mathbf{r}_x + 12$ has type $\mathtt{ptr}\ T$.

In the next section, I discuss how we can turn this verifier into an untrusted verifier for use in the Open Verifier architecture.

4.2 The Cool Extension

In this section I will discuss how to take the standalone Cool verifier described above, and build on top of it a Cool extension to work in the Open Verifier framework. This discussion is meant to describe a prototypical technique for building extensions: start with a "conventional verifier" which performs abstract interpretation on the code of interest, after the fashion of Java bytecode verifiers; then use the methods outlined in this section to add the extra information necessary for packaging as an extension.

In real verifications the Cool extension must deal constantly with the conventions of the *stack* and *function calls*; in this section, I will mostly suppress such considerations, and will discuss general ways in which any extension might handle these issues in Section 4.3 and Section 4.4.

Generally a conventional verifier works by establishing some abstract state (like a typing state) at each program point. Consider the tasks which a conventional verifier must perform:

1. to parse each instruction in the code;
2. to check that the abstract state before an instruction is strong enough to ensure safe execution of the instruction;
3. to adjust the internal abstract state according to the semantics of each instruction, and to find the successors of an instruction; and
4. to scan the code until all instructions have been verified and until the abstract state before an instruction is "weaker" than that established by all its predecessors.

The first task, of parsing the instructions, is left entirely to the trusted framework. The last task, of ensuring that the verification process complete, is also the task of the trusted framework (in fact, of the *director* module). The middle two tasks are the interesting ones from the standpoint of developing an extension. Each operates locally, one instruction at a time. Both the extension and

the decoder will perform these tasks, with the extension's results being checked against the decoder's requirement.

In effect, then, the role of the extension is to take the results of of the conventional verifier, and package them in a way that the decoder can understand, together with a proof that the results are correct. In particular, the extension must take the conventional verifier's checks of memory safety and produce a *proof* of memory safety; and the extension must take the conventional verifier's successor abstract states and produce a *proof* (in fact, a coverage proof) that they follow according to the (decoder-given) semantics of the executed instruction.

This will require, in particular, that the abstract state be described via logical predicates which are defined in such a way that these proofs can be constructed. In fact, just to be put in a form which the decoder can understand, each abstract state must be packaged as a locinv.

Recall now the interface to be exported by the extension:

$$\textbf{init} : \texttt{locinv} \rightarrow \\ (\texttt{paramsType} \rightarrow \texttt{Prop}) \times \texttt{proof} \times \texttt{locinv list} \times \texttt{coverproof}$$

and

$$\textbf{scan} : \texttt{locinv} \rightarrow \texttt{proof} \times \texttt{locinv list} \times \texttt{coverproof}.$$

The behavior discussed here corresponds to the **scan** method, though I will also bring in the `Lemmas` to be produced by **init**. For the initial coverage proof see Section 4.2.3.1.

4.2.1 Local Invariants for Cool

To create locinvs corresponding to the state of the Cool verifier, we must interpret the various features of the abstract state in logic. This is the most difficult part of building the extension. It requires observing and making explicit the *invariants* that are implicit in the behavior of the conventional verifier, and defining in the logic of locinvs those invariants and all the typing and other assertions made in the abstract states.

Finding the correct definitions is, as is often the case in logic, a matter of balance. On the one hand the definitions need to be strong enough so that the invariants and typing assertions entail enough to prove memory safety and the preservation of the invariants. On the other hand, the definitions need to be

weak enough so that the invariants can be established to hold on the initial state of execution (see Section 4.2.3.1). In practice we have found it useful to begin with only a rough idea of the invariants, and then to start in directly on the lemmas which need to hold for all the program instructions to be handled in the desired way. Then, the definitions of the invariants and other assertions can be refined in order that the lemmas become provable, while trying to retain the provability of the initial state.

4.2.1.1 Handling Recursive Types

An important early consideration has to be the handling of recursive types. How can we express recursive Cool classes in the (limited) logic of locinvs? Other approaches to recursive types in foundational proof-carrying code, such as described in [6], have relied on an *extensional* notion of types.[3] By this I mean defining the typing predicate in such a way that any block of data with the correct internal structure will have a given type—so for instance, even if a sequence of words from the compiled code of a function happens to "look like" an object of class `C`, the typing predicate would in fact consider that location in the code to be an object of class `C`.

In contrast, I will here advocate the use of an *intensional* notion of types. In practice, programs will tend use some data as a value of a certain type, only if the data was explicitly set up to be of that type (or of a type which coerces to it). For instance, in Cool, a location in memory will be considered an object of class `C`, only if that location is static object of class `C` in the data block of the program; or was allocated as such by use of `new C` (or `new D` for some subclass of `C`); or by use of `new SELF_TYPE` in a context where `self` has dynamic type some subclass of `C`; or else by a call to `Object.copy` from an object of some subclass of `C`. In fact, since the `new` expressions are implemented by the `coolc` compiler as calls to `Object.copy`, we can simplify and say that objects of class `C` are only the static objects and those arising from the use of `Object.copy`. Other objects do not happen accidentally.

The formalization of extensional recursive types is rather difficult. In fact, the authors of [5] had thought (prior to the work of [6]) that it would require formalizing substantial amounts of the mathematics of computability theory and the theory of complete metric spaces! The formalization of intensional

[3] More recently, that group has moved toward incorporating intensional aspects to handle mutable types [1].

recursive types is much simpler, as I will show. It is true that by restricting to intensional recursive types one is restricting the domain of programs which will be considered type-safe (and thus memory-safe). It seems likely, however, that one is not losing any "reasonable" programs, especially not programs produced by "reasonable" compilers.

Because of Cool's use of `Object.copy`, it is very easy to recognize at what points in the compiled code the intensional allocation state might change. With other languages and compilers this might be harder; it is not yet clear how difficult it will be to adapt a compiler for a language like OCaml so that it indicates where all allocations of objects of recursive type occur.

4.2.1.2 The Cool Typing Predicate

The most important predicate used by the Cool extension will be the `hasType` predicate, which holds of a `val` and a `coolType`, which latter is an inductive type to be defined by the extension. However `hasType` will also have other parameters. First, typing clearly depends on the particular class structure used in a given program: the inheritance hierarchy, the types and offsets of the attributes of each class, the location of the static objects in the data block of the program, etc. Using the framework of initial-state lemmas as discussed in Section 3.1.5, the extension could define a different `hasType` for each program, and prove the necessary lemmas about each particular `hasType` each time. To simplify the current discussion, I will assume a single `hasType` parametrized over a predicate, denoted by H, of the inductive type `hierarchyType`. Then the lemmas will be proven to hold parametrically in any H which meets certain properties. The extension will have to define H and prove those properties as part of the initial coverage proof for each particular program.

In order to implement the intensional recursive types as discussed above, `hasType` and other types will depend on another parameter, called *Alloc*, of the extension-supplied type `allocType`. Values of type `allocType` are association lists associating locations in memory to classes. At any point in execution (and for every locinv the extension will consider), there is a particular *Alloc* of type `allocType` describing the actual (intensional) state of the memory, including the static objects supplied with the program and all objects since allocated by `Object.copy`. Each locinv produced by the Cool extension will include an existential variable *Alloc*, and all the typing assertions in the assumptions of the locinv will use that *Alloc* as a parameter. The only points in a Cool

program where the allocation state changes is at calls to `Object.copy`; see Section 4.2.3.2.

We can now present an example of part of the definition of `hasType`:

$$\begin{aligned}\texttt{hasType}_H\ \mathit{Alloc}\ A\ (\texttt{class}\ C) \iff &\\ & A = 0 \lor \\ & (A, C) \in \mathit{Alloc} \lor \\ & \exists D.(\texttt{subtype}_H\ (\texttt{class}\ D)\ (\texttt{class}\ C)) \land (A, D) \in \mathit{Alloc}.\end{aligned}$$

Here, `class` is the type constructor for *non-basic* classes—the basic classes, which do not include the null value, require a different definition. The predicate `subtype` is the subtyping relation which, as indicated, depends on the information contained in the hierarchy data structure H. Essentially the definition fragment states that A is an object of class C if and only if A is null, or A has been explicitly allocated as a type of class C or some subclass.

4.2.1.3 Invariants

Observe that the decoder and the rest of the trusted infrastructure of the Open Verifier has no notion of the "correct" allocation state at a given point in the execution, and so no way of checking whether the extension has set it up correctly. The point is, of course, that the extension needn't set it up correctly, as long as it has the correct properties with respect to what it implies about memory safety. From the standpoint of soundness, it is perfectly acceptable for the extension to use an allocation state or even class hierarchy that has nothing to do with the program in question, as long as it (verifiably) has the same memory-safety properties. Of course, from the standpoint of creating an extension which can reliably verify Cool programs, it is usually more useful to think of the *Alloc* as being the actual one reflected by the execution of the program.

The extension does need to communicate that the *Alloc* and other parameters in use do have sensible properties. For this purpose each locinv will include among its assumptions certain invariants. The first is

$$(\texttt{hierarchyOk}\ H)$$

which states that the various information about the class structure of the program has all the consistency properties that will be needed. This includes a

range of claims such as that static objects do not overlap, and that the subclass relation is single-inheritance.

The second is

$$(\texttt{allocOk}_H \; \mathit{Alloc})$$

which states that the allocation state is consistent (with respect to the class hierarchy). This includes such claims as that the allocation state includes all the static objects from H, and that the various areas supposedly allocated do not overlap. Crucially, this invariant includes the claims that all addresses supposedly inside of allocated objects according to *Alloc* are, in fact, accessible addresses. This is the link with memory-safety and the proof obligations which will be generated by the decoder.

Finally, and most importantly, is the invariant

$$(\texttt{memOk}_H \; \mathit{Alloc} \; M).$$

This holds of a memory M. It is here that the extension finally claims that H and *Alloc* reflect something about the actual state of the machine, because the M used will be the value claimed for the contents of the memory in the `regs` of each locinv. This invariant makes such claims as that if a class C has (according to H) an attribute of type D at offset OFF, and $(x, C') \in \mathit{Alloc}$ for some subclass C' of C, then either $(\texttt{sel}\, M \; (x + \mathit{OFF}))$ is null, or $(\texttt{sel}\, M \; (x + \mathit{OFF}), D') \in \mathit{Alloc}$ for some subclass D' of D.

In practice, the extension always maintains *Alloc* and even M as existential variables. All that matters is that they satisfy the invariants; apart from that the particular values can be forgotten. The same could be done with the hierarchy H, but in practice it is helpful to maintain H explicitly, as then the extension's theorem prover can work with the H to derive needed facts, e.g. the types of attributes at various offsets, or the subtyping relation.

4.2.1.4 The Form of Cool's Local Invariants

I can now describe how the Cool extension uses the conventional Cool verifier to produce new locinvs by a translation of the conventional Cool verifier's abstract state. For example, consider the conventional Cool verifier state before the execution of line 1 in Figure 4.2, $\boxed{\mathbf{r}_{arg_0} : R}$. This translates to the following

local invariant.

$$\begin{aligned}
&\lambda\rho.\ \exists \mathit{Alloc}, \mathit{Alloc}_0 : \texttt{allocType}.\\
&\quad (\texttt{pc}\ \rho) = 1 \wedge\\
&\quad (\texttt{hierarchyOk}\ H) \wedge\\
&\quad (\texttt{allocOk}_H\ \mathit{Alloc}) \wedge\\
&\quad (\mathit{Alloc}_0 \subseteq \mathit{Alloc}) \wedge\\
&\quad (\texttt{memOk}_H\ \mathit{Alloc}\ (\mathbf{r}_M\ \rho)) \wedge\\
&\quad (\texttt{hasType}\ \mathit{Alloc}\ (\mathbf{r}_{arg_0}\ \rho)\ (\texttt{class}\ R))
\end{aligned}$$

For convenience I use the state predicate form to write the locinv; this can be easily translated to the usual form by replacing all references to registers with existential variables.[4] H is used here not as an existential variable, but as a stand-in for the explicit hierarchy value for the program.

The existential variable Alloc_0 is used to represent the allocation state at function start; it also appears in a progress continuation used to handle function return, which I have omitted here. Essentially, every function call maintains the postcondition that the allocation state at function return extends the allocation state at function call; this allows the callee to ensure that types are preserved by the function call even though the allocation state may have changed. Function calls are discussed in Section 4.4.

4.2.2 Proofs for Cool

Besides packaging the coventional verifier state as a local invariant, the Cool extension must produce the proofs required by the decoder, first to establish the local safety conditions at each potentially unsafe instruction, and second to establish (by coverage of the decoder's continuations) that the conventional verifier's state transitions do in fact follow from the semantics of the instructions executed.

I will first discuss some of the engineering aspects of proof production, from our implementation of the Open Verifier. All of the proofs to be produced

[4]For practical reasons, in the implementation it is useful to ensure that claims of *equality* between registers are represented by having the same existential variable for each, rather than introducing an equality assertion into the assumptions. This is because the Prolog interpreter Kettle, which we use to establish proofs, deals better with identity than with provable equality.

as answers to the decoder's results are obtained via an untrusted theorem prover. We use a proof-generating Prolog interpreter called Kettle. After the translation of the conventional verifier's abstract states into locinvs, the next large piece of work is creating the lemmas, which are the Prolog program rules, to be used by Kettle to produce the proofs. (Recall that even the coverage proofs can be broken down into first-order proof obligations by means of the covers proof rules of Figure 2.1.)

As mentioned above, we have found it useful to start with the lemmas/rules which are needed to prove the proof obligations that occur when verifying sample Cool programs. Only after getting a good sense of the lemmas do we then complete the definitions of the typing predicates, invariants, and other assertions used. In any case, it is important to note that during the instruction-by-instruction verification, the Cool extension never needs to refer back to the definitions themselves; the lemmas suffice for all the proof obligations occuring in this stage.

It can then be considered a separate task to actually produce the proofs of the lemmas. The lemmas and their proofs are available as a Coq script.

Now I will show by example the use of the lemmas, in the verification of a memory load and the indirect jump from the example given in Figure 4.1. I use C for the input locinv (scanned by the decoder and the extension), P for the local safety condition, $\mathcal{D}$ for the list of the decoder's locinvs, and $\mathcal{E}$ is the list of the extension's locinvs. In each of the following examples, we give one step of the director algorithm.

4.2.2.1 Memory Read

Consider the scanning of the memory read on line 5. Its safety is verified based on the fact that the contents of a dispatch table is accessible memory. Additionally, we check that the conventional verifier's state after the load, as expressed by the Cool extension, is an accurate description of the state:

$_5$ `read` $\mathtt{r}_t$ $(\mathtt{r}_t$ `+ 4)`

C: $(\mathtt{pc} = 5) \wedge (\mathtt{allocOk}\ \mathit{Alloc}) \wedge (\mathtt{memOk}\ \mathit{Alloc}\, \mathbf{r}_M) \wedge$
$(\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_t\ (\mathtt{dispatchPtr}\ \mathbf{r}_x)) \wedge$
$(\mathbf{r}_x \neq 0) \wedge (\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_x\ (\mathtt{class}\ R)) \wedge$
$(\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_{arg0}\ (\mathtt{class}\ R))$

P: $(\mathtt{addr}(\mathbf{r}_t + 4))$

$\mathcal{D}$: $\{D\}$ where $D = \exists t : \mathtt{val}.$
$(\mathtt{pc} = 6) \wedge (\mathbf{r}_t = (\mathtt{sel}\, \mathbf{r}_M\ (t + 4))) \wedge$
$(\mathtt{allocOk}\ \mathit{Alloc}) \wedge (\mathtt{memOk}\ \mathit{Alloc}\, \mathbf{r}_M) \wedge$
$(\mathtt{hasType}\ \mathit{Alloc}\, t\ (\mathtt{dispatchPtr}\ \mathbf{r}_x)) \wedge$
$(\mathbf{r}_x \neq 0) \wedge (\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_x\ (\mathtt{class}\ R)) \wedge$
$(\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_{arg0}\ (\mathtt{class}\ R))$

$\mathcal{E}$: $\{E\}$ where $E =$
$(\mathtt{pc} = 6) \wedge (\mathtt{allocOk}\ \mathit{Alloc}) \wedge (\mathtt{memOk}\ \mathit{Alloc}\, \mathbf{r}_M) \wedge$
$(\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_t\ (\mathtt{method}\ \mathbf{r}_x\ 4)) \wedge$
$(\mathbf{r}_x \neq 0) \wedge (\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_x\ (\mathtt{class}\ R)) \wedge$
$(\mathtt{hasType}\ \mathit{Alloc}\, \mathbf{r}_{arg0}\ (\mathtt{class}\ R))$

The extension must provide a proof of the local safety condition showing $\forall \rho : \mathtt{state}.\ (C\ \rho) \implies (P\ \rho)$, which is done by using the following lemma, written as an inference rule:

$$\frac{\begin{array}{c}(\mathtt{allocOk}\ \mathit{Alloc}),\\ (\mathtt{hasType}\ \mathit{Alloc}\ A_1\ (\mathtt{dispatchPtr}\ A_2)),\\ (\mathtt{hasType}\ \mathit{Alloc}\ A_2\ (\mathtt{class} T)),\\ (\mathtt{hasMethAt}\ T\ \mathit{off})\end{array}}{(\mathtt{addr}(A_1 + \mathit{off}))}$$

with $(\mathbf{r}_M\ \rho)$ for M, $(\mathbf{r}_t\ \rho)$ for A_1, $(\mathbf{r}_x\ \rho)$ for A_2, 4 for *off*, and R for T. Note that `hasMethAt` is also implicitly parameterized by the hierarchy H. Provided that class T has a method at offset *off* in its dispatch table in hierarchy H, $(\mathtt{hasMethAt}_H\ T\ \mathit{off})$ can be deduced from the structure of H.

The Cool extension is also required to give the coverage proof (i.e., a proof of $\mathcal{E}\ \mathtt{covers}\ \mathcal{D}$). We are omitting here considerations of progress continuations, so this amounts to showing $\forall \rho.(D\ \rho) \implies (E\ \rho)$. The only non-trivial assertion to be shown is $(\mathtt{hasType}\ \mathit{Alloc}\ \mathbf{r}_t\ (\mathtt{method}\ \mathbf{r}_x\ 4))$, which is shown using the

following lemma:

$$\frac{\begin{array}{c}(\texttt{memOk}\ \mathit{Alloc}\ M),\\ (\texttt{hasType}\ \mathit{Alloc}\ A_1\ (\texttt{dispatchPtr}\ A_2)),\\ (\texttt{hasType}\ \mathit{Alloc}\ A_2\ (\texttt{class}T)),\\ (\texttt{hasMethAt}\ T\ \mathit{off})\end{array}}{(\texttt{hasType}\ \mathit{Alloc}\ (\texttt{sel}\ M\ (a_1 + \mathit{off}))\ (\texttt{method}\ a_2\ \mathit{off}))}$$

with $(\mathbf{r}_M\ \rho)$ for M, t for A_1, $(\mathbf{r}_x\ \rho)$ for A_2, 4 for *off*, and R for T, along with the key assumption that $(\mathbf{r}_t\ \rho) = (\texttt{sel}\ \mathbf{r}_M\ (t+4))$ from the decoder's locinv D. Note that the above lemma has the form of a typing rule, and in fact, it can be viewed as part of the encoding of the Cool type system.

4.2.2.2 Dynamic Dispatch

Consider the scanning of the method dispatch on line 8. There are two challenges here. First, the extension must replace the indirect local invariant produced by the decoder with the list of direct local invariants that correspond to the methods that could possibly be invoked. This replacement must be accompanied by a corresponding coverage proof. Second, the coverage proof must also argue that the continuation for returning safely in each of the replacement direct invariants is itself covered by the invariant for the instruction following the call. This latter requirement is essentially similar to the situation with ordinary function calls, which are discussed further in Section 4.4.

We consider the dispatch on the second iteration of the verifier through the loop when the static type of $\mathbf{r}_x$ is assumed to be S.

$_8$ `ijump` $\mathbf{r}_t$

$$
\begin{aligned}
C\text{:}\quad & (\texttt{pc} = 8) \wedge (\texttt{ra} = \&\text{L}_{\texttt{ret}}) \wedge (\texttt{allocOk}\ \mathit{Alloc}) \wedge (\texttt{memOk}\ \mathit{Alloc}\ \mathbf{r}_M) \wedge \\
& \quad (\mathbf{r}_{arg_0} = \mathbf{r}_x) \wedge (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_t\ (\texttt{method}\ \mathbf{r}_x\ 4)) \wedge \\
& \quad (\mathbf{r}_x \neq 0) \wedge (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_x\ (\texttt{class}\ S)) \\
P\text{:}\quad & \texttt{true} \\
\mathcal{D}\text{:}\quad & \{D\} \text{ where } D = \\
& (\texttt{pc} = \mathbf{r}_t) \wedge (\texttt{ra} = \&\text{L}_{\texttt{ret}}) \wedge (\texttt{allocOk}\ \mathit{Alloc}) \wedge (\texttt{memOk}\ \mathit{Alloc}\ \mathbf{r}_M) \wedge \\
& \quad (\mathbf{r}_{arg_0} = \mathbf{r}_x) \wedge (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_t\ (\texttt{method}\ \mathbf{r}_x\ 4)) \wedge \\
& \quad (\mathbf{r}_x \neq 0) \wedge (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_x\ (\texttt{class}\ S)) \\
\mathcal{E}\text{:}\quad & \{E, I_{\texttt{R.next}}, I_{\texttt{S.next}}\} \text{ where} \\
& E = (\texttt{pc} = 9) \wedge (\texttt{allocOk}\ \mathit{Alloc}) \wedge (\texttt{memOk}\ \mathit{Alloc}\ \mathbf{r}_M) \wedge \\
& \quad (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_{ret}\ (\texttt{class}\ S)) \\
& \quad (\mathbf{r}_x \neq 0) \wedge (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_x\ (\texttt{class}\ S)) \\
& I_{\texttt{R.next}} = (\texttt{pc} = \&\texttt{R.next}) \wedge (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_{arg0}\ (\texttt{class}\ R)) \wedge \\
& \quad (\texttt{allocOk}\ \mathit{Alloc}) \wedge (\texttt{memOk}\ \mathit{Alloc}\ \mathbf{r}_M) \wedge (\mathbf{r}_{arg_0} \neq 0) \\
& I_{\texttt{S.next}} = (\texttt{pc} = \&\texttt{S.next}) \wedge (\texttt{hasType}\ \mathit{Alloc}\ \mathbf{r}_{arg0}\ (\texttt{class}\ S)) \wedge \\
& \quad (\texttt{allocOk}\ \mathit{Alloc}) \wedge (\texttt{memOk}\ \mathit{Alloc}\ \mathbf{r}_M) \wedge (\mathbf{r}_{arg_0} \neq 0)
\end{aligned}
$$

The locinvs $I_{\texttt{R.next}}$ and $I_{\texttt{S.next}}$ are the local invariants for the start of the methods `R.next` and `S.next`, respectively. Above, I have shown only the assumptions; both locinvs should contain a progress continuation stating that the method may return satisfying a certain postcondition. In both locinvs the progress continuation is the same:

$$
\begin{aligned}
C_{\texttt{RET}}\ \mathit{Alloc}\ \rho = \texttt{safe}\big(\lambda\rho'.\ \exists \mathit{Alloc}'.\ ((\texttt{pc}\ \rho') = (\texttt{ra}\ \rho)) \wedge \\
(\texttt{allocOk}\ \mathit{Alloc}') \wedge (\texttt{memOk}\ \mathit{Alloc}'\ (\mathbf{r}_M\ \rho')) \wedge (\mathit{Alloc} \subseteq \mathit{Alloc}') \wedge \\
(\texttt{hasType}\ \mathit{Alloc}'\ (\mathbf{r}_{ret}\ \rho')\ (\texttt{class}\ S)) \wedge (\mathit{CS}\ \rho\ \rho')\big).
\end{aligned}
$$

Here, ρ is the state bound by the host locinv (that is, by $I_{\texttt{R.next}}$ and $I_{\texttt{S.next}}$), and Alloc' is the existential variable for the allocation state in the host locinv. The continuation also binds its own state ρ' and existential variable Alloc'. The parameterized predicate $\mathit{CS} : \texttt{state} \to \texttt{state} \to \texttt{Prop}$ asserts that the callee-save registers are preserved. In this case,

$$\mathit{CS}\ \rho\ \rho' \implies (\mathbf{r}_x\ \rho') = (\mathbf{r}_x\ \rho).$$

Note that both $I_{\texttt{R.next}}$ and $I_{\texttt{S.next}}$ specify that on method return, the return value $\mathbf{r}_{ret}$ will be an object of class S.

In an actual verification, all of the locinvs considered by the extension (including for instance C and E in this example) would have a progress continuation specifying the postcondition of an eventual function return. I have omitted these since except for $C_{\texttt{RET}}$ as they do not affect these particular examples.

The local safety condition is immediately satisfied, but the locinv produced by the decoder is indirect, so the extension must find some way to cover it using direct locinvs. In this case, the extension recognizes this as a method call and continues after the call assuming the postcondition of the method. Note that E is not sufficient to cover D. Because the extension recognizes this indirect jump as a method call, it includes $I_{\texttt{R.next}}$ and $I_{\texttt{S.next}}$ in $\mathcal{E}$. Intuitively, we need to show it is "safe to jump to" $\mathbf{r}_t$ and that the extension's local invariants cover the possible paths after the jump (including a return).

To show that $\mathcal{E}\,\texttt{covers}\{D\}$, assume $\rho : \texttt{state}$ and $\mathit{Alloc} : \texttt{allocType}$, and let $(A_D\ \mathit{Alloc}\ \rho)$ be the assumptions of the decoder's locinv D instantiated with the existential variable Alloc and the registers of state ρ:

$$\begin{aligned}((\texttt{pc}\,\rho) = (\mathbf{r}_t\ \rho)) \wedge ((\texttt{ra}\ \rho) = \&\texttt{L}_{\texttt{ret}}) \wedge \\ (\texttt{allocOk}\ \mathit{Alloc}) \wedge (\texttt{memOk}\ \mathit{Alloc}\ (\mathbf{r}_M\ \rho)) \wedge \\ ((\mathbf{r}_{arg_0}\rho) = (\mathbf{r}_x\rho)) \wedge (\texttt{hasType}\ \mathit{Alloc}\ (\mathbf{r}_t\ \rho)\ (\texttt{method}\ (\mathbf{r}_x\ \rho)\ 4)) \wedge \\ ((\mathbf{r}_x\ \rho) \neq 0) \wedge (\texttt{hasType}\ \mathit{Alloc}\ (\mathbf{r}_x\ \rho)\ (\texttt{class}\ S))\end{aligned}$$

The Cool extension first shows the decoder's locinv is a valid method call, in this case showing

$$\begin{aligned}\big((\texttt{hasType}\ \mathit{Alloc}\ (\mathbf{r}_{arg_0}\ \rho)\ (\texttt{class}\ R)) \wedge ((\mathbf{r}_t\ \rho) = \&\texttt{R.next})\big) \vee \\ \big((\texttt{hasType}\ \mathit{Alloc}\ (\mathbf{r}_{arg_0}\ \rho)\ (\texttt{class}\ S)) \wedge ((\mathbf{r}_t\ \rho) = \&\texttt{S.next})\big)\end{aligned}$$

using the following lemma:

$$\frac{(\texttt{hasType}\ \mathit{Alloc}\ A_1\ (\texttt{method}\ A_2\ \mathit{off})),\quad (\texttt{hasType}\ \mathit{Alloc}\ A_2\ (\texttt{class}\ T))}{\bigvee_{(T',L)\in(\texttt{methLblsAt}\ T\ \mathit{off})} \big((\texttt{hasType}\ \mathit{Alloc}\ A_2\ (\texttt{class}\ T')) \wedge ((\mathbf{r}_t\ \rho) = \&L)\big)}$$

with $(\mathbf{r}_t\ \rho)$ for A_1, $(\mathbf{r}_x\ \rho)$ for A_2, 4 for *off*, and S for T. A list of each of the descendants of T (including T) along with the value of the label for its method

at offset *off* is obtained with `methLblsAt`, which is (here implicitly) a function of the hierarchy H.

The Cool extension now performs a case analysis over this disjunction, choosing in each case the appropriate locinv in $\mathcal{E}$ to cover D, in this case, either $I_{\texttt{R.next}}$ or $I_{\texttt{S.next}}$. In each case, the assumptions of D together with the the extra facts from the particular case of the disjunction provide exactly the assumptions for the covering locinv $I_{\texttt{R.next}}$ or $I_{\texttt{S.next}}$. To finish the coverage proof, in each case the Cool extension simply needs to provide a coverage proof of the progress continuation, which is the same $C_{\texttt{ret}}$ in both cases. Let $I_{\texttt{ret}} = (C_{\texttt{ret}}\ \mathit{Alloc}\ \rho)$. This "return locinv" $I_{\texttt{ret}}$ is exactly covered by the extension's new locinv E. To see this, assume a fresh ρ' : `state` and Alloc' : `allocType`, and assume $(A_{I_{\texttt{ret}}}\ \mathit{Alloc}'\ \rho')$ where $A_{I_{\texttt{ret}}}$ are the assumptions of locinv $I_{\texttt{ret}}$, i.e., we have the following:

$$((\texttt{pc}\ \rho') = (\texttt{ra}\ \rho)) \wedge \\ (\texttt{allocOk}\ \mathit{Alloc}') \wedge (\texttt{memOk}\ \mathit{Alloc}'\ (\mathbf{r}_M\ \rho')) \wedge (\mathit{Alloc} \subseteq \mathit{Alloc}') \wedge \\ (\texttt{hasType}\ \mathit{Alloc}'\ (\mathbf{r}_{ret}\ \rho')\ (\texttt{class}\ S)) \wedge ((\mathbf{r}_x\ \rho') = (\mathbf{r}_x\ \rho)).$$

We need to show that for some Alloc'' : `allocType`, that $(A_E\ \mathit{Alloc}''\ \rho')$, where A_E are the assumptions of locinv E, i.e.,

$$((\texttt{pc}\,\rho') = 9) \wedge (\texttt{allocOk}\ \mathit{Alloc}'') \wedge (\texttt{memOk}\ \mathit{Alloc}''\ (\mathbf{r}_M\ \rho')) \wedge \\ (\texttt{hasType}\ \mathit{Alloc}''\ (\mathbf{r}_{ret}\ \rho')\ (\texttt{class}\ S)) \\ ((\mathbf{r}_x\ \rho') \neq 0) \wedge (\texttt{hasType}\ \mathit{Alloc}''\ (\mathbf{r}_x\ \rho')\ (\texttt{class}\ S))$$

This is satisfied with Alloc'' taken equal to the Alloc' from $I_{\texttt{ret}}$. In particular,

$$(\texttt{pc}\,\rho') = (\texttt{ra}\,\rho) = \&L_{\texttt{ret}} = 9,$$

where the middle equality comes from the assumptions of D; similarly using that $(\mathbf{r}_x\ \rho') = (\mathbf{r}_x\ \rho)$, the assumptions of D provide

$$((\mathbf{r}_x\ \rho') \neq 0) \wedge (\texttt{hasType}\ \mathit{Alloc}\ (\mathbf{r}_x\ \rho')\ (\texttt{class}\ S)).$$

Finally we need to establish that $(\texttt{hasType}\ \mathit{Alloc}'\ (\mathbf{r}_x\ \rho')\ (\texttt{class}\ S))$, given that it holds at the "old" allocation state *Alloc*. This can be establish using

the assumption of $I_{\texttt{ret}}$ that $Alloc \subseteq Alloc_0$, as a case of the following general lemma:

$$\frac{(\texttt{hasType}\ Alloc\ A\ T), \quad (Alloc \subseteq Alloc')}{(\texttt{hasType}\ Alloc'\ A\ T)}$$

4.2.3 Completing the Cool Extension

In order to complete the Cool extension, it is necessary to describe the handling of the initial coverage proof, and of the run-time support functions. These require a somewhat different approach. In the implementation of scanning described by examples above, the proofs boil down to facts which come more or less directly from the Cool type system. In the initial locinv and the run-time functions, however, the Cool type system is not enforced: in the case of the initial locinv it is not yet in place, and in the certain of the run-time functions it is temporarily set aside. The job of the extension, then, is to show that the framework we have set up for the Cool type system can in fact be superimposed on the initial locinv, and will in fact be preserved by the end of each call to a run-time function.

At time of writing, our prototype implementation has only just begun to incorporate this work. I sketch here the anticipated approach.

4.2.3.1 The Initial Coverage

So far I have concentrated on the main iterative behavior of the Cool extension: responding, step by step, to the proof obligations given by the decoder after scanning other of the extension's locinvs. It is still necessary to consider the beginning of the process, that is, the initial locinvs created by the extension. Recall from Section 3.4.4 that the decoder produces an initial locinv describing in substantial detail the initial state of the program. In particular, the initial state of the memory is given. Thus, the decoder's initial locinv asserts that each memory address in the stack, heap, and static data of the program satisfies the `addr` predicate; and for the static data, it also specifies the contents of every memory location.

The extension's job is to produce its own initial locinvs, and to establish that its locinvs cover the decoder's. In practice we have found it convenient to have the extension produce all of the locinvs corresponding to the start of

each function in the program at this time. These locinvs can be referred to again at the function call points, rather than being created then. However, from the standpoint of producing the initial coverage proof, all that is required is one locinv corresponding to the actual entry point. In Cool programs this is a function provided by the run-time library `trap.handler`, which essentially corresponds to the Cool code `(new Main).main()`.

The decoder's initial locinv contains a lot of information, but not organized in a way which is useful to the extension. The extension's initial locinv will abstract away the contents of the memory M, only it will need to establish the invariant that ($\texttt{memOk}_H$ *Alloc* M), and the other invariants. This initial coverage proof, unlike the proofs described earlier which relied only on the lemmas given, will depend more directly on the definitions of the predicates.

For instance, the invariants include the claims that for every address A which can be proven to have the type (`ptr` T), for some T, then (`addr` A) holds, and furthermore (`sel` M A) has the type T. The approach we will use to this is to first prove an exhaustive categorization of the the typing relationship which holds under the initial, explicitly given *Alloc*. We give, in fact, an explicit list of all the addresses which have a `ptr` type; and then for each member of this list, we check that it has the properties required of it. This work would be tedious for a human, but it does not seem likely to be difficult to implement.

4.2.3.2 Cool's Run-time Functions

One thing is left to complete the Cool extension. Most of a compiled Cool program can be understood using a particular set of lemmas reflecting the types and invariants described above. But the run-time support functions will mostly require a more general approach. For instance, the allocation state *Alloc* remains unchanged outside of calls to `Object.copy`. Lemmas about the `hasType` predicate with respect to a single particular *Alloc* suffice everywhere else; and the call sites can be handled by maintaining that $Alloc_0 \subseteq Alloc$ where $Alloc_0$ is the allocation state before the call, and *Alloc* is the allocation state after the call. Then a general lemma can be used to tranfer the typing state before a function call to the state after a function call. (This is used at every function, since any function might potentially call `Object.copy`.)

Verifying the body of `Object.copy` itself is more difficult. For simplicity I will ignore garbage collection and assume a simple `Object.copy` which just finds the next block of free space of the appropriate size, and produces a byte-

wise copy of its argument in that space. The difficulty comes from establishing that the invariants hold with the new memory and *Alloc* afterwards.

The approach I propose is similar to that described above for handling the initial coverage proof. A characterization of all the addresses with pointer types is given, separated into all of those which have that type in the old allocation state, and an explicit list of the "new" pointers resulting from the allocation. Then the invariants are proven directly from the definitions with reference to this characterization.

4.3 Stack Handling

In actual Cool programs, many of the values which I have carried in registers in the examples of the last section would in fact be carried on the *stack*.

The trusted infrastructure of the Open Verifier does not have a special notion of a stack. Reads and writes to the stack are interpreted exactly as any other memory reads and writes. It is up to the extension to maintain special invariants which allow it to handle stack accesses in a particular way. This allows us to keep the trusted infrastructure as generic as possible, and in particular to leave all handling of software conventions to the extension.

In this section I will set up a modular approach to stack handling which could be incorporated into various extensions.

There are several features relating to the conventional use of the stack which I intend to model:

- Most invariants, not dealing with explicitly with the stack, only depend on the state of the memory *apart from* the stack. For the sake of efficiency this dependence should be explicit in the logic, so that the (rather frequent) writes to the stack do not require re-establishing that the non-stack invariants hold.

- A function only modifies a certain area of the stack (its *stack frame*). In particular, the caller function can assume that the stack above the callee's frame is preserved.

- Each word in a function's stack frame should be able to be handled essentially as if it were a machine register.

- Finally, if the program is actually memory safe, it must have some provision for checking that *stack overflow* does not occur.

4.3.1 Memory Regions

To handle the notion that stack changes should not affect various invariants to be maintained about the rest of the memory, I introduce the concept of *memory regions*. The type `region` will be the type of *sets of memory addresses*, where membership in a region is given by the predicate

$$\texttt{InRegion} : \texttt{region} \to \texttt{val} \to \texttt{Prop}.$$

In examples, and in the implemented extensions, there are two regions `stack` and `heap` (which latter name may be slightly abusive, as it is intended to include the program's data segment as well as any other memory given to it by the operating system).

A `regionSet` is an association list which associates regions to memories. So for instance, for memories S and H of type `mem`,

$$[(\texttt{stack}, S); (\texttt{heap}, H)] : \texttt{regionSet}.$$

The idea is that the extension will use a `regionSet` in place of the actual current memory, introducing functions

$$\texttt{regUpd} : \texttt{regionSet} \to \texttt{region} \to \texttt{val} \to \texttt{val} \to \texttt{regionSet};$$
$$\texttt{regSel} : \texttt{regionSet} \to \texttt{region} \to \texttt{val} \to \texttt{val}$$

corresponding to `upd` and `sel` for type `mem`. (In principle the functions need not take the `region` in which the update or select occurs, instead determining which region the given address belongs too. But practically one always can determine which region is intended beforehand, and this approach is more efficient.)

For `regUpd` and `regSel` to be sensible, the `regionSet` needs to satisfy an invariant

$$\texttt{regions} : \texttt{mem} \to \texttt{regionSet} \to \texttt{Prop}.$$

The predicate (`regions` M R) holds when

- for each (reg, M_{reg}) in R, M and M_{reg} agree on all addresses in reg;
- the regions in R are pairwise disjoint;

- for each region *reg* in R, `addr` holds of all addresses in *reg*.

The first requirement is clearly what is needed to use a `regionSet` in place of the memory. The other two requirements are a convenience: pairwise-disjointness means one never needs to worry about updating more than one region, and the requirement that all the addresses be `addr` means that the `InRegion` predicate will suffice to establish local memory-safety conditions.

There are straightforward lemmas which allow an extension to use a `regionSet` R for which (`regions` M R), such as selecting values in such a way that the value selected is the same as that selected from M, and updating the `regionSet` so that the predicate `regions` is preserved. I will not go into detail here.

An extension using the regions framework will maintain in each local invariant existential variables S and H of type `mem`, and include an assumption

$$\texttt{regions}\ M\ [(\texttt{stack}, S); (\texttt{heap}, H)]$$

where M is the memory of the locinv. All the assumptions relevant to the stack will use S instead of M; all the assumptions relevant to the heap (such as the Cool extension's `memOk`) will use H instead of M. In particular, this shields assumptions about the heap from any memory changes occuring only on the stack.

It is not difficult to work with regions which change over the course of the program, due for instance to memory allocation, but I will not discuss this further here.

4.3.2 Stack Frames and Stack Preservation

Nothing in the memory-safety policy requires that a given function only touch some particular part of the stack. But enforcing a notion of *stack frame* is extremely helpful in ensuring general memory safety. Within any given function it is assumed that there is a given range of accessibly memory addresses from L to H. L and H are specified as offsets from the initial stack pointer $\texttt{sp}_0$, the stack pointer on function entry. The relationship between $\texttt{sp}_0$, L, and H is determined by the calling convention used by the function, which can be obtained by the extension via code annotations (which can generally be automatically generated). Usually, any arguments to the function which are placed on the stack are at $\texttt{sp}_0$, $\texttt{sp}_0 + 4$, $\ldots H$; and then L is less than H by

the size of the function's stack frame. The stack frame size can be given by annotations, or can be fixed at some amount like four kilobytes (in which case $L = H - 4092$).

This process is implemented by including in each locinv the assumption (`StackFrame` L H). This is defined so that it implies

$$\forall A. L \leq A \implies A \leq H \implies (\texttt{InRegion stack } A).$$

(The full definition will include more in order to handle stack-overflow checking mechanisms, see below.) The intention is that the extension will use only this method to prove that addresses are on the stack (and are thus valid to read and write).

Some additional work is needed to correctly handle function calls. The caller needs to be able to establish that its own stack, outside of the callee's stack frame, has not been modified. This is entailed by the predicate (`PreservedStack` S_0 H S), which is defined to mean

$$\forall A.\ H \leq (A - 4) \implies (\texttt{sel}\, S_0\ A) = (\texttt{sel}\, S\ A).$$

That is, the stack memories S_0 and S coincide at addresses strictly above H. Each locinv within a function will carry a `PreservedStack` assumption with S_0 the initial stack memory at function entry, H the highest member of the function's stack frame, and S the current stack memory; the function return progress continuation (see Section 4.4) will also carry such an assumption. With this in place, the caller function will have the assumption available after the call, and so can establish that all of its stack slots include the values they had before the call.

4.3.3 Stack Slots and Stack Pointers

Internally, the extension maintains registers and stack slots in essentially the same fashion; in fact, the current implementation uses a single array, in which indices higher than the number of machine registers correspond to the stack slots. For each register or stack slot a symbolic expression is stored which gives its value. To present this as a locinv, the values of the registers are stored in the `regs` field, whereas the values of the stack slots are stored as assumptions that $(\texttt{sel}\, S\ (\texttt{sp}_0 + \mathit{offset})) = \texttt{slot}_{\mathit{offset}}$, where S is the current stack memory, and

$\texttt{sp}_0$ is the initial stack pointer at function entry. (In fact, to avoid extra non-determinism in how the Kettle theorem prover uses equality facts, a defined predicate ($\texttt{FrameContents}$ ($\texttt{sp}_0 + \mathit{offset}$) $\texttt{slot}_{\mathit{offset}}\ S$) is used instead.)

Whenever a memory read or write occurs, the stack module first checks to see whether it was on the stack. If so, it proves the local safety condition using the `StackFrame` assumption; then it proceeds as if the instruction were a simple register move, only involving a stack-slot pseudo-register. In the case of a read, the correctness of the result is established using the relevant `FrameContents` assumption; in the case of a write, a `FrameContents` assumption is added or altered. At function calls the stack slots of the caller and callee have to be related by a translation of the different offsets.

Meanwhile, the extension must keep track of which registers and stack slots are not storing values directly, but are storing pointers to stack locations—are stack pointers. Stack pointers are always maintained in the form $\texttt{sp}_0 + \mathit{offset}$ where $\texttt{sp}_0$ is the initial stack pointer on function entry.

4.3.4 Stack Overflow

Each locinv generated by the extension will contain a `StackFrame` assumption. At function calls, the stack frame will change. For non-recursive calls, the callee's stack frame can be considered to be contained in the caller's stack frame. For recursive calls this is insufficient: the stack frame needs to grow, and thus for memory safety, there needs to be some mechanism to check stack overflow. At the level of the logic this is primarily reflected in the eventual complete definition of `StackFrame`. Here I describe two approaches to stack overflow.

4.3.4.1 The Stack on a 1MB Page

Under this approach, it is assumed that an entire 1MB page has been devoted to the stack, and thus that if two addresses are on the same 1MB page, one of which is a stack address, then both are stack addresses. A program can guarantee that there is no stack overflow by adding run-time checks before recursive function calls, to ensure that the low address of the callee's frame is in the same 1MB page as the low address of the caller's frame. Explicitly, if the two addresses are l_{old} and l_{new}, it is checked that

$$\texttt{srl}\ (\texttt{bxor}\ l_{old}\ l_{new})\ 20 = 0,$$

where `bxor` is the bitwise exclusive or, and `srl` is a shift-right (in this case, by 20 bits). The condition ensures that the two addresses are on the same 1MB page by ensuring that all bits to the left of the 20th agree. If this condition fails, the program (safely) aborts.

The logical support for this approach is the following definition of `StackFrame`:

Definition 4.3.1. The predicate (`StackFrame` l h) holds if and only if (`InRegion stack` a) holds for all a such that either:

- $l \leq a \leq h$; or
- `srl` (`bxor` x a) $20 = 0$ for some x such that $l \leq x \leq h$.

That is, every address between l and h, or else on the same 1MB page as such an address, is on the stack.

The lemma actually used to handle the frame change at stack checks and function calls is:

Lemma 4.3.2. *Let l_{old}, h_{old}, l_{new}, and h_{new} be addresses, where* (`StackFrame` l_{old} h_{old})*. Then* (`StackFrame` l_{new} h_{new}) *if $h_{new} \leq h_{old}$ and either of the following two conditions hold:*

- $l_{old} \leq l_{new}$*, or*
- *there exists an address l_{check} such that $l_{old} \leq l_{check} \leq h_{old}$ and* `srl` (`bxor` l_{check} l_{new}) $20 = 0$.

4.3.4.2 Guard Pages

A technique often used to prevent stack overflow is to ensure that at there is a certain area of unmapped virtual memory at the bottom of the stack. Any attempt to read or write this memory—which will tend to happen when a stack overflow occurs—will result in a (potentially handleable) segmentation fault. This technique is usually considered important for multithreaded applications, but is entirely applicable to single-threaded programs which must be memory safe.

Under this approach, checking against stack overflow is merely reading or writing the stack, where if a stack overflow occurs, the read or write is in the

guard page, and a segmentation fault results. To use this approach, the safety policy must consider the addresses in the guard page to be valid addresses, even though reading or writing there results in a segmentation fault. There is a potentialy confusion about this approach, which might seem to enable one to claim only vacuously that "the program is memory safe, or else a segmentation fault occurs". In fact it is guaranteed that any segmentation fault is due precisely to stack overflow, and that every memory access up until the segmentation fault occurs is genuinely valid. In effect, the segmentation fault takes the place of an ordinary call to safely abort in the presence of stack overflow.

In order for the guard page to guarantee that any stack overflow will cause a segmentation fault, the program must ensure that no two stack accesses are separated by more than the size of the guard page. The `gcc` compiler will introduce any extra checks needed, when it is invoked with the argument `-fstack-check`. In particular `gcc -fstack-check` will produce a "stack probe" at the beginning of every function which calls other functions. If the size of the guard page is 4096 bytes, then the stack probe will write a zero into the address 4096 bytes below the lowest argument of the caller's stack frame.[5] Observe that the program will only be guaranteed safe with respect to stack overflow under this scheme, if no function has a stack frame larger than 4096 bytes.

The Open Verifier framework will support this approach to stack overflow, but the decoder must be modified, to allow for the fact that memory reads and writes to valid addresses may cause an abort (actually, a segmentation fault). In effect, the semantics of memory accesses are as if they were branch instructions. If the address is in the guard page, the program aborts (safely). Otherwise the program continues to the next instruction.

I introduce the logical predicate `NotInGuardPage` : `val` → `Prop` which holds of all addresses not in the guard page. The decoder of Figure 3.3 is then modified as follows:

[5] The actual behavior of `gcc -fstack-check` is slightly more complicated. Firstly, it tries to ensure that 300 bytes of stack are available for any stack overflow recovery mechanism. This can be handled by considering each function's stack frame to be 300 bytes larger than the stack space which is actually used by the function. Secondly, it handles the additional checks necessary when allocating large objects on the stack, which I do not discuss here.

Instruction	P	$\mathcal{D}$
`read` r a	$\lambda\rho.\,$`addr`$(a\ \rho)$	$\{C$ with `regs` $= \lambda\boldsymbol{x}.\rho[$`pc` $\mapsto$ (`pc` $\rho +_{\mathrm{SAL}} 1)$; $r \mapsto \big($`sel`$(\mathbf{r}_M\ \rho)\ (a\ \rho)\big)]$, `assume` $= \lambda\boldsymbol{x}.\,($`NotInGuardPage` $(a\ \rho)) \wedge (C.$`assume` $\boldsymbol{x})\}$
`write` a e	$\lambda\rho.\,$`addr`$(a\ \rho)$	$\{C$ with `regs` $= \lambda\boldsymbol{x}.\rho[$`pc` $\mapsto$ (`pc` $\rho +_{\mathrm{SAL}} 1)$; $\mathbf{r}_M \mapsto \big($`upd`$(\mathbf{r}_M\ \rho)\ (a\ \rho)\ (e\ \rho)\big)]$, `assume` $= \lambda\boldsymbol{x}.\,($`NotInGuardPage` $(a\ \rho)) \wedge (C.$`assume` $\boldsymbol{x})\}$

The decoder result could be considered as having a second continuation, corresponding to the abort state resulting from an attempt to access the guard page; but since this state is considered safe, we can ignore it. (Recall from Definition 2.6.8 that safe next states do not have to be included in the decoder's output continuations.)

We now use the following definition for `StackFrame`:

Definition 4.3.3. The predicate (`StackFrame` l h) holds if and only if (`InRegion stack` a) holds for all a such that $(l - 4096) \leq a \leq h$.

So the stack—including the guard page—is always assumed to extend a full page below the bottom of the current function's frame. We use the following lemma (which is obvious given a contiguous stack with a guard page at the borrom):

Lemma 4.3.4. *Let a be an address such that* (`InRegion stack` a) *and* (`NotInGuardPage` a)*. Then for any address b such that $(a - 4096) \leq b \leq a$, we have that* (`InRegion stack` b)*.*

It is now easy to establish the lemma which allows the stack frame to change at function calls and stack checks:

Lemma 4.3.5. *Let l_{old}, h_{old}, l_{new}, and h_{new} be addresses, where* (`StackFrame` l_{old} h_{old})*. Then* (`StackFrame` l_{new} h_{new}) *if $h_{new} \leq h_{old}$ and either of the following two conditions hold:*

- *$l_{old} \leq l_{new}$, or*

- (NotInGuardPage l_{new}), *and* $(l_{old} - 4096) \leq l_{new} \leq h_{old}$.

When a stack probe occurs, the decoder will add the assumption that the probed address is not in the stack guard page. (To repeat, if it had been in the guard page, the program would have aborted safely.) The extension can then use this lemma to extend the stack frame down to the probed address.

4.4 Function Calls

In SAL a function call is simply a direct jump. Those features which make a jump into a function call are just conventions, typically enforced by the higher-level programming language and compiler. These conventions typically center around the notion of *return*: the function may eventually return control, via an indirect jump, to some code specified by the initial values of the registers after the function call. Often there is a specified *return-address register* which holds the code location to jump to. The software conventions further specify what conditions must hold on return, e.g. that the callee can assume that certain registers were not changed during the function call.

I will now discuss methods by which an extension can describe these conventions, and use them to prove the safety of a program involving functions.

4.4.1 Using Progress Continuations

Function returns are the prototypical instance of using the progress continuations of locinvs. Recall that any locinv C can contain a list C.`progress` of locinvs, parametrized by C.`type`. The meaning, as given by the satisfaction relation, is essentially that these locinvs are assumed to be safe: should the machine reach a state satisfying one of the progress continuations, (indefinite) further progress is possible.

So to handle function calls, it is assumed that it is safe to return. For simplicity assume that there is a dedicated return-address register `ra`. A function F which contains no potentially unsafe instructions, thereby requiring no preconditions to be safe, and which guarantees nothing to the caller, thereby requiring no postconditions, would be described by a locinv C where $\rho \models_i C$ if any only if

$$\exists x.(\mathtt{pc}\,\rho) = F \land (\mathtt{ra}\,\rho) = x \land \forall j \leq i.\ \mathtt{safe}_j(\lambda\rho'.(\mathtt{pc}\,\rho') = x).$$

Note that this locinv refers to the initial state of execution of the function. The existential variable x is the original value of the return address register. If the return address register changes, the extension has to keep track of this x in some other way, for it is used in the progress continuation. So for instance if by line n the ra has been incremented, one might have

$$\exists x.(\mathtt{pc}\,\rho) = n \wedge (\mathtt{ra}\,\rho) = x + 1 \wedge \mathtt{safe}_i(\lambda\rho'.(\mathtt{pc}\,\rho') = x).$$

Most commonly, of course, the value represented by x will be saved (on the stack) whenever the ra register needs to hold another value (for another function call). Then the ra can be restored to have the value x before the return (an indirect jump to ra).

Under this assumption that it is safe to return, the locinvs from the body of the function can be established as safe. Within the body of the function, the progress continuation is carried along unchanged and does not affect the proofs. At function call sites, the continuation of the callee must be covered by a locinv corresponding to the actual return site for that call; an example of this is shown in Section 4.2.2.2. Finally, at the return instruction, the progress continuation is itself used to establish coverage. Consider, for example, the method R.next mentioned in Section 4.2.2.2. The extension establishes the following locinv for the start of that function:

$$\begin{aligned} I_{\mathtt{R.next}} = \lambda\rho.\ \exists Alloc.\ & ((\mathtt{pc}\,\rho) = \&\mathtt{R.next}) \wedge \\ & (\mathtt{allocOk}\ Alloc) \wedge (\mathtt{memOk}\ Alloc\ (\mathbf{r}_M\ \rho)) \wedge \\ & (\mathtt{hasType}\ Alloc\ (\mathbf{r}_{arg0}\ \rho)\ (\mathtt{class}\ R)) \wedge ((\mathbf{r}_{arg_0}\ \rho) \neq 0) \wedge \\ & \mathtt{safe}\big(\lambda\rho'.\ \exists Alloc'.\ ((\mathtt{pc}\ \rho') = (\mathtt{ra}\ \rho)) \wedge \\ & \quad (\mathtt{allocOk}\ Alloc') \wedge (\mathtt{memOk}\ Alloc'\ (\mathbf{r}_M\ \rho')) \wedge (Alloc \subseteq Alloc') \wedge \\ & \quad (\mathtt{hasType}\ Alloc'\ (\mathbf{r}_{ret}\ \rho')\ (\mathtt{class}\ S)) \wedge ((\mathbf{r}_x\ \rho') = (\mathbf{r}_x\ \rho))\big). \end{aligned}$$

Observe that the progress continuation refers to the return address register ra, the callee-save register $\mathbf{r}_x$, and the allocation state *Alloc* of the host locinv. During the execution of the function these values might change, but the values relevant to the progress continuation will continue to be the values at function state; so if any change, a new existential variable will have to be introduced to hold the original value. We can do this pre-emptively and re-express the locinv

as

$$I_{\texttt{R.next}} = \lambda\rho.\ \exists ra_0, x_0, Alloc_0, Alloc.\ ((\texttt{pc}\ \ \rho) = \&\texttt{R.next}) \wedge$$
$$((\texttt{ra}\,\rho) = ra_0) \wedge ((\mathbf{r}_x\ \rho) = x_0) \wedge (Alloc = Alloc_0) \wedge$$
$$(\texttt{allocOk}\ Alloc) \wedge (\texttt{memOk}\ Alloc\ (\mathbf{r}_M\ \rho)) \wedge$$
$$(\texttt{hasType}\ Alloc\ (\mathbf{r}_{arg0}\ \rho)\ (\texttt{class}\ R)) \wedge ((\mathbf{r}_{arg_0}\ \rho) \neq 0) \wedge$$
$$\texttt{safe}\big(\lambda\rho'.\ \exists Alloc'.\ ((\texttt{pc}\ \ \rho') = ra_0) \wedge$$
$$(\texttt{allocOk}\ Alloc') \wedge (\texttt{memOk}\ Alloc'\ (\mathbf{r}_M\ \rho')) \wedge (Alloc_0 \subseteq Alloc') \wedge$$
$$(\texttt{hasType}\ Alloc'\ (\mathbf{r}_{ret}\ \rho')\ (\texttt{class}\ S)) \wedge ((\mathbf{r}_x\ \rho') = x_0)\big).$$

Now suppose that the verification reaches a return instruction. If `ra` or $\mathbf{r}_x$ have changed during the function, they have been restored to their original value. In the input locinv C, the progress continuation has been carried unchanged since the start of the function. Generally C might contain many more assumptions; here I have shown only those which are relevant to the return.

n `ijump ra`

C: $\lambda\rho.\ \exists ra_0, x_0, Alloc_0, Alloc.$
$((\texttt{pc}\ \ \rho) = n) \wedge (\texttt{hasType}\ Alloc\ (\mathbf{r}_{ret}\ \rho)\ (\texttt{class}\ S)) \wedge$
$((\texttt{ra}\,\rho) = ra_0) \wedge ((\mathbf{r}_x\ \rho) = x_0) \wedge$
$(\texttt{allocOk}\ Alloc) \wedge (\texttt{memOk}\ Alloc\ (\mathbf{r}_M\ \rho)) \wedge (Alloc_0 \subseteq Alloc) \wedge$
$\texttt{safe}\big(\lambda\rho'.\ \exists Alloc'.\ ((\texttt{pc}\ \ \rho') = ra_0) \wedge$
$(\texttt{allocOk}\ Alloc') \wedge (\texttt{memOk}\ Alloc'\ (\mathbf{r}_M\ \rho')) \wedge (Alloc_0 \subseteq Alloc') \wedge$
$(\texttt{hasType}\ Alloc'\ (\mathbf{r}_{ret}\ \rho')\ (\texttt{class}\ S)) \wedge ((\mathbf{r}_x\ \rho') = x_0)\big).$

P: `true`

$\mathcal{D}$: $\{D\}$ where $D = \lambda\rho.\ \exists ra_0, x_0, Alloc_0, Alloc.$
$((\texttt{pc}\ \ \rho) = (\texttt{ra}\,\rho)) \wedge (\texttt{hasType}\ Alloc\ (\mathbf{r}_{ret}\ \rho)\ (\texttt{class}\ S)) \wedge$
$((\texttt{ra}\,\rho) = ra_0) \wedge ((\mathbf{r}_x\ \rho) = x_0) \wedge$
$(\texttt{allocOk}\ Alloc) \wedge (\texttt{memOk}\ Alloc\ (\mathbf{r}_M\ \rho)) \wedge (Alloc_0 \subseteq Alloc) \wedge$
$\texttt{safe}\big(\lambda\rho'.\ \exists Alloc'.\ ((\texttt{pc}\ \ \rho') = ra_0) \wedge$
$(\texttt{allocOk}\ Alloc') \wedge (\texttt{memOk}\ Alloc'\ (\mathbf{r}_M\ \rho')) \wedge (Alloc_0 \subseteq Alloc') \wedge$
$(\texttt{hasType}\ Alloc'\ (\mathbf{r}_{ret}\ \rho')\ (\texttt{class}\ S)) \wedge ((\mathbf{r}_x\ \rho') = x_0)\big).$

$\mathcal{E}$: $\{\ \}$

The extension proves that the decoder's locinv D is covered by *no* locinvs; that is, D proves its own safety, because it is covered by its own progress

continuation. To see this, fix ρ and the existential variables x_0, ra_0, $Alloc_0$, and $Alloc$; and assume that the assumptions of D hold:

$$((\texttt{pc}\ \rho) = (\texttt{ra}\,\rho)) \wedge (\texttt{hasType}\ Alloc\ (\mathbf{r}_{ret}\ \rho)\ (\texttt{class}\ S)) \wedge \\ ((\texttt{ra}\,\rho) = ra_0) \wedge ((\mathbf{r}_x\ \rho) = x_0) \wedge \\ (\texttt{allocOk}\ Alloc) \wedge (\texttt{memOk}\ Alloc\ (\mathbf{r}_M\ \rho)) \wedge (Alloc_0 \subseteq Alloc)$$

We just need to establish that for some value of the existential variable $Alloc'$, the assumptions of the progress continuation hold at the same state ρ:

$$((\texttt{pc}\ \rho) = ra_0) \wedge \\ (\texttt{allocOk}\ Alloc') \wedge (\texttt{memOk}\ Alloc'\ (\mathbf{r}_M\ \rho')) \wedge (Alloc_0 \subseteq Alloc') \wedge \\ (\texttt{hasType}\ Alloc'\ (\mathbf{r}_{ret}\ \rho)\ (\texttt{class}\ S)) \wedge ((\mathbf{r}_x\ \rho) = x_0)$$

But this is clearly the case just taking $Alloc' = Alloc$.

4.4.2 Another Approach to Returns

Above I have advocated handling function returns by setting up a progress continuation which refers to some content of the host locinv:

$$\exists x.(\texttt{pc}\,\rho) = F \wedge (\texttt{ra}\,\rho) = x \wedge \forall j \leq i.\ \texttt{safe}(\lambda\rho'.(\texttt{pc}\,\rho') = x).$$

Here, the progress continuation references the variable x set up by the surrounding context of the host locinv.

Another approach is to statically enumerate all possible return points of function F. For instance, suppose that F is called three times, with the return points being lines 17, 191, and 321. Then the following locinv could be used for F:

$$\exists x.(\texttt{pc}\,\rho) = F \wedge (\texttt{ra}\,\rho) = x \wedge (x = 17 \vee x = 191 \vee x = 321).$$

There is no need for any progress continuation in this locinv. Instead, at the return instruction, a case analysis is performed. The three locinvs corresponding to lines 17, 191, and 321 will cover the decoder's locinv.

This could be lemmatized, by covering once and for all the single locinv

$$(\texttt{pc}\,\rho) = 17 \vee (\texttt{pc}\,\rho) = 191 \vee (\texttt{pc}\,\rho) = 321.$$

Then this "global progress continuation" can be used at each return instruction. Such lemmatization improves efficiency if there are many return instructions, and many possible return points.

This approach is simpler in terms of the complexity needed from the framework: no progress continuations are involved. However it suffers from potential drawbacks in extensibility. Using progress continuations, we could potentially prove the safety of library code: the locinv for F corresponds to all function calls to F, even those not part of the library code and thus not available when safety is being established.

It is not as obvious how to do this using the disjunction approach. It could be possible to extend the disjunction with some kind of parameter:

$$x = 17 \lor x = 191 \lor x = 321 \lor x \in R,$$

where R is intended to be the list of return points in linked code. The library code could then be proven safe for all possible values of R, under the assumption that various locinvs from the linked code are safe. I have not worked out exactly how this could be accomplished, and in any case it seems more complicated overall than the version using progress continuations.

In the case of a having the completely linked code available for static analysis, the approach using disjunctions can be easier. I would advocate its use for certain kinds of indirect jumps where extensibility is probably not an issue, for instance the indirect jumps used to implement switch statements, where the various cases are all locally available. The current implementation uses disjunctions to implement method dispatch for the object-oriented language Cool (Section 4.2.2.2); further work would be needed if it were to apply extensibly to the safety of Cool libraries.

Chapter 5

Conclusions

In the preceding chapters I have described the Open Verifier system for enforcing safety properties, such as memory safety, of untrusted code. The system allows the code producer to provide an untrusted verifier, called the *extension*, as executable code; the trusted components of the Open Verifier then work together with the untrusted extension to produce a trustworthy verification. I have described the *logic* and *architecture* of the interaction between trusted and untrusted components which makes this possible. I have also described an approach to developing extensions, in particular a *proof development approach*, which we have used successfully to verify the memory safety of compiled code from the Java-like Cool language.

In this final chapter I conclude by giving a preliminary evaluation of the Open Verifier as a security enforcement mechanism; a comparison with related work; and directions for future research.

5.1 Evaluating the Open Verifier

At this point it is appropriate to return to the metrics of *trustworthiness*, *flexibility*, and *scalability* which I put forward in the Introduction. To get useful measurements it is necessary to refer to a real implementation; our implementation is still in a preliminary stage, so some of what I discuss here is also preliminary. Nonetheless the project is now advanced enough for a somewhat accurate appraisal.

	Untrusted		Trusted	
	module	# lines	module	# lines
OCaml	`Convcool`	3,400	`Openver`	1,500
	`Convstack`,`Convfunc`	500	`Proof`	700
	`Coolext`	1,500	`SAL`	900
	`Stackext`,`Funcext`	1,400	`MIPSparse`	1,000
	`Kettle` (approx.)	3,000		
	Total	9,800	**Total**	4,100
Kettle	`cool.rules`	1,000		
	`stack.rules`	350		
Coq	`cool.v` (est.)	2,500	Soundness theorem	150
	`stack.v`	400		

Figure 5.1: Number of lines of code in the Open Verifier.

5.1.1 Trustworthiness

Trustworthiness can be roughly measured by the amount of code in the trusted code base of the Open Verifier. Figure 5.1 shows the lines of code in both the trusted and untrusted modules (written in OCaml) used to verify Cool, as well as the lines of lemma statements (written as Prolog proof rules for the automatic theorem prover) and proofs of the lemmas (written in Coq). I will refer to this figure again as an estimate of the work required to build from a conventional verifier to an extension for the Open Verifier.

`Convcool` is the conventional verifier for Cool, which works similarly to a bytecode verifier but directly on MIPS code rather than specialized bytecodes. `Convstack` and `Convfunc` are also used in the conventional verifier, but are generic modules for handling the run-time stack and function calls, which can also be used by other conventional verifiers. `Coolext` is the code for the Cool extension. One of its main jobs is to take the abstract states given by `Convcool` and wrap them into locinvs. `Stackext` and `Funcext` are generic modules to be used by extensions to work with `Convstack` and `Convfunc`.

`Kettle` is a proof-generating Prolog interpreter used by the extension to automatically establish proof obligations. Using Kettle requires providing lemmas given as Prolog proof rules. These are `cool.rules` and the generic

`stack.rules` (no special predicates or lemmas are needed to handle functions, only a particular way to organize the locinvs and coverage proofs). Finally these rules/lemmas need to be proven, for which we use the Coq interactive proof development system. These are `cool.v` and `stack.v`.

The Cool extension (and its Coq script) will need to be extended in order to handle the run-time system.

On the trusted side, the main module is `Openver` which encodes the notion of locinv and the algorithms for the director, the decoder, and the checking of coverage proofs. `Proof` is the proof checker for the first-order proofs, produced either by hand or by Kettle. `SAL` encodes the simple assembly language which is verified, while `MIPSparse` is a translator from MIPS into SAL. Note that the table does not show the code used to check the Coq proofs. We would like to be able to expand the simple `Proof` proof checker to handle them, but currently we are using Coq to check them. The type-checking Coq kernel contains approximately 8000 lines of OCaml code, but even the kernel of Coq handles much more than we need. For comparison I include the size of the Coq script for the soundness of the Open Verifier algorithm (Section 2.6); this includes the general notions of locinv, decoder, and coverage, but not their particular implementations.

The size of the Open Verifier's trusted code base (TCB) compares favorably to an estimated 15,000 to 25,000 lines of code in the TCB of Touchstone. Although Touchstone works with a Java compiler, which is more complex than our Cool compilers, no more trusted code would be required for the Open Verifier to handle Java (and in fact even the amount of extra untrusted code needed should not be excessive). The TCB of an FPCC system is estimated in [4] to require between 500 and 1000 lines of logic about the encoding of instructions as machine integers, plus about 600 lines of logic encoding the semantics of the instructions and the safety policy, and finally the executable code of a proof checker for higher-order logic. We believe that the trusted code base of the Open Verifier is small enough that it can be understood and checked by hand in a convincing way, and that we have used more trusted code in the right way, not to sweep difficult proofs under the rug, but rather to permit developers of certifying compilers/verifiers to concentrate less on the tedious and more on the challenging and interesting aspects, and overall to make their job more feasible.

5.1.2 Flexibility

The Open Verifier has obvious advantages in flexibility compared with standard virtual machines, or even with Touchstone. Whereas programs written for, say, the Java virtual machine have to be constrained according to allowable bytecodes, an Open Verifier client can use any compilation strategies or optimizations whatsoever, as long as they can in each instance be proven safe. This also saves the cost of interpretation or even just-in-time compilation.

In principle, the Open Verifier should be substantially flexible with respect to the source language being verified. It is a matter for future experiment, however, to tell in practice whether more complicated languages (such as functional languages like ML) can be feasible handled. Preliminary results with verifying TAL indicate that more powerful type systems can be handled just as well as Cool's Java-like type system.

It is worth noting in this respect that we advocate a particular restricted use of logic in the Open Verifier, such that the assumptions of local invariants are first-order, and any higher-order reasoning is restricted to the particular forms of progress continuations and coverage proofs. This restricts the kinds of *proof* techniques which work with the Open Verifier; for example the handling of recursive types in semantic FPCC as in [6] cannot be used. However (considering our work with Cool and our preliminary success with TAL) this doesn't seem to restrict the kinds of *programs* which can be handled, and the kinds of proof techniques we advocate seem considerably simpler.

5.1.3 Scalability and Usability

With the Open Verifier the main efficiency concern is efficiency of verification. Similar to PCC, efficiency of execution is unimportant because executable code is used directly. The architecture of the Open Verifiers offers an advantage over PCC, in that transmission is also not much of an issue, because an executable verifier is transmitted so that proof creation occurs on-site; this means in particular that we don't need to worry so much about how most efficiently to structure and transmit the proofs.

To measure the efficiency of verification we have used the conventional verifier as a baseline. Again, the conventional verifier is structured similar to most bytecode verifiers; so this is analogous to measuring the cost of going from an ordinary Java bytecode verifier to a proof-generating, certifying bytecode

verifier. At the time of writing the Cool extension runs 20 times slower than the Cool conventional verifier. We hope to do much better, and in fact have only just begun to work on improving performance; for instance, a single small change in performing substitutions yielded a factor of 2 improvement. Many other such improvements may be possible.

A more important measure is the ease of developing extensions. Again, the appropriate baseline is a conventional verifier; if someone can write a straightforward conventional verifier, how much more work is required to wrap it into an Open Verifier extension? Figure 5.1 gives a rough idea of the number of lines of code and proof required. There is a fair amount of effort, but we believe it to be manageable. Also, much of the implementation effort is common to different extensions; we have already succeeded in separating out certain elements into fully reusable modules (such as the handling of stacks and function calls). In the process of working on various extensions, we have developed some software engineering techniques which have proved very helpful. In particular we have a graphical user interface which works like an integrated debugger, only via verification steps rather than execution steps. It would be difficult for me to overstate the usefulness of such a tool in developing extensions.

5.2 Related Work

5.2.1 Virtual Machines

I have already discussed relationships with work on virtual machines in the Introduction. Here I just want to mention another line of research on making a more trustworthy virtual machine, which is to actually verify the bytecode verifier (a survey of such work is in [23]). It is substantially easier to certify each individual verification, as is done by the Open Verifier. Moreover, verified bytecode verifiers are really formally verified *algorithms* for bytecode verification; it is another step to to ensure that the implementation is correct with respect to the algorithm. One approach to this step is to automatically extract the bytecode verifier from the proof that its algorithm is correct [8]. Proof extraction technology is still relatively primitive; also, if one certifies individual verifications instead, one is more free to introduce optimizations into the verifier code.

5.2.2 Typed Assembly Language (TAL)

Typed assembly language (TAL, [28, 27, 26, 15]) is a another approach to certified code. Instead of using an intermediate language of bytecodes, TAL provides a way to use the target machine language with a powerful type system. In this sense it can be considered an approach to a generic virtual machine, where the "intermediate" language is in fact the target language.

Compilers could be built from many source languages to TAL, but as with other approaches to generic virtual machines, this requires that the type system of the source language and compilation strategy be coerced into TAL's type system. The Open Verifier framework allows particular extensions to be tailored to particular requirements. (The work of [15] proposes a model for a client-specifiable type system for TAL, using a framework similar to that of syntactic FPCC, described below.) Also, TAL is restricted to the notion of typing, whereas Open Verifier extensions can work with more general logical assertions including typing. In some cases this may be only a superficial difference (such as our preference for an equality predicate over singleton types), but we think that not being limited to types may allow new and easier solutions to certain verification problems.

5.2.3 Foundational Proof-Carrying Code (FPCC)

Superficially, the Open Verifier is very different from work on foundational proof-carrying code (FPCC, see [5, 25, 6, 4, 1] and [19, 39, 43]). First, with the Open Verifier code need not carry a *proof* at all but rather an executable *verifier*. I have already discussed the advantages of this approach. Second, the Open Verifier is not emphatically foundational. We are willing to include into the trusted code base (TCB) certain components, such as the director and decoder, which are common to the kinds of verifiers we hope to handle.

Nonetheless, there are important commonalities with FPCC. Even though a verifier is sent rather than an explicit proof, the verifier has to provide many parts of the proof in order for its results to be trusted. Also, those parts of the foundational proof, which in the Open Verifier are part of the TCB, are in fact the least difficult-to-trust parts—I might say the least interesting. For instance, the Open Verifier works on assembly code and uses the decoder module to determine, via a simple strongest-postcondition calculation, the result of executing each instruction. In a strict FPCC system, the TCB would in-

stead include a complete description *in logic* of which machine integers specify which instructions, and what state transitions are affected by the instructions; then the foundational proof much refer to these axioms at each step in order to show that a particular state transition is taking place. There are indeed interesting questions about how best to do this in FPCC (see [25]), nonetheless the Open Verifier code does not seem vastly more difficult to trust than this. Similarly, the director module of the Open Verifier encodes the main induction common to all foundational proofs. The difficult-to-trust parts of the proof are program-specific or type-system-specific, and these have to be given explicitly also using the Open Verifier.

In principle, an Open Verifier verification could be built up into an FPCC proof, using some formalization of the decoder and director (such as the Coq formalization of Section 2.6 which is already available). We do not propose to do this, however; we feel that such concerns as ease of extension development are more important for future research.

If we do consider the Open Verifier alongside FPCC efforts, there are interesting points to be made about the proof development approaches, what might be called "proof engineering". Further research is required to work out precisely similarities and differences; here I just want to give some highlights.

All FPCC efforts share with the Open Verifier the large structure of a co-induction, where some invariant (in the Open Verifier, this would be the disjunction of all the local invariants) implies that safe progress is possible to a state which still satisfies the invariant. The main difference between the two main FPCC schools has been considered the "semantic" and "syntactic" style. I propose that a better way of looking at the difference is "local" and "global", as follows. The original "semantic" FPCC uses invariants which are local assertions about individual points in the execution; there is a specified meaning given to "in machine state s, value v has type t". Syntactic FPCC uses a global notion of well-typedness; there is not particular local meaning to some value having a type, only what can be derived from the (syntactic) proof that the entire machine state is well-typed.

Our extensions for the Open Verifier, such as the Cool extension, also use the "local" approach. We define a particular meaning for the typing predicate local to a particular state. However, unlike original semantic FPCC, we have used always a particular extension-specific *syntactic* family of types and intensional typing of recursive types (Section 4.2.1.1). The original semantic FPCC has tried to remain close to the spirit of denotational semantics, but has had to

introduce various syntactic, intensional, and operational elements in order to make progress. We believe that by embracing such elements, we have had an easier time constructing our proofs.

It is more difficult to compare with syntactic FPCC, which seems to use a fundamentally different "global" technique. However I note that this group proposes to use the local approach to handle run-time library functions [43]. There may be benefits to using the same framework for both the run-time and for those parts of the program which obey the type system, as we do with the Open Verifier.

Part of the reason, that our proof construction has required less effort, may be that we have been concentrating on building extensions specific to a single source language and compilation strategy, whereas the FPCC groups have concentrated more on a general approach; for instance, the semantic FPCC method doesn't ever restrict to a single specific type system, instead using an entirely general notion of type which, once the necessary proofs are complete, could in principle be used to handle a very broad class of compilers. I note that the final step, of fully expressing a given language and compilation strategy in terms of the general notion of types, may not be trivial; there is not yet any published work on this aspect of "productionizing" FPCC. For the Open Verifier, improving proof re-use is important for future research; at the same time, the ability to tailor specific extensions to specific compilers (or even specific programs) can be a very important advantage over efforts which concentrate on a single general system.

It is also worth noting that we expect to be able to verify the output of existing compilers, rather than having to write a new certifying compiler from scratch.

Finally, I wish to note that we have developed some very useful software engineering techniques for extension development. Our approach allows us to "play" with the structure of the proof very easily. We can introduce lemmas and see very quickly what they fix (or break) in the verification of test cases; if at a later stage the lemma seems difficult to prove, we can tweak its statement and check the repercussions very easily. This has contributed greatly to the ease of extension development.

5.3 Future Research

I close this dissertation with a list of some avenues for continuing research on the Open Verifier.

A complete Cool verifier. I think that the first order of business should be to complete the Cool extension, so that there are no unanswered questions about potential difficulties with verifying the initial state and the run-time system. It seems likely that tedium rather than difficulty is the main obstacle, but it is important to be sure. Although we have been working with a version of Cool without garbage collection, verifying the garbage collector would be a great advance.

There are also performance issues to be addressed in order to obtain a production-quality verifier.

Capacity for re-use among extensions. It is clear that there are many commonalities among extension, both in code and in proof. Currently though we often can only re-use heuristics, or at best re-use pieces by a cut-and-paste process. There could be substantial work done to modularize the process, in particularly at the level of the interactive proofs, which are probably the most labor-intensive part of the whole effort. For instance, it would be valuable if we could make our proof strategy explicit enough to modularize the type system, so that any type system for which the appropriate lemmas could be proved can be easily used with our invariants.

There is a tension here between the desire for re-use, and the fact that one of the main benefits of the Open Verifier approach is that the extension can be tailored specifically to the given program (or more generally, compilation strategy).

With re-use in mind, it would be good to have experimental validation of the ease of writing extensions. For instance, now that our techniques are relatively mature and we have a working Cool extension, how much more difficult is it to build an extension for the very similar Java language, perhaps using the Touchstone compiler?

Other extensions. Since the Java-like Cool type system is relatively weak, it is useful to have experiments showing that the Open Verifier can handle much

stronger type systems. An extension to handle programs compiled to TAL is also close to completion.

Logical questions. There are some interesting purely logical questions raised by the Open Verifier. For instance, we have managed to layer the proof-generation effort so that per-program proofs are (mostly) in Horn logic, per-compiler proofs in first-order logic, and higher-order features are mostly restricted to once-only proofs. Can this be made explicit, to some claim about the logical strength required to establish memory safety for certain classes of programs?

Similarly it would be worthwhile to see if the proof strategy we have used with extensions can be applied more generally in other areas of program analysis.

Stronger safety policies. Although the Open Verifier framework can in principal be used with a broad class of safety policies, in practice our examples have all been about memory safety. To what extent can the same techniques be adapted to establish stronger properties?

Bibliography

[1] A. J. Ahmed, A. W. Appel, and R. Virga. A stratified semantics of general references embeddable in higher-order logic. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS 2002)*, pages 75–86, July 2002.

[2] A. Aiken. A Tour of the Cool Support Code, 1996.

[3] A. Aiken. The Cool Reference Manual, 2000.

[4] A. W. Appel. Foundational proof-carrying code. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[5] A. W. Appel and A. P. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-00)*, pages 243–253. ACM Press, Jan. 2000.

[6] A. W. Appel and D. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, Sept. 2001.

[7] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*, volume 34(1) of *ACM SIGPLAN Notices*, pages 129–140. ACM Press, June 1999.

[8] Y. Bertot. A Coq formalization of a type checker for object initialization in the Java virtual machine, 2000. Research report 4047, INRIA, 2000. Also published in the proceedings of CAV'01.

[9] P. Bothner. Kawa: Compiling Scheme to Java. In *40th Anniversary of Lisp Conference: Lisp in the Mainstream*, Berkeley, California, Nov. 1998.

[10] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java™ programming language. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 183–200. ACM Press, Oct. 18–22 1998.

[11] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 95–107. ACM Press, May 2000.

[12] Coq Development Team. The Coq proof assistant reference manual, version 7.4. http://coq.inria.fr, Jan. 2003.

[13] T. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *Proc. European Conf. on Computer Algebra (EUROCAL'85), LNCS 203*, pages 151–184. Springer-Verlag, 1985.

[14] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[15] K. Crary. Toward a foundational typed assembly language. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages (POPL-03)*, volume 38(1) of *ACM SIGPLAN Notices*, pages 198–212. ACM Press, Jan. 15–17 2003.

[16] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-01)*, pages 248–260, London, United Kingdom, Jan. 2001.

[17] J. Gough. *Compiling for the .NET Common Language Runtime*. .NET series. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[18] K. J. Gough and D. Corney. Evaluating the Java virtual machine as a target for languages other than Java. In *Joint Modula Languages Conference*, Sept. 2000.

[19] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Proc. Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS'02)*, pages 89–100, Copenhagen, Denmark, July 2002. IEEE.

[20] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36(5) of *ACM SIGPLAN Notices*, pages 1–12. ACM Press, June 20–22 2001.

[21] J. R. Larus. Assemblers, linkers, and the SPIM simulator. In *Computer Organization and Design: The Hardware/Software Interface*, Appendix A. Morgan Kaufmann, 1994.

[22] C. League. Touchstone soundness bug report. Personal communication with George Necula, Oct. 2001.

[23] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 2003. To appear in the special issue on Java bytecode verification.

[24] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.

[25] N. G. Michael and A. W. Appel. Machine instruction syntax and semantics in higher-order logic. In *Proceedings of the 17th International Conference on Automated Deduction*, pages 7–24. Springer-Verlag, June 2000.

[26] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic. TALx86: A realistic typed assembly language. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, 1999.

[27] G. Morrisett, K. Crary, N. Glew, and D. Walker. Stack-based typed assembly language. In *Second International Workshop on Types in Compilation*,

pages 95–117, Kyoto, Mar. 1998. Published in Xavier Leroy and Atsushi Ohori, editors, *Lecture Notes in Computer Science*, volume 1473, pages 28–52. Springer-Verlag, 1998.

[28] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[29] G. C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, Jan. 1997.

[30] G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Sept. 1998.

[31] G. C. Necula. Translation validation for an optimizing compiler. In *ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI)*, pages 83–94, Vancouver, BC, Canada, June18–21 2000. ACM SIGPLAN.

[32] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Second Symposium on Operating Systems Design and Implementations*, pages 229–243. Usenix, Oct. 1996.

[33] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Thirteenth Annual Symposium on Logic in Computer Science*, pages 93–104, Indianapolis, June 1998. IEEE Computer Society Press.

[34] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *The 29th Annual ACM Symposium on Principles of Programming Languages*, pages 128–139. ACM, Jan. 2002.

[35] G. C. Necula and S. P. Rahul. Oracle-based checking of untrusted programs. In *The 28th Annual ACM Symposium on Principles of Programming Languages*, pages 142–154. ACM Press, Jan. 2001.

[36] G. C. Necula and R. R. Schneck. Proof-carrying code with untrusted proof rules. In M. Okada, editor, *Software Security – Theories and Systems. Mext-NSF-JSPS International Symposium, ISSS 2002.*, Lecture Notes in Computer Science, pages 283–289, 2003.

[37] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In B. Steffen, editor, *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98*, volume LNCS 1384, pages 151–166. Springer, 1998.

[38] F. B. Schneider. Enforceable security policies. Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department, Sept. 1998.

[39] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL-02)*, pages 217–232, Portland, Oregon, Jan. 16–18, 2002.

[40] D. Syme. ILX: Extending the .NET common IL for functional language interoperability. *Electronic Notes in Theoretical Computer Science*, 59(1), 2001.

[41] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles*, pages 203–216. ACM, Dec. 1993.

[42] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, Nov. 1997.

[43] D. Yu, N. A. Hamid, and Z. Shao. Building certified libraries for PCC: Dynamic storage allocation. In *Proc. 2003 European Symposium on Programming (ESOP'03)*, April 2003.